# Chapter 11 - Automating Your Automation - Ansible Tower and CI/CD

At this point, you should be able to convert almost any bit of your infrastructure's configuration into Ansible playbooks, roles, and inventories. And before deploying any infrastructure changes, you should test the changes in a non-production environment (just like you would with application releases). Manually running a playbook that configures your entire infrastructure, then making sure it does what you expect, is a good start towards order and stability.

Since all your infrastructure is defined in code, you can start automating all the aspects of infrastructure deployment, and even run unit, functional, and integration tests on your infrastructure, just like you do for your applications.

This section will cover different levels of infrastructure automation and testing, and highlight tools and techniques you can use to automate and streamline infrastructure operations.

## Ansible Tower

All the examples in this book use Ansible's CLI to run playbooks and report back the results. For smaller teams, especially when everyone on the team is well-versed in how to use Ansible, YAML syntax, and security best practices, using the CLI is a sustainable approach.

But for many organizations, basic CLI use is inadequate:

- The business needs detailed reporting of infrastructure deployments and failures, especially for audit purposes.

- Team-based infrastructure management requires varying levels of involvement in playbook management, inventory management, and key and password access.
- A thorough visual overview of the current and historical playbook runs and server health helps identify potential issues before they affect the bottom line.
- Playbook scheduling ensures infrastructure remains in a known state.

Ansible Tower checks off these items—and many more—and provides a great mechanism for team-based Ansible usage. The product is currently free for teams managing ten or fewer servers (it's basically an 'unlimited trial' mode), and has flexible pricing for teams managing dozens to thousands of servers.

While this book includes a brief overview of Tower, it is highly recommended you read through Ansible, Inc's extensive Tower User Guide[137], which includes details this book won't be covering such as LDAP integration and multiple-team playbook management workflows.

## Getting and Installing Ansible Tower

Ansible has a very thorough Ansible Tower User Guide[138], which details the installation and configuration of Ansible Tower. For the purposes of this chapter, since we just want to download and try out Tower locally, we are going to use Ansible's official Vagrant box to quickly build an Ansible Tower VM.

Make sure you have Vagrant[139] and VirtualBox[140] installed, then create a directory (e.g. `tower`) and do the following within the directory:

[137]http://releases.ansible.com/ansible-tower/docs/tower_user_guide-latest.pdf
[138]http://releases.ansible.com/ansible-tower/docs/tower_user_guide-latest.pdf
[139]https://www.vagrantup.com/downloads.html
[140]https://www.virtualbox.org/wiki/Downloads

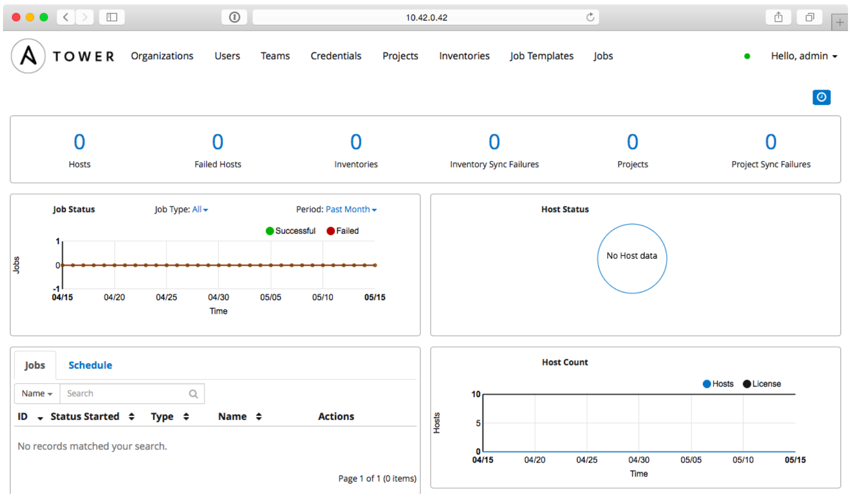
```
# Create a new Vagrantfile using the Tower base box from Ansible.
$ vagrant init tower http://vms.ansible.com/ansible-tower-2.3.1-\
virtualbox.box

# Build the Tower VM.
$ vagrant up

# Log into the VM (Tower will display connection information).
$ vagrant ssh
```

The above installation instructions and Vagrant box come from a blog post on Ansible's official blog, Ansible Tower and Vagrant[141].

Visit the URL provided by the login welcome message (e.g. `https://10.42.0.42/`), and after confirming a security exception for the Ansible Tower certificate, login with the credentials from step 3.

At this point, you will need to register a free trial license of Ansible Tower following the instructions on the screen. The free trial allows you to use all of Tower's features for up to 10 servers, and is great for experimenting and seeing how Tower fits into your workflow. After you get the license (it's a block of JSON which you paste into the license field), you should get to Tower's default dashboard page:

[141]http://www.ansible.com/blog/ansible-vagrant



Ansible Tower's Dashboard

## Using Ansible Tower

Ansible Tower is centered around the idea of organizing *Projects* (which run your playbooks via *Jobs*) and *Inventories* (which describe the servers on which your playbooks should be run) inside of *Organizations*. *Organizations* are then set up with different levels of access based on *Users* and *Credentials* grouped in different *Teams*. It's a little overwhelming at first, but once the initial structure is configured, you'll see the power and flexibility Tower's Project workflow affords.

Let's get started with our first project!

The first step is to make sure you have a test playbook you can run using Ansible Tower. Generally, your playbooks should be stored in a source code repository (e.g. Git or Subversion), with Tower configured to check out the latest version of the playbook from the repository and run it. For this example, however, we will create a playbook in Tower's default `projects` directory located in `/var/lib/awx/projects`:

1. Log into the Tower VM: `vagrant ssh`
2. Switch to the `awx` user: `sudo su - awx`
3. Go to Tower's default `projects` directory: `cd /var/lib/awx/projects`
4. Create a new project directory: `mkdir ansible-for-devops && cd ansible-for-devops`

5. Create a new playbook file, `main.yml`, within the new directory, with the following contents:

```
---
- hosts: all
  gather_facts: no
  connection: local

  tasks:
    - name: Check the date on the server.
      command: date
```

Switch back to your web browser and get everything set up to run the test playbook inside Ansible Tower's web UI:

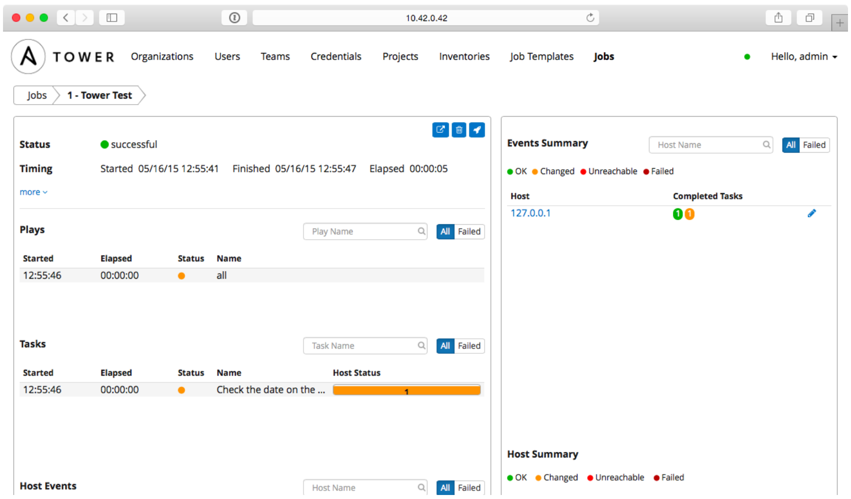
1. Create a new *Organization*, called 'Ansible for DevOps'.
2. Add a new User to the Organization, named John Doe, with the username `johndoe` and password `johndoe1234`.
3. Create a new *Team*, called 'DevOps Engineers', in the 'Ansible for DevOps' Organization.
4. Under the Team's Credentials section, add in SSH credentials by selecting 'Machine' for the Credential type, and setting 'Name' to `Vagrant`, 'Type' to `Machine`, 'SSH Username' to `vagrant`, and 'SSH Password' to `vagrant`.
5. Under the Team's Projects section, add a new *Project*. Set the 'Name' to `Tower Test`, 'Organization' to `Ansible for DevOps`, 'SCM Type' to `Manual`, and 'Playbook Directory' to `ansible-for-devops` (Tower automatically detects all folders placed inside `/var/lib/awx/projects`, but you could also use an alternate Project Base Path if you want to store projects elsewhere).
6. Under the Inventories section, add an *Inventory*. Set the 'Name' to `Tower Local`, and 'Organization' set to `Ansible for DevOps`. Once the inventory is saved: 1. Add a 'Group' with the Name `localhost`. Click on the group once it's saved. 2. Add a 'Host' with the Host Name `127.0.0.1`.

New *Credentials* have a somewhat dizzying array of options, and offer login and API key support for a variety of services, like SSH, AWS, Rackspace, VMWare vCenter, and SCM systems. If you can login to a system, Tower likely supports the login mechanism!

Now that we have all the structure for running playbooks configured, we need only create a *Job Template* to run the playbook on the localhost and see whether we've succeeded. Click on 'Job Templates', and create a new Job Template with the following configuration:

- Name: `Tower Test`
- Inventory: `Tower Local`
- Project: `Tower Test`
- Playbook: `main.yml`
- Machine Credential: `Vagrant`

Save the Job Template, then click the small Rocketship button to start a job using the template. You'll be redirected to a Job status page, which provides live updates of the job status, and then a summary of the playbook run when complete:



Status ● successful

Timing Started 05/16/15 12:55:41 Finished 05/16/15 12:55:47 Elapsed 00:00:05

Plays

| Started | Elapsed | Status | Name |
|---|---|---|---|
| 12:55:46 | 00:00:00 | ● | all |

Tasks

| Started | Elapsed | Status | Name | Host Status |
|---|---|---|---|---|
| 12:55:46 | 00:00:00 | ● | Check the date on the ... | 1 |

Host Events

Events Summary

● OK ● Changed ● Unreachable ● Failed

| Host | Completed Tasks |
|---|---|
| 127.0.0.1 | 1 1 |

Host Summary

● OK ● Changed ● Unreachable ● Failed

**Tower Test job completed successfully!**

You can view the playbook run's standard output in real-time (or review it after the fact) with the 'View standard out' button. You can also stop a running job, delete a job's record, or relaunch a job with the same parameters using the respective buttons on the job's page.

The job's dashboard page is very useful for giving an overview of how many hosts were successful, how many tasks resulted in changes, and the timing of the different parts of the playbook run.

## Other Tower Features of Note

In our walkthrough above, we used Tower to run a playbook on the local server; setting up Tower to run playbooks on real-world infastructure or other local VMs is just as easy, and the tools Ansible Tower provides are very handy, especially when working in larger team environments.

This book won't walk through the entirety of Ansible Tower's documentation, but a few other features you should try out include:

- Setting up scheduled Job runs (especially with the 'Check' option instead of 'Run') for CI/CD.
- Integrating user accounts and Teams with LDAP users and groups for automatic team-based project management.
- Setting different levels of permissions for Users and Teams so certain users can only edit, run, or view certain jobs within an Organization.
- Configuring Ansible Vault credentials to easily and automatically use Vault-protected variables in your playbooks.
- Setting up Provisioning Callbacks so newly-provisioned servers can self-provision via a URL per Job Template.
- Surveys, which allow users to add extra information based on a 'Survey' of questions per job run.
- Inventory Scripts, which allow you to build inventory dynamically.
- Built-in Munin monitoring (to monitor the Tower server), available with the same admin credentials at `https://[tower-hostname]/munin`.

Ansible Tower continues to improve rapidly, and is one of the best ways to run Ansible Playbooks from a central CI/CD-style server with team-based access and extremely detailed live and historical status reporting.

## Tower Alternatives

Ansible Tower is purpose-built for use with Ansible playbooks, but there are many other ways to run playbooks on your servers with a solid workflow. If price is a major concern, and you don't need all the bells and whistles Tower provides, you can use other popular tools like Jenkins[142], Rundeck[143], or Go CI[144].

All these tools provide flexibility and security for running Ansible Playbooks, and each one requires a different amount of setup and configuration before it will work well for common usage scenarios. One of the most popular and long-standing CI tools is Jenkins, so we'll explore how to configure a similar Playbook run in Jenkins next.

## Jenkins CI

Jenkins is a Java-based open source continuous integration tool. It was forked from the Hudson project in 2011, but has a long history as a robust build tool for almost any software project.

Jenkins is easy to install and configure, with the Java SDK as its only requirement. Jenkins runs on any modern OS, but for the purposes of this demonstration, we'll build a local VM using Vagrant, install Jenkins inside the VM using Ansible, then use Jenkins to run an Ansible playbook.

### Build a local Jenkins server with Ansible

Create a new directory for the Jenkins VM named `jenkins`. Inside the directory, create a `Vagrantfile` to describe the machine and the Ansible provisioning to Vagrant, with the following contents:

[142]http://jenkins-ci.org/
[143]http://rundeck.org/
[144]http://www.go.cd/

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "geerlingguy/ubuntu1604"
  config.vm.hostname = "jenkins.test"
  config.vm.network :private_network, ip: "192.168.76.76"
  config.ssh.insert_key = false

  config.vm.provider :virtualbox do |v|
    v.memory = 512
  end

  # Ansible provisioning.
  config.vm.provision "ansible" do |ansible|
    ansible.playbook = "provision.yml"
    ansible.become = true
  end
end
```

This Vagrantfile will create a new VM running Ubuntu 16.04, with the IP address `192.168.76.76` and the hostname `jenkins.test`. Go ahead and add an entry for `192.168.76.76  jenkins.test` to your hosts file, and then create a new `provision.yml` playbook so Vagrant can run it with Ansible (as described in the `config.vm.provision` block in the Vagrantfile). Put the following in the `provision.yml` file:

```
---
- hosts: all

  vars:
    firewall_allowed_tcp_ports:
      - "22"
      - "8080"
    jenkins_plugins:
      - ansicolor

  roles:
    - geerlingguy.firewall
    - geerlingguy.ansible
    - geerlingguy.java
    - geerlingguy.jenkins
```

This playbook uses a set of roles from Ansible Galaxy to install all the required components for our Jenkins CI server. To make sure you have all the required roles installed on your host machine, add a `requirements.yml` file in the `jenkins` folder, containing all the roles being used in the playbook:

```
---
- src: geerlingguy.firewall
- src: geerlingguy.ansible
- src: geerlingguy.java
- src: geerlingguy.jenkins
```

The `geerlingguy.ansible` role installs Ansible on the VM, so Jenkins can run Ansible playbooks and ad-hoc commands. The `geerlingguy.java` role is a dependency of `geerlingguy.jenkins`, and the `geerlingguy.firewall` role configures a firewall to limit access on ports besides 22 (for SSH) and 8080 (Jenkins' default port).

Finally, we tell the `geerlingguy.jenkins` role a set of plugins to install through the `jenkins_plugins` variable; in this case, we just want the `ansicolor` plugin, which gives us full color display in Jenkins' console logs (so our Ansible playbook output is easier to read).

There is an official Ansible plugin for Jenkins[145] which can be used to run Ansible Ad-Hoc tasks and Playbooks, and may help you integrate Ansible and Jenkins more easily.

To build the VM and run the playbook, do the following (inside the `jenkins` folder):

1. Run `ansible-galaxy install -r requirements.yml` to install the required roles.

[145]https://wiki.jenkins-ci.org/display/JENKINS/Ansible+Plugin

2. Run `vagrant up` to build the VM and install and configure Jenkins.

After a few minutes, the provisioning should complete, and you should be able to access Jenkins at `http://jenkins.test:8080/` (if you configured the hostname in your hosts file).

## Create an Ansible playbook on the Jenkins server

It's preferred to keep your playbooks and server configuration in a code repository (e.g. Git or SVN), but for simplicity's sake, this example requires a playbook stored locally on the Jenkins server, similar to the earlier Ansible Tower example.

1. Log into the Jenkins VM: `vagrant ssh`
2. Go to the `/opt` directory: `cd /opt`
3. Create a new project directory: `sudo mkdir ansible-for-devops && cd ansible-for-devops`
4. Create a new playbook file, `main.yml`, within the new directory, with the following contents (use sudo to create the file, e.g. `sudo vi main.yml`):

```
---
- hosts: 127.0.0.1
  gather_facts: no
  connection: local

  tasks:
    - name: Check the date on the server.
      command: date
```

If you want, test the playbook while you're logged in: `ansible-playbook main.yml`.

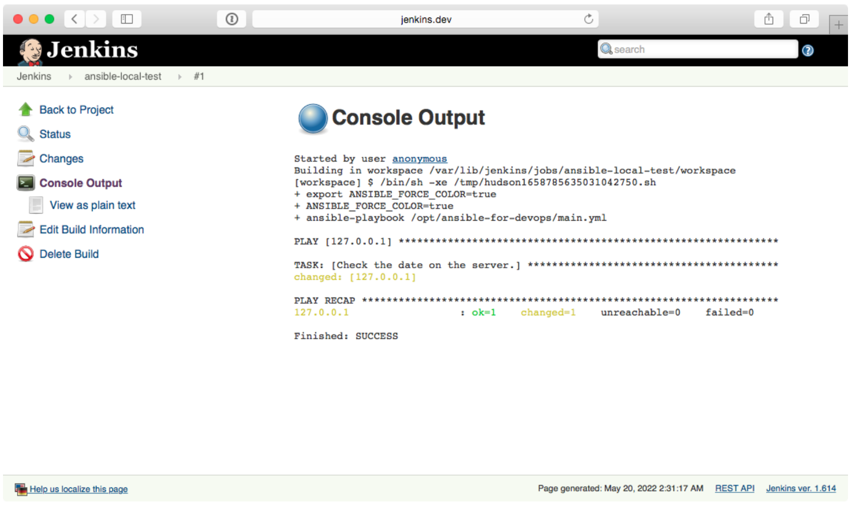
## Create a Jenkins job to run an Ansible Playbook

With Jenkins running, configure a Jenkins job to run a playbook on the local server with Ansible. Visit `http://jenkins.test:8080/`, and once the page loads, click the 'New Item' link to create a new 'Freestyle project' with a title 'ansible-local-test'. Click 'OK' and when configuring the job, and set the following configuration:

- Under ‘Build Environment’, check the ‘Color ANSI Console Output’ option. This allows Ansible’s helpful colored output to pass through the Jenkins console, so it is easier to read during and after the run.
- Under ‘Build’, click ‘Add Build Step’, then choose ‘Execute shell’. In the ‘Command’ field, add the following code, which will run the local Ansible playbook:

```
# Force Ansible to output jobs with color.
export ANSIBLE_FORCE_COLOR=true

# Run the local test playbook.
ansible-playbook /opt/ansible-for-devops/main.yml
```

Click ‘Save’ to save the ‘Ansible Local Test’ job, and on the project’s page, click the ‘Build Now’ link to start a build. After a few seconds, you should see a new item in the ‘Build History’ block. Click on the (hopefully) blue circle to the left of ‘#1’, and it will take you to the console output of the job. It should look something like this:



```
Started by user anonymous
Building in workspace /var/lib/jenkins/jobs/ansible-local-test/workspace
[workspace] $ /bin/sh -xe /tmp/hudson1658785635031042750.sh
+ export ANSIBLE_FORCE_COLOR=true
+ ANSIBLE_FORCE_COLOR=true
+ ansible-playbook /opt/ansible-for-devops/main.yml

PLAY [127.0.0.1] ***************************************************************

TASK: [Check the date on the server.] *****************************************
changed: [127.0.0.1]

PLAY RECAP *********************************************************************
127.0.0.1                  : ok=1    changed=1    unreachable=0    failed=0

Finished: SUCCESS
```

**Jenkins job completed successfully!**

This is a basic example, but hopefully it’s enough to show you how easy it is to get at least some of your baseline CI/CD automation done using a free and open source tool. Most of the more difficult aspects of managing infrastructure through

Jenkins surrounds the ability to manage SSH keys, certificates, and other credentials through Jenkins, but there is already plenty of documentation surrounding these things elsewhere online and in Jenkins documentation, so this will be left as an exercise for the reader.

The rest of this chapter focuses on ways to test and debug your playbooks and your infrastructure as a whole, and while many examples use Travis CI or plain command line options, anything you see can be automated with Jenkins jobs!

## Unit, Integration, and Functional Testing

When determining how you should test your infrastructure, you need to understand the different kinds of testing, and then determine the kinds of testing on which you should focus more effort.

*Unit* testing, when applied to applications, is testing of the smallest units of code (usually functions or class methods). In Ansible, unit testing would typically apply to individual playbooks. You could run individual playbooks in an isolated environment, but it's often not worth the effort. What *is* worth your effort is at least checking the playbook syntax, to make sure you didn't just commit a YAML file that will break an entire deployment because of a missing quotation mark, or a whitespace issue!

*Integration* testing, which is definitely more valuable when it comes to Ansible, is the testing of small groupings of individual units of code, to make sure they work correctly together. Breaking your infrastructure definition into many task-specific roles and playbooks allows you to do this; if you've structured your playbooks so they have no or limited dependencies, you could test each role individually in a fresh virtual machine, before you use the role as part of a full infrastructure deployment.

*Functional* testing involves the whole shebang. Basically, you set up a complete infrastructure environment, and then run tests against it to make sure *everything* was successfully installed, deployed, and configured. Ansible's own reporting is helpful in this kind of testing, and there are external tools available to test infrastructure even more deeply.

It is often possible to perform all the testing you need on your own local workstation, using Virtual Machines (as demonstrated in earlier chapters), using tools like VirtualBox or VMWare. And with most cloud services providing robust control APIs and hourly billing, it's inexpensive and just as fast to test directly on cloud instances mirroring your production infrastructure!

We'll begin with the most basic tests using Ansible, along with common debugging techniques, then progress to full-fledged functional testing methods with an automated process.

## Debugging and Asserting

For most playbooks, testing configuration changes and the result of commands being run as you go is all the testing you need. And having tests run *during your playbook runs* using some of Ansible's built-in utility modules means you have immediate assurance the system is in the state you want.

If at all possible, you should try to bake all simple test cases (e.g. comparison and state checks) into your playbooks directly. Ansible has three modules that simplify this process.

### The `debug` module

When actively developing an Ansible playbook, or even for historical logging purposes (e.g. if you're running Ansible playbooks using Tower or another CI system), it's often handy to print values of variables or output of certain commands during the playbook run.

For this purpose, Ansible has a `debug` module, which prints variables or messages during playbook execution.

As an extremely basic example, here are two of the ways I normally use debug while building a playbook:

```
- hosts: 127.0.0.1
  gather_facts: no
  connection: local

  tasks:
    - name: Register the output of the 'uptime' command.
      command: uptime
      register: system_uptime

    - name: Print the registered output of the 'uptime' command.
      debug: var=system_uptime.stdout

    - name: Print a simple message if a command resulted in a change.
      debug: msg="Command resulted in a change!"
      when: system_uptime.changed
```

Running this playbook gives the following output:

```
$ ansible-playbook debug.yml

PLAY [127.0.0.1] ****************************************************

TASK: [Register the output of the 'uptime' command.] ****************
changed: [127.0.0.1]

TASK: [Print the registered output of the 'uptime' command.] ********
ok: [127.0.0.1] => {
    "var": {
        "system_uptime.stdout":
            "15:01  up 15:18, 2 users, load averages: 1.23 1.33 1.42"
    }
}

TASK: [Print a simple message if a command resulted in a change.] ***
ok: [127.0.0.1] => {
    "msg": "Command resulted in a change!"
}

PLAY RECAP *********************************************************
127.0.0.1              : ok=3    changed=1    unreachable=0    failed=0
```

Debug messages are helpful when actively debugging a playbook or when you need extra verbosity in the playbook's output, but if you need to perform an explicit test on some variable, or bail out of a playbook for some reason, Ansible provides the `fail` module, and its more terse cousin, `assert`.

## The `fail` and `assert` modules

Both `fail` and `assert`, when triggered, will abort the playbook run, and the only difference is in the simplicity of their usage. To illustrate, let's look at an example:

```
- hosts: 127.0.0.1
  gather_facts: no
  connection: local

  vars:
    should_fail_via_fail: true
    should_fail_via_assert: false
    should_fail_via_complex_assert: false

  tasks:
    - name: Fail if conditions warrant a failure.
      fail: msg="There was an epic failure."
      when: should_fail_via_fail

    - name: Stop playbook if an assertion isn't validated.
      assert: that="should_fail_via_assert != true"

    - name: Assertions can have contain conditions.
      assert:
        that:
          - should_fail_via_fail != true
          - should_fail_via_assert != true
          - should_fail_via_complex_assert != true
```

Switch the boolean values of `should_fail_via_fail`, `should_fail_via_assert`, and `should_fail_via_complex_assert` to trigger each of the three `fail`/`assert` tasks, to see how they work.

For most test cases, `debug`, `fail`, and `assert` are all you need to ensure your infrastructure is in the correct state during a playbook run.

## Checking syntax and performing dry runs

Two checks you should include in an automated playbook testing workflow are `--syntax-check` (which checks the playbook syntax to find quoting, formatting, or whitespace errors) and `--check` (which will run your entire playbook in `check` mode.

Syntax checking is extremely straightforward, and only requires a few seconds for even larger, more complex playbooks with dozens or hundreds of includes. You should include an `ansible-playbook my-playbook.yml --syntax-check` in your basic CI tests, and it's best practice to run a syntax check in a pre-commit hook when developing playbooks.

Running a playbook in `check` mode is more involved, since Ansible runs the entire playbook on your live infrastructure, but without performing any changes. Instead, Ansible highlights tasks that *would've* resulted in a change to show what will happen when you *actually* run the playbook later.

This is helpful for two purposes:

1. To prevent 'configuration drift', where a server configuration may have drifted away from your coded configuration. This could happen due to human intervention or other factors. But it's good to discover configuration drift without forcefully changing it.
2. To make sure changes you make to a playbook that shouldn't break idempotency *don't*, in fact, break idempotency. For example, if you're changing a configuration file's structure, but with the goal of maintaining the same resulting file, running the playbook with `--check` alerts you when you might accidentally change the live file as a result of the playbook changes. Time to fix your playbook!

When using `--check` mode, certain tasks may need to be forced to run to ensure the playbook completes successfully: (e.g. a `command` task that registers variables used in later tasks). You can set `check_mode: no` to do this:

```
- name: A task that runs all the time, even in check mode.
  command: mytask --option1 --option2
  register: my_var
  check_mode: no
```

For even more detailed information about what changes would occur, add the `--diff` option, and Ansible will output changes that *would've* been made to your servers line-by-line. This option produces a lot of output if `check` mode makes a lot of changes, so use it conservatively unless you want to scroll through a lot of text!

You can add conditionals with `check_mode` just like you can with `when` clauses, though most of the time you will probably just use `yes` or `no`.

In addition to Ansible's `--syntax-check` and `--check` modes, you might be interested in also running Ansible Lint[146] on your playbooks. Ansible Lint allows you to check for deprecated syntax or inefficient task structures, and is highly configurable so you can set up the linting to follow the playbook standards you and your team choose.

## Automated testing on GitHub using Travis CI

Automated testing using a continuous integration tool like Travis CI (which is free for public projects and integrated very well with GitHub) allows you to run tests against Ansible playbooks or roles you have hosted on GitHub with every commit.

There are four main things to test when building and maintaining Ansible playbooks or roles:

1. The playbook or role's syntax (are all the .yml files formatted correctly?).

[146] https://github.com/willthames/ansible-lint

2. Whether the playbook or role will run through all the included tasks without failing.
3. The playbook or role's idempotence (if run again, it should not make any changes!).
4. The playbook or role's success (does the role do what it should be doing?).

The most important part is #4—the *functional* test—because what's the point of a playbook or role if it doesn't do what you want it to do (e.g. start a web server, configure a database, deploy an app, etc.)?

For the purposes of this example, we're going to make the following assumptions:

- You are testing an Ansible role (though this process applies just as well to testing an entire playbook).
- Your role's repository is hosted on GitHub.
- You are using Travis CI and it's enabled for your role's repository.

Note that you can apply the test setup detailed here to almost any SCM and CI tool (e.g. GitLab, Jenkins, Circle, etc.), with minor variations.

## Testing on multiple OSes with Docker

Travis CI provides a VM in which you can run your tests. You can choose between a few flavors of Linux or macOS, but there's not a lot of flexibility in terms of *infrastructure* testing, and Travis bakes in a lot of software by default (e.g. Ruby, Python, etc.).

Because we want to test our Ansible roles in as clean an environment as possible, we have two options:

1. Choose from one of the few Travis default OS environments and try to clean out all the existing software installs before running our tests.
2. Build our own clean test environments inside Travis using Docker containers and run tests in containers.

Historically, the first solution was easier to implement, but recent improvements in Travis's Docker support makes the second solution a better choice.

Because multi-OS, clean-slate tests are important to us, we will do the following for each test:

1. Start a fresh, minimal OS container for each OS our role supports.
2. Run our role inside the container (and then test idempotence and functionality).

For many of my roles and playbooks, I support the following OSes, therefore I maintain images on Docker Hub for the explicit purpose of testing Ansible roles and playbooks:

- CentOS 6[147]
- CentOS 7[148]
- Fedora 27[149]
- Fedora 29[150]
- Debian 8[151]
- Debian 9[152]
- Ubuntu 14.04[153]
- Ubuntu 16.04[154]
- Ubuntu 18.04[155]

The rest of this section will demonstrate how to test an example Ansible role against all these OSes with one simple Travis configuration file.

## Setting up the test

Create a new 'tests' directory in your role or project directory, and create a test playbook inside:

[147]https://hub.docker.com/r/geerlingguy/docker-centos6-ansible/
[148]https://hub.docker.com/r/geerlingguy/docker-centos7-ansible/
[149]https://hub.docker.com/r/geerlingguy/docker-fedora27-ansible/
[150]https://hub.docker.com/r/geerlingguy/docker-fedora29-ansible/
[151]https://hub.docker.com/r/geerlingguy/docker-debian8-ansible/
[152]https://hub.docker.com/r/geerlingguy/docker-debian9-ansible/
[153]https://hub.docker.com/r/geerlingguy/docker-ubuntu1404-ansible/
[154]https://hub.docker.com/r/geerlingguy/docker-ubuntu1604-ansible/
[155]https://hub.docker.com/r/geerlingguy/docker-ubuntu1804-ansible/

```
# Directory structure:
my_role/
  tests/
    test.yml <-- the test playbook
```

Inside `test.yml`, add:

```
---
- hosts: all

  roles:
    - role_under_test
```

In this playbook we tell Ansible to run our role on all hosts; since the playbook will run inside a Docker container with the option `--connection=local`, this basically means "run it on localhost". You can add `vars`, `vars_files`, `pre_tasks`, etc. if you need to adjust anything or prep the environment before your role runs, but I try to avoid overriding pre-packaged defaults, since they should ideally work across all environments—including barebones test environments.

The next step is to add a `.travis.yml` file to your role so Travis CI knows how to run your tests. Add the file to the root level of your role, and add the following scaffolding:

```
---
# We need sudo for some of the Docker commands.
sudo: required

env:
  # Provide a list of OSes we want to use for testing.

# Tell Travis to start Docker when it brings up an environment.
services:
  - docker

before_install:
  # Pull the image from Docker Hub for the OS under test.

script:
  # Start the container from the image and perform tests.

notifications:
  # Notify Ansible Galaxy when a role builds successfully.
```

This is a fairly standard Travis file layout, and if you want to dive deeper into how Travis works, read through the guide Customizing the Build[156]. Next, we need to fill in each section of the file, starting with the parts that control the Docker container lifecycle.

## Building Docker containers in Travis

The first thing we need to do is decide on which OSes we'd like to test. For my `geerlingguy.java`[157] role, I support CentOS, Fedora, Debian, and Ubuntu, so at a minimum I want to support the latest LTS release of each, and for CentOS and Ubuntu, the previous LTS release as well:

```
env:
  - distro: centos7
  - distro: centos6
  - distro: fedora24
  - distro: ubuntu1604
  - distro: ubuntu1404
  - distro: debian8
```

One other thing that needs to be configured per-OS is the init system. Because we're dealing with OSes that have a mixture of `systemd` and `sysv` init systems, we have to specify in Travis' environment the path to the init system to use, and any extra options that we need to pass to the `docker run` command to get the image in the right state for Ansible testing. So we'll add two variables for each distribution, `init` and `run_opts`:

[156]https://docs.travis-ci.com/user/customizing-the-build
[157]https://galaxy.ansible.com/geerlingguy/java/

```
env:
  - distro: centos7
    init: /usr/lib/systemd/systemd
    run_opts: "--privileged --volume=/sys/fs/cgroup:/sys/fs/cgroup:ro"
  - distro: centos6
    init: /sbin/init
    run_opts: ""
  - distro: fedora24
    init: /usr/lib/systemd/systemd
    run_opts: "--privileged --volume=/sys/fs/cgroup:/sys/fs/cgroup:ro"
  - distro: ubuntu1604
    init: /lib/systemd/systemd
    run_opts: "--privileged --volume=/sys/fs/cgroup:/sys/fs/cgroup:ro"
  - distro: ubuntu1404
    init: /sbin/init
    run_opts: ""
  - distro: debian8
    init: /lib/systemd/systemd
    run_opts: "--privileged --volume=/sys/fs/cgroup:/sys/fs/cgroup:ro"
```

Why use an `init` system in Docker? With Docker, it's preferable to either run apps directly (as 'PID 1') inside the container, or use a tool like Yelp's dumb-init[158] as a wrapper for your app. For our purposes, we're testing an Ansible role or playbook that could be run inside a container, but is also likely used on full VMs or bare-metal servers, where there will be a real init system controlling multiple internal processes. We want to emulate the real servers as closely as possible, therefore we set up a full init system (`systemd` or `sysv`) according to the OS.

Now that we've defined the OS distributions we want to test, and what init system we want Docker to call, we can manage the Docker container's lifecycle—we need to `pull` the image, `run` the image with our options, `exec` some commands to test our project, then `stop` the container once finished. Here's the basic structure, starting with the `before_install` step:

[158] https://github.com/Yelp/dumb-init

```
before_install:
  # Pull container from Docker Hub.
  - 'docker pull geerlingguy/docker-${distro}-ansible:latest'

script:
  # Create a random file to store the container ID.
  - container_id=$(mktemp)

  # Run container detached, with our role mounted inside.
  - 'docker run --detach --volume="${PWD}":/etc/ansible/roles/role_unde\
r_test:ro ${run_opts} geerlingguy/docker-${distro}-ansible:latest "${in\
it}" > "${container_id}"'

  # TODO - Test the Ansible role.
```

Let's run through these initial commands that set up our OS environment:

- `docker pull` (in `before_install`): This pulls down the appropriate OS image from Docker Hub with Ansible baked in. Note that `docker run` automatically pulls any images that don't already exist, but it's a best practice to always pull images prior to running them, in case the image is cached and there's a newer version.
- `container_id=$(mktemp)`: We need a file to store the container ID so we can perform operations on it later; we could also name the container, but we treat containers (like infrastructure) like cattle, not pets. So no names.
- `docker run`: This command starts a new container, with the Ansible role mounted inside (as a `--volume`), and uses the `run_opts` and `init` system described earlier in the `env:` section, then saves the container ID (which is output by Docker) into the temporary file we created in the previous step.

At this point, we have a Docker container running, and we can perform actions inside the container using `docker exec` just like we would if we were logged into a VM with the same OS.

For example, if you wanted to check up on disk space inside the container (assuming the `df` utility is present), you could run the command:

```
script:
  ...
  - 'docker exec "$(cat ${container_id})" df -h'
```

You can also run the command with `--tty`, which will allocate a pseudo-TTY, allowing things like colors to be passed through to Travis for prettier output.

Note: In Docker < 1.13, you have to set the `TERM` environment variable when using `docker exec` with the `--tty` option, like: `docker exec --tty "$(cat ${container_id})" env TERM=xterm df -h` (see: exec does not set TERM env when -t passed[159]). Also note that some older sysvinit scripts, when run through Ansible's `service` module, can cause strange issues when run inside a Docker container (see: service hangs the whole playbook[160]).

Now that we have a Docker container running (one for each of the distributions listed in the `env` configuration), we can start running some tests on our Ansible role or playbook.

## Testing the role's syntax

This is the easiest test; `ansible-playbook` has a built in command to check a playbook's syntax (including all the included files and roles), and return `0` if there are no problems, or an error code and some output if there were any syntax issues.

```
ansible-playbook /etc/ansible/roles/role_under_test/test.yml --syntax-c\
heck
```

Add this as a command in the `script` section of `.travis.yml`:

[159]https://github.com/docker/docker/issues/9299
[160]https://github.com/ansible/ansible-modules-core/issues/2459#issuecomment-246880847

```
script:
  # Check the role/playbook's syntax.
  - >
    docker exec --tty "$(cat ${container_id})" env TERM=xterm
    ansible-playbook /etc/ansible/roles/role_under_test/tests/test.yml
    --syntax-check
```

If there are any syntax errors, Travis will fail the build and output the errors in the log.

## Role success - first run

The next aspect to check is whether the role runs correctly or fails on its first run. Add this after the `--syntax-check` test:

```
# Run the role/playbook with ansible-playbook.
- >
  docker exec --tty "$(cat ${container_id})" env TERM=xterm
  ansible-playbook /etc/ansible/roles/role_under_test/tests/test.yml
```

Ansible returns a non-zero exit code if the playbook run fails, so Travis will know whether the command succeeded or failed.

## Role idempotence

Another important test is the idempotence test—does the role change anything if it runs a second time? It should not, since all tasks you perform via Ansible should be idempotent (ensuring a static/unchanging configuration on subsequent runs with the same settings).

```
# Run the role/playbook again, checking to make sure it's idempotent.
- idempotence=$(mktemp)
- >
  docker exec "$(cat ${container_id})"
  ansible-playbook /etc/ansible/roles/role_under_test/tests/test.yml
  | tee -a ${idempotence}
- >
  tail ${idempotence}
  | grep -q 'changed=0.*failed=0'
  && (echo 'Idempotence test: pass' && exit 0)
  || (echo 'Idempotence test: fail' && exit 1)
```

This command runs the exact same command as before, but pipes the results into another temporary file (using `tee`, which pipes the output to the console and the file), and then the next command reads the output and checks to make sure 'changed' and 'failed' both report `0`. If there were no changes or failures, the idempotence test passes (and Travis sees the `0` exit and is happy). If there were any changes or failures, the test fails (and Travis sees the `1` exit and reports a build failure).

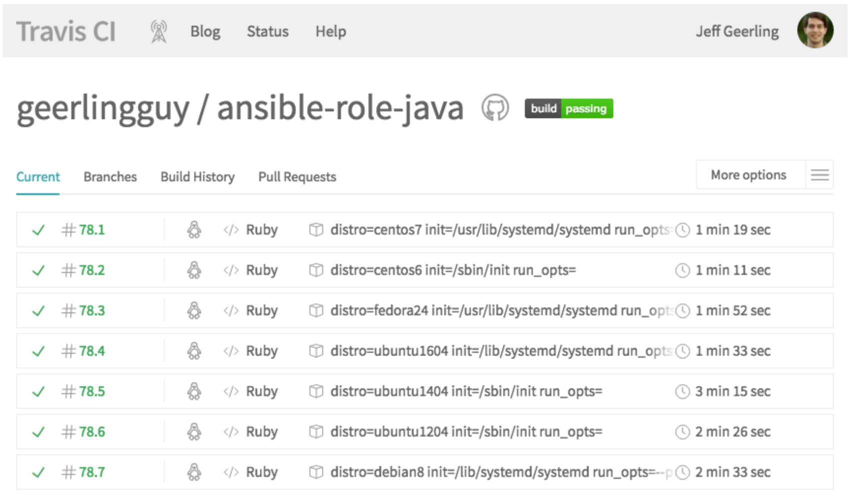
## Role success - final result

The last thing I check is whether the role actually did what it was supposed to do. If it configured a web server, is the server responding on port 80 or 443 without any errors? If it configured a command line application, does the application work when invoked, and do the things it's supposed to do?

```
# Ensure Java is installed.
- 'docker exec --tty "$(cat ${container_id})" env TERM=xterm which java'
```

In this example, a simple test of whether or not `java` is installed is used as a functional test of the role. In other cases, I might run the command `curl http://localhost:3000/` (to check if an app is responding on a particular port), or some other command that verifies an application is installed and running correctly.

Here's what the final test result looks like in Travis CI:



geerlingguy / ansible-role-java

| | | | | |
|---|---|---|---|---|
| ✓ | # 78.1 | Ruby | distro=centos7 init=/usr/lib/systemd/systemd run_opts | 1 min 19 sec |
| ✓ | # 78.2 | Ruby | distro=centos6 init=/sbin/init run_opts= | 1 min 11 sec |
| ✓ | # 78.3 | Ruby | distro=fedora24 init=/usr/lib/systemd/systemd run_opt | 1 min 52 sec |
| ✓ | # 78.4 | Ruby | distro=ubuntu1604 init=/lib/systemd/systemd run_opts | 1 min 33 sec |
| ✓ | # 78.5 | Ruby | distro=ubuntu1404 init=/sbin/init run_opts= | 3 min 15 sec |
| ✓ | # 78.6 | Ruby | distro=ubuntu1204 init=/sbin/init run_opts= | 2 min 26 sec |
| ✓ | # 78.7 | Ruby | distro=debian8 init=/lib/systemd/systemd run_opts=--p | 2 min 33 sec |

**Travis CI test result for the geerlingguy.java role**

Taking this a step further, you could even run a deployed application or service's own automated tests after Ansible is finished with the deployment, thus testing your infrastructure and application in one go—but we're getting ahead of ourselves here… that's a topic for later!

## Some notes about Travis CI

There are a few things you need to know about Travis CI, especially if you're testing Ansible, which will rely heavily on the VM environment inside which it is running:

- **Docker Environment**: The default Docker installation runs on a particular Docker engine version, which may or may not be the latest stable release. Read through Travis' documentation for more: Using Docker in Builds[161].
- **Networking/Disk/Memory**: Travis CI continuously shifts the VM specs you're using, so don't assume you'll have X amount of RAM, disk space, or network capacity. Add commands like `cat /proc/cpuinfo`, `cat /proc/meminfo`, `free -m`, etc. in the `.travis.yml` `before_install` section if you need to figure out the resources available in your VM.

[161]https://docs.travis-ci.com/user/docker/

See much more information about the VM environment on the Travis CI Build Environment page[162].

## Real-world examples

This style of testing is integrated into many of the `geerlingguy.*` roles on Ansible Galaxy; here are a few example roles using Travis CI integration in the way outlined above:

- https://github.com/geerlingguy/ansible-role-java
- https://github.com/geerlingguy/ansible-role-apache
- https://github.com/geerlingguy/ansible-role-mysql

I'd like to give special thanks to Bert Van Vreckem, who helped me to get the initial versions of this Docker-based test workflow working on GitHub; he wrote a bit about the process on his blog, too: Testing Ansible roles with Travis-CI: Multi-platform tests[163].

## Functional testing using serverspec

Serverspec[164] is a tool to help automate server tests using RSpec tests, which use a Ruby-like DSL to ensure your server configuration matches your expectations. In a sense, it's another way of building well-tested infrastructure.

Serverspec tests can be run locally, via SSH, through Docker's APIs, or through other means, without the need for an agent installed on your servers, so it's a lightweight tool for testing your infrastructure (just like Ansible is a lightweight tool for *managing* your infrastructure).

There's a lot of debate over whether well-written Ansible playbooks themselves (especially along with the dry-run `--check` mode) are adequate for well-tested infrastructure, but many teams are more comfortable maintaining infrastructure

[162]http://docs.travis-ci.com/user/ci-environment/
[163]http://bertvv.github.io/notes-to-self/2015/12/13/testing-ansible-roles-with-travis-ci-part-2-multi-platform-tests/
[164]http://serverspec.org/

tests in Serverspec instead (especially if the team is already familiar with how Serverspec and Rspec works!).

Consider this: a truly idempotent Ansible playbook is already a great testing tool if it uses Ansible's robust core modules and `fail`, `assert`, `wait_for` and other tests to ensure a specific state for your server. If you use Ansible's `user` module to ensure a given user exists and is in a given group, and run the same playbook with `--check` and get `ok` for the same task, isn't that a good enough test your server is configured correctly?

This book will not provide a detailed guide for using Serverspec with your Ansible-managed servers, but here are a few resources in case you'd like to use it:

- A brief introduction to server testing with Serverspec[165]
- Testing Ansible Roles with Test Kitchen, Serverspec and RSpec[166]
- Testing infrastructure with serverspec[167]

## Other server and role testing tools

There are also a number of other projects which abstract the testing process a little further than the above approach; some allowing more control and easier use outside of the Travis CI environment, others focused more on Ansible roles in particular:

- molecule[168] - A generalized solution for testing Ansible roles in any environment.
- goss[169] - A generalized server validation tool.
- rolespec[170] - A library for testing Ansible roles on Travis or locally.

Each of the options has some benefits and drawbacks; you should check them all out and find out which one works best in your workflow and skill-set.

[165]https://www.debian-administration.org/article/703/A_brief_introduction_to_server-testing_with_serverspec
[166]http://www.slideshare.net/MartinEtmajer/testing-ansible-roles-with-test-kitchen-serverspec-and-rspec-48185017
[167]http://vincent.bernat.im/en/blog/2014-serverspec-test-infrastructure.html
[168]https://github.com/ansible-community/molecule
[169]https://github.com/aelsabbahy/goss
[170]https://github.com/nickjj/rolespec

## Summary

Tools to help manage, test, and run playbooks regularly and easily, such as Travis CI, Jenkins, and Ansible Tower, also help deliver certainty when applying changes to your infrastructure using Ansible. In addition the information contained in this chapter, read through the Testing Strategies[171] documentation in Ansible's documentation for a comprehensive overview of infrastructure testing and Ansible.

```
 ________________________________________
/ The first rule of any technology used  \
| in a business is that automation       |
| applied to an efficient operation will |
| magnify the efficiency. The second is  |
| that automation applied to an          |
| inefficient operation will magnify the |
\ inefficiency. (Bill Gates)             /
 ----------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

[171]http://docs.ansible.com/test_strategies.html