Hand on Ansible

1. How to Install and Configure Ansible on Ubuntu 16.04

Introduction

Configuration management systems are designed to make controlling large numbers of servers easy for administrators and operations teams. They allow you to control many different systems in an automated way from one central location.While there are many popular configuration management systems available for Linux systems, such as Chef and Puppet, these are often more complex than many people want or need. Ansible is a great alternative to these options because it has a much smaller overhead to get started.In this guide, we will discuss how to install Ansible on a Ubuntu 16.04 server and go over some basics of how to use the software.

How Does Ansible Work?

Ansible works by configuring client machines from an computer with Ansible components installed and configured.

It communicates over normal SSH channels in order to retrieve information from remote machines, issue commands, and copy files. Because of this, an Ansible system does not require any additional software to be installed on the client computers.This is one way that Ansible simplifies the administration of servers. Any server that has an SSH port exposed can be brought under Ansible's configuration umbrella, regardless of what stage it is at in its life cycle. Any computer that you can administer through SSH, you can also administer through Ansible.

Ansible takes on a modular approach, making it easy to extend to use the functionalities of the main system to deal with specific scenarios. Modules can be written in any language and communicate in standard JSON. Configuration files are mainly written in the YAML data serialization format due to its expressive nature and its similarity to popular markup languages. Ansible can interact with clients through either command line tools or through its configuration scripts called Playbooks.

Prerequisites

To follow this tutorial, you will need:

One Ubuntu 16.04 server with a sudo non-root user and SSH keys, which you can set up by following this initial server setup tutorial, including step 4

Step 1 — Installing Ansible

To begin exploring Ansible as a means of managing our various servers, we need to install the Ansible software on at least one machine. We will be using an Ubuntu 16.04 server for this section.The best way to get Ansible for Ubuntu is to add the project's PPA (personal package archive) to your system. We can add the Ansible PPA by typing the following command:

```
$ sudo apt-add-repository ppa:ansible/ansible
```

Press ENTER to accept the PPA addition.

Next, we need to refresh our system's package index so that it is aware of the packages available in the PPA. Afterwards, we can install the software:

```
$ sudo apt-get update
$ sudo apt-get install ansible
```

As we mentioned above, Ansible primarily communicates with client computers through SSH. While it certainly has the ability to handle password-based SSH authentication, SSH keys help keep things simple. You can follow the tutorial linked in the prerequisites to set up SSH keys if you haven't already.

We now have all of the software required to administer our servers through Ansible.

Step 2 — Configuring Ansible Hosts

Ansible keeps track of all of the servers that it knows about through a "hosts" file. We need to set up this file first before we can begin to communicate with our other computers.

Open the file with root privileges like this:

```
$ sudo vi /etc/ansible/hosts
```

You will see a file that has a lot of example configurations, none of which will actually work for us since these hosts are made up. So to start, let's comment out all of the lines in this file by adding a "#" before each line.

We will keep these examples in the file to help us with configuration if we want to implement more complex scenarios in the future.

Once all of the lines are commented out, we can begin adding our actual hosts.

The hosts file is fairly flexible and can be configured in a few different ways. The syntax we are going to use though looks something like this:

Example hosts file

```
[group_name]
alias ansible_ssh_host=your_server_ip
```

The group_name is an organizational tag that lets you refer to any servers listed under it with one word. The alias is just a name to refer to that server.

So in our scenario, we are imagining that we have three servers we are going to control with Ansible. These servers are accessible from the Ansible server by typing:

```
$ ssh root@your_server_ip
```

You should not be prompted for a password if you have set this up correctly. We will assume that our servers' IP addresses are 192.0.2.1, 192.0.2.2, and 192.0.2.3. We will set this up so that we can refer to these individually as host1, host2, and host3, or as a group as servers.

This is the block that we should add to our hosts file to accomplish this:

```
[servers]
host1 ansible_ssh_host=192.0.2.1
host2 ansible_ssh_host=192.0.2.2
host3 ansible_ssh_host=192.0.2.3
```

Hosts can be in multiple groups and groups can configure parameters for all of their members. Let's try this out now.

With our current settings, if we tried to connect to any of these hosts with Ansible, the command would fail (assuming you are not operating as the root user). This is because your SSH key is embedded for the root user on the remote systems and Ansible will by default try to connect as your current user. A connection attempt will get this error:

Ansible connection error

```
host1 | UNREACHABLE! => {
    "changed": false,
    "msg": "Failed to connect to the host via ssh.",
    "unreachable": true
}
```

On the Ansible server, we're using a user called demo. Ansible will try to connect to each host with ssh demo@server. This will not work if the demo user is not on the remote system.

We can create a file that tells all of the servers in the "servers" group to connect using the root user.

To do this, we will create a directory in the Ansible configuration structure called group_vars. Within this folder, we can create YAML-formatted files for each group we want to configure:

```
$ sudo mkdir /etc/ansible/group_vars
$ sudo vi /etc/ansible/group_vars/servers
```

We can put our configuration in here. YAML files start with "---", so make sure you don't forget that part.

```
/etc/ansible/group_vars/servers

    ---
    ansible_ssh_user: root
```

Save and close this file when you are finished.

If you want to specify configuration details for every server, regardless of group association, you can put those details in a file at /etc/ansible/group_vars/all. Individual hosts can be configured by creating files under a directory at /etc/ansible/host_vars.

Step 3 — Using Simple Ansible Commands

Now that we have our hosts set up and enough configuration details to allow us to successfully connect to our hosts, we can try out our very first command.

Ping all of the servers you configured by typing:

```
$ ansible -m ping all
```

Ping output

```
    host1 | SUCCESS => {
        "changed": false,
        "ping": "pong"
    }

    host3 | SUCCESS => {
        "changed": false,
        "ping": "pong"
    }

    host2 | SUCCESS => {
        "changed": false,
        "ping": "pong"
    }
```

This is a basic test to make sure that Ansible has a connection to all of its hosts.

The "all" means all hosts. We could just as easily specify a group:

```
$ ansible -m ping servers
```

We could also specify an individual host:

```
$ ansible -m ping host1
```

We can specify multiple hosts by separating them with colons:

```
$ ansible -m ping host1:host2
```

The -m ping portion of the command is an instruction to Ansible to use the "ping" module. These are basically commands that you can run on your remote hosts. The ping module operates in many ways like the normal ping utility in Linux, but instead it checks for Ansible connectivity.
The ping module doesn't really take any arguments, but we can try another command to see how that works. We pass arguments into a script by typing -a.

The "shell" module lets us send a terminal command to the remote host and retrieve the results. For instance, to find out the memory usage on our host1 machine, we could use:

```
$ ansible -m shell -a 'free -m' host1
```

Shell output

```
    host1 | SUCCESS | rc=0 >>
    total       used       free     shared    buffers     cached
    Mem:       3954        227       3726          0         14         93
    -/+ buffers/cache:        119       3834
    Swap:            0          0          0
```

Conclusion

By now, you should have your Ansible server configured to communicate with the servers that you would like to control. We have verified that Ansible can communicate with each host and we have used theansible command to execute simple tasks remotely.

Although this is useful, we have not covered the most powerful feature of Ansible in this article:

Playbooks. We have set up a great foundation for working with our servers through Ansible, but the

heavy lifting will be done in a future article, when we cover how to use Playbooks to automate configuration of your remote computers.

2. Configuration Management: Writing Ansible Playbooks

Introduction

In a nutshell, server configuration management (also popularly referred to as IT Automation) is a solution for turning your infrastructure administration into a codebase, describing all processes necessary for deploying a server in a set of provisioning scripts that can be versioned and easily reused. It can greatly improve the integrity of any server infrastructure over time.

In a previous guide, we talked about the main benefits of implementing a configuration management strategy for your server infrastructure, how configuration management tools work, and what these tools typically have in common.
This part of the series will walk you through the process of automating server provisioning using Ansible, a configuration management tool that provides a complete automation framework and orchestration capabilities, while maintaining a goal of ultimate simplicity and minimalism. We will focus on the language terminology, syntax, and features necessary for creating a simplified example to fully automate the deployment of an Ubuntu web server using Apache.

This is the list of steps we need to automate in order to reach our goal:

1. Update the apt cache
2. Install Apache
3. Create a custom document root directory
4. Place an index.html file in the custom document root
5. Apply a template to set up our custom virtual host
6. Restart Apache

We will start by having a look at the terminology used by Ansible, followed by an overview of the main language features that can be used to write playbooks. At the end of this guide, we will share the complete example so you can try it by yourself.

Getting Started

Before we can move to a more hands-on view of Ansible, it is important that we get acquainted with important terminology and concepts introduced by this tool.

Ansible Terms

- Controller Machine: the machine where Ansible is installed, responsible for running the provisioning on the servers you are managing
- Inventory: an INI file that contains information about the servers you are managing
- Playbook: the entry point for Ansible provisionings, where the automation is defined through tasks using YAML format
- Task: a block that defines a single procedure to be executed, e.g.: install a package
- Module: a module typically abstracts a system task, like dealing with packages or creating and changing files. Ansible has a multitude of built-in modules, but you can also create custom ones
- Role: a pre-defined way for organizing playbooks and other files in order to facilitate sharing and reusing portions of a provisioning
- Play: a provisioning executed from start to finish is called a play
- Facts: global variables containing information about the system, like network interfaces or operating system
- Handlers: used to trigger service status changes, like restarting or stopping a service

Ansible provisionings are written using YAML, a simple data serialization language.

Tasks

A task defines a single step that should be executed by the provisioning. It typically involves the usage of a module or the execution of a raw command (which, in reality, is just a module created to handle raw commands). This is how a task looks:

```
- name: This is a task
  apt: pkg=vim state=latest
```

The name part is actually optional, but recommended, as shows up in the output of the provisioning when the task is executed. The apt part is a built-in Ansible module that abstracts the management of packages on Debian-based distributions. This tasks tells Ansible that the package vim should have its state changed to latest, which will cause the package manager to install this package in case it is not installed yet.

Playbook Format

Playbooks are the entry point of an Ansible provisioning. They contain information about the systems where the provisioning should be executed, as well as the directives or steps that should be executed. Below you can find an example of a simple playbook that perform two tasks: updates the apt cache and installs vim afterwards:

```
---
- hosts: all
  become: true
  tasks:
     - name: Update apt-cache
       apt: update_cache=yes

     - name: Install Vim
       apt: name=vim state=latest
```

YAML relies on indentation to serialize data structures. For that reason, when writing playbooks and especially when copying examples, you need to be extra careful to maintain the correct indentation.Before the end of this guide we will see a more real-life example of a playbook, explained in detail. The next section will give you an overview of the most important elements and features that can be used to write Ansible playbooks.

Writing Playbooks

Working with Variables

There are different ways in which you can define variables in Ansible. The simplest way is by using thevars section of a playbook. The example below defines a variable package that later is used inside a task:

```
---
- hosts: all
  become: true
  vars:
     package: vim
  tasks:
     - name: Install Package
       apt: name={{ package }} state=latest
```

The variable package can be accessed from any point of the provisioning, even included files and templates.

Using Loops

Loops are typically used to repeat a task using different input values. For instance, instead of creating 10 tasks for installing 10 different packages, you can create a single task and use a loop to repeat the task with all the different packages you want to install.

To create a loop within a task, include the option with_items with an array of values. The content can be accessed through the loop variable item, as shown in the example below:

```
- name: Install Packages
  apt: name={{ item }} state=latest
  with_items:
     - vim
     - git
     - curl
```

You can also use an array variable to define your items:

```
---
- hosts: all
  sudo: true
  vars:
     packages: [ 'vim', 'git', 'curl' ]
  tasks:
     - name: Install Package
       apt: name={{ item }} state=latest
       with_items: packages
```

Using Conditionals

Conditionals can be used to dynamically decide whether or not a task should be executed, based on a variable or an output from a command, for instance.

Below is an example of a task that will only be executed on Debian based systems:

```
- name: Shutdown Debian Based Systems
  command: /sbin/shutdown -t now
  when: ansible_os_family == "Debian"
```

The conditional when receives as argument an expression to be evaluated. The task only gets executed in case the expression is true. In this case, we tested a fact to see if the system is from the Debian family.

A common use case for conditionals in IT automation is when the execution of a task depends on the output of a command. With Ansible, the way we implement this is by registering a variable to hold the results a command execution, and then testing this variable in a subsequent task. We can test for the

command's exit status (if failed or successful). We can also check for specific contents inside the output, although this might require the usage of regex expressions and string parsing commands.

The example below shows two conditional tasks based on the output from a php -v command. We will test for the exit status of the command, since it will fail to execute in case PHP is not installed on this server. The ignore_errors portion of the task is important to make sure the provisioning continues even when the command fails execution.

```
- name: Check if PHP is installed
  register: php_installed
  command: php -v
  ignore_errors: true

- name: This task is only executed if PHP is installed
  debug: var=php_install
  when: php_installed|success

- name: This task is only executed if PHP is NOT installed
  debug: msg='PHP is NOT installed'
  when: php_installed|failed
```

The debug module used here is a useful module for showing contents of variables or debug messages. It can either print a string (when using the msg argument) or print the contents of a variable (when using thevar argument).

Working with Templates

Templates are typically used to set up configuration files, allowing for the use of variables and other features intended to make these files more versatile and reusable. Ansible uses the Jinja2 template engine.

Below is an example of a template for setting up an Apache virtual host, using a variable for setting up the document root for this host:

```
<VirtualHost *:80>
    ServerAdmin webmaster@localhost
    DocumentRoot {{ doc_root }}

    <Directory {{ doc_root }}>
        AllowOverride All
        Require all granted
    </Directory>
</VirtualHost>
```

The built-in module template is used to apply the template from a task. If you named the template file above vhost.tpl, and you placed it in the same directory as your playbook, this is how you would apply the template to replace the default Apache virtual host:

```
- name: Change default Apache virtual host
  template: src=vhost.tpl dest=/etc/apache2/sites-available/000-default.conf
```

Defining and Triggering Handlers

Handlers are used to trigger a state change in a service, such as a restart or a stop. Handlers have a behavior very similar to tasks, being executed once in the same order they are defined, however they are only executed if previously triggered from a notify directive in a task. Handlers are typically defined as an array in a handlers section of the playbook.

Let's take into consideration our previous template usage example, where we set up an Apache virtual host. If you want to make sure Apache is restarted after a virtual host change, you first need to create a handler for the Apache service. This is how handlers are defined inside a playbook:

```
handlers:
    - name: restart apache
```

```
    service: name=apache2 state=restarted

  - name: other handler
    service: name=other state=restarted
```

The name directive here is important because it will be the unique identifier of this handler. To trigger this handler from a task, you should use the notify option:

```
- name: Change default Apache virtual host
  template: src=vhost.tpl dest=/etc/apache2/sites-available/000-default.conf
  notify: restart apache
```

Example Playbook

Now let's have a look at a playbook that will automate the installation of an Apache web server within an Ubuntu system, as discussed in this guide's introduction.

The complete example, including the template file for setting up Apache and an HTML file to be served by the web server, can be found on Github. The folder also contains a Vagrantfile that lets you test the playbook in a simplified setup, using a virtual machine managed by Vagrant.

Below you can find the complete playbook:

playbook.yml

```
---
- hosts: all
  become: true
  vars:
    doc_root: /var/www/example
  tasks:
    - name: Update apt
      apt: update_cache=yes

    - name: Install Apache
      apt: name=apache2 state=latest

    - name: Create custom document root
      file: path={{ doc_root }} state=directory owner=www-data group=www-data

    - name: Set up HTML file
      copy: src=index.html dest={{ doc_root }}/index.html owner=www-data group=www-data
mode=0644

    - name: Set up Apache virtual host file
      template: src=vhost.tpl dest=/etc/apache2/sites-available/000-default.conf
      notify: restart apache
  handlers:
    - name: restart apache
      service: name=apache2 state=restarted
```

Playbook Explained

hosts: all
The playbook starts by stating that it should be applied to all hosts in your inventory (hosts: all). It is possible to restrict the playbook’s execution to a specific host, or a group of hosts.

become: true
The become: true portion tells Ansible to use privilege escalation (sudo) for executing all the tasks in this playbook. This option can be overwritten on a task-by-task basis.

vars
Defines a variable, doc_root, that is later used in a task. This section could contain multiple variables.

tasks
The section where the actual tasks are defined. The first updates the apt cache, and the second task installs the package apache2.

The third task uses the built-in module file to create a directory to serve as our document root. This module can be used to manage files and directories.

The fourth task uses the module copy to copy a local file to the remote server. We're copying a simple HTML file to be served as our website hosted by Apache.

handlers
Finally, we have the handlers section, where the services are declared. We define the restart apache handler that is notified from the fourth task, where the Apache template is applied.

Conclusion

Ansible is a minimalist IT automation tool that has a low learning curve, using YAML for its provisioning scripts. It has a great number of built-in modules that can be used to abstract tasks such as installing packages and working with templates. Its simplified infrastructure requirements and simple language can be a good fit for those who are getting started with configuration management. It might, however, lack some advanced features that you can find with more complex tools like Puppet and Chef.

In the next part of this series, we will see a practical overview of Puppet, a popular and well established configuration management tool that uses an expressive and powerful custom DSL based on Ruby to write provisioning scripts.

3. How To Configure Apache Using Ansible on Ubuntu

Introduction

Apache is one of the most popular web servers currently used on the Internet. It is easy to set up and configure on Linux distributions like Ubuntu and Debian, as it comes in the package repositories and includes a default configuration that works out of the box.

Ansible is an automation tool that allows you to remotely configure systems, install software, and perform complex tasks across a large fleet of servers without needing to manually log into each. Unlike other alternatives, Ansible is installed on a single host, which can even be your local machine, and uses SSH to communicate with each remote host. This allows it to be incredibly fast at configuring new servers, as there are no prerequisite packages to be installed on each new server. It is incredibly easy to use and understand, since it uses playbooks in yaml format using a simple module based syntax.

Prerequisites

For this tutorial, we will install Ansible on a new Ubuntu master server and use it to configure Apache on a second server. That said, keep in mind that one of the benefits of Ansible is that you can have it installed on your local machine and manage other hosts without needing to manually ssh into them.

For this tutorial, you will need:

- Two Ubuntu servers: one master server with Ansible and one secondary which will run Apache configured through Ansible
- Sudo non-root users for both servers.
- Ansible installed on the master server. Follow this tutorial (up to the Set Up SSH Keys section). Although that tutorial was written for Ubuntu 12.04, it is still relevant for Ubuntu .
- SSH keys for the master server to authorize login on the secondary server, which you can do following this tutorial on the master server.
- Active DNS records, or manually set up a local hosts file on your local machine (using your secondary server's IP address), in order to set up and use the Virtual Hosts that will be configured.

Note: This tutorial follows the concepts explained in the existing tutorial:
How To Configure the Apache Web Server on an Ubuntu or Debian VPS. Please review that tutorial if you would like more information, or would like to review the manual process alongside the Ansible process.

Step 1 — Configuring Ansible

In this section we will configure Ansible to be able to manage your server.

The first step, once Ansible is installed, is to tell Ansible which hosts to talk to. To do this, we need to create an Ansible hosts file. The Ansible hosts file contains groups of hosts, which we refer to when running Ansible commands. By default this is located in /etc/ansible/hosts. However, that is applied globally across your system and often requires admin permissions. Instead, to make things simpler, we need to tell Ansible to use a local hosts file.

Ansible always looks for an ansible.cfg file in the local directory that it is being run from, and if found will override the global configuration with the local values. With this in mind, all we need to do is tell Ansible that we want to use a hosts file in the local directory, rather than the global one.

Create a new directory (which we will use for the rest of this tutorial).

```
$ mkdir ansible-apache
```

Move into the new directory.

```
$ cd ~/ansible-apache/
```

Create a new file called ansible.cfg and open it for editing.

```
$ vi ansible.cfg
```

Within that file, we want to add in the hostfile configuration option with the value of hosts, within the[defaults] group. Copy the following into the ansible.cfg file, then save and close it.

```
[defaults]
hostfile = hosts
```

Next, the hosts file needs to be written. There are a lot of options available for the hosts file. However, we can start with something very simple.

Create a hosts file and open it for editing.

```
$ vi hosts
```

Copy the following into the hosts file.

```
[apache]
secondary_server_ip ansible_ssh_user=username
```

This specifies a host group called apache which contains one host. Replace secondary_server_ipwith the secondary server's hostname or IP address, and username with your SSH username. Now Ansible should be able to connect to your server.

Note: The ansible_ssh_user=username component is optional if you are running Ansible as the same user as the target host.

To test that Ansible is working and can talk to your host, you can run a basic ansible command. Ansible comes with a lot of default modules, but a good place to start is the ping module. It checks it can connect to each host, which makes checking the hosts file for correctness easy.

Basic usage of the ansible command accepts the host group, and the module name: ansible <group> -m <module>. To run the ping command, enter the following command.

```
$ ansible apache -m ping
```

The output should look like this:

```
111.111.111.111 | success >> {
    "changed": false,
    "ping": "pong"
}
```

Another Ansible module that is useful for testing is the command module. It runs custom commands on the host and returns the results. To run the command command using echo, a Unix command that echoes a string to the terminal, enter the following command.

```
$ ansible apache -m command -a "/bin/echo hello sammy"
```

The output should look like this:

```
111.111.111.111 | success | rc=0 >>

hello Sammy
```

This is basic usage of Ansible. The real power comes from creating playbooks containing multiple Ansible tasks. We will cover those next.

Step 2 — Creating a Playbook

In this section we will create a basic Ansible playbook to allow you to run more complicated modules easily.

A very basic Ansible playbook is a single yaml file which specifies the host group and one or more tasks to be run on the hosts within the specified group. They are quite simple and easy to read, which is one of the reasons why Ansible is so powerful.

Let's create a basic playbook version of the hello sammy command above.

Create a file called apache.yml and open it for editing.

```
$ vi apache.yml
```

Copy the following text into the file, then save and close it.

```
---
```

```
- hosts: apache
  tasks:
    - name: run echo command
      command: /bin/echo hello Sammy
```

The hosts: apache declaration is at the top, which tells Ansible that we are using the apache hosts group. This is the equivalent of passing it via the ansible command. Next there is a list of tasks. In this example, we have one task with the name run echo command. This is simply a description intended for the user to understand what the task is doing. Finally, the command: /bin/echo hello sammy line runs the command module with the arguments /bin/echo hello sammy.

The $ ansible-playbook command is used to run playbooks, and the simplest usage is: $ ansible-playbook your-playbook.yml. We can run the playbook we just created with the following command.

```
$ ansible-playbook apache.yml
```

The output should look like this.

```
PLAY [apache] ******************************************************************

GATHERING FACTS ****************************************************************
ok: [111.111.111.111]

TASK: [run echo command] *******************************************************
changed: [111.111.111.111]

PLAY RECAP *********************************************************************
111.111.111.111            : ok=2    changed=1    unreachable=0    failed=0
```

The most important thing to notice here is that playbooks do not return the output of the module, so unlike the direct command we used in Step 1, we cannot see if hello sammy was actually printed. This means that playbooks are better suited for tasks where you don't need to see the output. Ansible will tell you if there was an error during the execution of a module, so you generally only need to rely on that to know if anything goes wrong.

Step 3 — Installing Apache

Now that we have introduced playbooks, we will write the task to install the Apache web server.

Normally on Ubuntu, installing Apache is a simple case of installing the apache2 package via apt-get. To do this via Ansible, we use Ansible's apt module. The apt module contains a number of options for specialised apt-get functionality. The options we are interested in are:

- name: The name of the package to be installed, either a single package name or a list of packages.
- state: Accepts either latest, absent, or present. Latest ensures the latest version is installed, present simply checks it is installed, and absent removes it if it is installed.
- update_cache: Updates the cache (via apt-get update) if enabled, to ensure it is up to date.

Note: Package managers other than apt have modules too. Each module page has examples that usually cover all of the main use cases, making it very easy to get a feel for how to use each module. It is rare to have to look elsewhere for usage instructions.

Now let's update our apache.yml playbook with the apt module instead of the command module. Open up the apache.yml file for editing again.

```
$ vi apache.yml
```

Delete the text currently there and copy the following text into it.

```
---
- hosts: apache
  sudo: yes
  tasks:
    - name: install apache2
      apt: name=apache2 update_cache=yes state=latest
```

The apt line installs the apache2 package (name=apache2) and ensures we have updated the cache (update_cache=yes). Although it is optional, including state=latest to be explicit that it should be installed is a good idea.

Unless your Playbook is running as root on each host, sudo will be required to ensure the right privileges. Ansible supports sudo as part of a simple option within the Playbook. It can also be applied via the $ ansible-playbook command and on a per-task level.

Now run the playbook.

```
$ ansible-playbook apache.yml --ask-sudo-pass
```

The --ask-sudo-pass flag will prompt you for the sudo password on the secondary server. This is necessary because the installation requires root privileges; the other commands we've run so far did not.

The output should look like this.

```
PLAY [apache] *****************************************************************

GATHERING FACTS ***************************************************************
ok: [111.111.111.111]

TASK: [install apache2] *******************************************************
changed: [111.111.111.111]

PLAY RECAP ********************************************************************
111.111.111.111            : ok=2    changed=1    unreachable=0    failed=0
```

If you visit your secondary server's hostname or IP address in your browser, you should now get aApache2 Ubuntu Default Page to greet you. This means you have a working Apache installation on your server, and you haven't manually connected to it to run a command yet.

An important concept to note at this point is idempotence, which underlies how Ansible modules are supposed to behave. The idea is that you can run the same command repeatedly, but if everything was configured on the first run, then all subsequent runs make no changes. Almost all Ansible modules support it, including the apt module.

For example, run the same playbook command again.

```
$ ansible-playbook apache.yml --ask-sudo-pass
```

The output should look like this. Note the changed=0 section.

```
PLAY [apache] *****************************************************************
```

```
GATHERING FACTS ***************************************************************
ok: [111.111.111.111]

TASK: [install apache2] *******************************************************
ok: [111.111.111.111]

PLAY RECAP ********************************************************************
111.111.111.111            : ok=2    changed=0    unreachable=0    failed=0
```

This tells you that the apache2 package was already installed, so nothing was changed. When dealing with complicated playbooks across many hosts, being able to identify the hosts that were different becomes very useful. For example, if you notice a host always needs a specific config updated, then there is likely a user or process on that host which is changing it. Without idempotence, this may never be noticed.

Step 4 — Configuring Apache Modules

Now that Apache is installed, we need to enable a module to be used by Apache.

Let us make sure that the mod_rewrite module is enabled for Apache. Via SSH, this can be done easily by using a2enmod and restarting Apache. However, we can also do it very easily with Ansible using theapache2_module module and a task handler to restart apache2.

The apache2_module module takes two options:

- name -- The name of the module to enable, such as rewrite.
- state -- Either present or absent, depending on if the module needs to be enabled or disabled.

Open apache.yml for editing.

```
$ vi apache.yml
```

Update the file to include this task. The file should now look like this:

```
---
- hosts: apache
  sudo: yes
  tasks:
    - name: install apache2
      apt: name=apache2 update_cache=yes state=latest

    - name: enabled mod_rewrite
        apache2_module: name=rewrite state=present
```

However, we need to restart apache2 after the module is enabled. One option is to add in a task to restart apache2, but we don't want that to run every time we apply our playbook. To get around this, we need to use a task handler. The way handlers work is that a task can be told to notify a handler when it has changed, and the handler only runs when the task has been changed.

To do this we need to add the notify option into the apache2_module task, and then we can use theservice module to restart apache2 in a handler.

That results in a playbook that looks like this:

```
---
- hosts: apache
  sudo: yes
  tasks:
    - name: install apache2
      apt: name=apache2 update_cache=yes state=latest
```

```
    - name: enabled mod_rewrite
      apache2_module: name=rewrite state=present
      notify:
        - restart apache2

  handlers:
    - name: restart apache2
      service: name=apache2 state=restarted
```

Now, rerun the playbook.

```
$ ansible-playbook apache.yml --ask-sudo-pass
```

The output should look like

```
PLAY [apache] *****************************************************************


GATHERING FACTS ***************************************************************
ok: [111.111.111.111]

TASK: [install apache2] *******************************************************
ok: [111.111.111.111]

TASK: [enabled mod_rewrite] ***************************************************
changed: [111.111.111.111]

NOTIFIED: [restart apache2] ***************************************************
changed: [111.111.111.111]

PLAY RECAP ********************************************************************
111.111.111.111            : ok=4    changed=2    unreachable=0    failed=0
```

It looks good so far. Now, run the command again and there should be no changes, and the restart task won't be listed.

Step 5 — Configuring Apache Options

Now that we have a working Apache installation, with our required modules turned on, we need to configure Apache.

By default Apache listens on port 80 for all HTTP traffic. For the sake of the tutorial, let us assume that we want Apache to listen on port 8081 instead. With the default Apache configuration on Ubuntu x64, there are two files that need to be updated:

```
/etc/apache2/ports.conf
    Listen 80

/etc/apache2/sites-available/000-default.conf
    <VirtualHost *:80>
```

To do this, we can use the lineinfile module. This module is incredibly powerful and through the use of it's many different configuration options, it allows you to perform all sorts of changes to an existing file on the host. For this example, we will use the following options:

- dest -- The file to be updated as part of the command.
- regexp -- Regular Expression to be used to match an existing line to be replaced.
- line -- The line to be inserted into the file, either replacing the regexp line or as a new line on the end.
- state -- Either present or absent.

Note: The lineinfile module will append the line on the end of the file if it doesn't match an existing line with the regexp. The options insertbefore and insertafter can specify lines to add it before or after instead of at the end, if needed.
What we need to do to update the port from 80 to 8081 is look for the existing lines which define port80, and change them to define port 8081.

Open the apache.yml file for editing.

```
$ vi apache.yml
```

Amend the additional lines so that the file looks like this:

```
---
- hosts: apache
  sudo: yes
  tasks:
    - name: install apache2
      apt: name=apache2 update_cache=yes state=latest

    - name: enabled mod_rewrite
      apache2_module: name=rewrite state=present
      notify:
        - restart apache2

    - name: apache2 listen on port 8081
      lineinfile: dest=/etc/apache2/ports.conf regexp="^Listen 80" line="Listen 8081"
state=present
      notify:
        - restart apache2

    - name: apache2 virtualhost on port 8081
      lineinfile: dest=/etc/apache2/sites-available/000-default.conf regexp="^<VirtualHost
\*:80>" line="<VirtualHost *:8081>" state=present
      notify:
        - restart apache2

  handlers:
    - name: restart apache2
      service: name=apache2 state=restarted
```

It is important to notice that we also need to restart apache2 as part of this process, and that we can re-use the same handler but the hanlder will only be triggered once despite multiple changed tasks.

Now run the playbook.

```
$ ansible-playbook apache.yml --ask-sudo-pass
```

Once Ansible has finished, you should be able to visit your host in your browser and it will respond on port8081, rather than port 80. In most web browsers, this can be easily achieved by adding :port onto the end of the URL: http://111.111.111.111:8081/.

The lineinfile module is very powerful, and makes mangling existing configurations really easy. The only catch is that you need to know what to expect in the files you are changing with it, but it supports a wide variety of options that support most simple use cases.

Step 6 — Configuring Virtual Hosts

Ansible features a couple of modules that provide the ability to copy a local (to Ansible) template file onto the hosts. The two most commonly used modules for this purpose are the copy module and the templatemodule. The copy module copies a file as-is and makes no changes to it, whereas the more powerfultemplate module copies across a template and applies variable substitution to areas you specify by using double curly brackets (i.e. {{ variable }}).

In this section we will use the template module to configure a new virtual host on your server. There will be a lot of changes, so we'll explain them piece by piece, and include the entire updated apache.yml file at the end of this step.

Create Virtual Host Configuration

The first step is to create a new virtual host configuration. We'll create the virtual host configuration file on the master server and upload it to the secondary server using Ansible.

Here's an example of a basic virtual host configuration which we can use as a starting point for our own configuration. Notice that both the port number and the domain name, hilighted below, are hardcoded into the configuration.

```
<VirtualHost *:8081>
    ServerAdmin webmaster@example.com
    ServerName example.com
    ServerAlias www.example.com
    DocumentRoot /var/www/example.com
    ErrorLog ${APACHE_LOG_DIR}/error.log
    CustomLog ${APACHE_LOG_DIR}/access.log combined
</VirtualHost>
```

Create a new file called virtualhost.conf.

```
$ vi virtualhost.conf
```

Paste the following into virtualhost.conf. Because we are using templates, it is a good idea to change the hard coded values above to variables, to make them easy to change in the future.

```
<VirtualHost *:{{ http_port }}>
    ServerAdmin webmaster@{{ domain }}
    ServerName {{ domain }}
    ServerAlias www.{{ domain }}
    DocumentRoot /var/www/{{ domain }}
    ErrorLog ${APACHE_LOG_DIR}/error.log
    CustomLog ${APACHE_LOG_DIR}/access.log combined
</VirtualHost>
```

Use Template Variables

Next, we need to update our playbook to push out the template and use the variables.

The first step is to add in a section into the playbook for variables. It is called vars and goes on the same level as hosts, sudo, tasks, and handlers. We need to put in both variables used in the template above, and we will change the port back to 80 in the process.

```
---

- hosts: apache
  sudo: yes
  vars:
    http_port: 80
    domain: example.com
  tasks:
    - name: install apache2
...
```

Variables can be used in tasks and templates, so we can update our existing lineinfile modules to use the specified http_port, rather than the hard coded 8081 we specified before. The variable needs to be added into the line, and the regexp option needs to be updated so it's not looking for a specific port. The changes will look like this:

```
lineinfile: dest=/etc/apache2/ports.conf regexp="^Listen " line="Listen {{ http_port }}"
state=present
lineinfile: dest=/etc/apache2/sites-available/000-default.conf regexp="^<VirtualHost \*:"
line="<VirtualHost *:{{ http_port }}>"
```

Add Template Module

The next step is to add in the template module to push the configuration file onto the host. We will use these options to make it happen:

- dest -- The destination file path to save the updated template on the host(s), i.e./etc/apache2/sites-available/{{ domain }}.conf.
- src -- The source template file, i.e. virtualhost.conf.

Applying these to your playbook will result in a task that looks like this:

```
- name: create virtual host file
  template: src=virtualhost.conf dest=/etc/apache2/sites-available/{{ domain }}.conf
```

Enable the Virtual Host

Almost done! What we need to do now is enable the virtual host within Apache. This can be done in two ways: by running the sudo a2ensite example.com command or manually symlinking the config file into /etc/apache2/sites-enabled/. The former option is safer, as it allows Apache to control the process. For this, the command module comes in use again.

The usage is quite simple, as we discovered above:

```
- name: a2ensite {{ domain }}
  command: a2ensite {{ domain }}
  notify:
  - restart apache2
```

Prevent Extra Work

Finally, the command module needs to know when it should and shouldn't run, so the module is not run unnecessarily if the playbook is run multiple times. In our case, it only needs to be run if the .conf file hasn't been created on the host yet.

This is done using the creates option, which allows you to tell the module what file is being created during the module execution. If the file exists, the module won't run. Because Apache creates a symlink when sites are enabled, checking for that solves the problem.

The changes will look like this:

```
- name: a2ensite {{ domain }}
  command: a2ensite {{ domain }}
  args:
    creates: /etc/apache2/sites-enabled/{{ domain }}.conf
  notify:
  - restart apache2
```

It is important to note the use of the args section in the task. This is an optional way of listing the module options, and in this case removes any confusion between what is a module option and what is the command itself.

Final `apache.yml` Playbook

Now let's apply these changes. Open apache.yml.

```
$ vi apache.yml
```

With all of the changes above, change your apache.yml playbook to look like this.

```
---
- hosts: apache
  sudo: yes
  vars:
    http_port: 80
    domain: example.com
  tasks:
    - name: install apache2
      apt: name=apache2 update_cache=yes state=latest

    - name: enabled mod_rewrite
      apache2_module: name=rewrite state=present
      notify:
        - restart apache2

    - name: apache2 listen on port {{ http_port }}
      lineinfile: dest=/etc/apache2/ports.conf regexp="^Listen " line="Listen {{ http_port }}"
state=present
      notify:
        - restart apache2

    - name: apache2 virtualhost on port {{ http_port }}
      lineinfile: dest=/etc/apache2/sites-available/000-default.conf regexp="^<VirtualHost \*:"
line="<VirtualHost *:{{ http_port }}>"
      notify:
        - restart apache2

    - name: create virtual host file
      template: src=virtualhost.conf dest=/etc/apache2/sites-available/{{ domain }}.conf

    - name: a2ensite {{ domain }}
      command: a2ensite {{ domain }}
      args:
        creates: /etc/apache2/sites-enabled/{{ domain }}.conf
      notify:
        - restart apache2

  handlers:
    - name: restart apache2
      service: name=apache2 state=restarted
```

Save and close the file, then run the playbook.

```
$ ansible-playbook apache.yml --ask-sudo-pass
```

If you now visit the hostname or IP address of your secondary server in your browser, you will see it responds on port 80 again, not port 8081. Next, visit the domain (i.e. example.com) we specified for the new virtual host. Because we haven't added any files in yet, it should show an Apache 404 error page rather than the Apache welcome page. If so, your virtual host is working correctly, and you still haven't SSH'ed into your secondary server to run a single command.

Step 7 — Using a Git Repository For Your Website

In this section we will use Ansible to clone a Git repository in order to set up your website content.

Every website needs content, and although it is normal to SSH in and manually clone a Git repository to set up a new website, Ansible provides us with the tools we need to do it automatically. For this example, the git module will do what is required.

The git module has a lot of options, with the relevant ones for this tutorial being:

- dest -- The path on the host where the repository will be checked out to.
- repo -- The repository url that will be cloned. This must be accessible by the host.
- update -- When set to no, this prevents Ansible from updating the repository when it already exists.
- accept_hostkey -- Tells SSH to accept any unknown host key when connecting via SSH. This is very useful as it saves the need to login via SSH to accept the first login attempt, but it does remove the ability to manually check the host signature. Depending on your repository, you may need this option.

For the purposes of the tutorial, there is a simple Git repository with a single index.html page that can be cloned onto your host. If you already have another public repository that contains similar, feel free to substitute it. With that in mind, the git task will look like this:

```
- name: clone basic html template
  git: repo=https://github.com/do-community/ansible-apache-tutorial.git
dest=/var/www/example.com update=no
```

However, if you ran the Playbook now, you would probably get an error. We first need to install the gitpackage so Ansible can use it to clone the repository. The apt task needs to be updated to install both the apache2 package and the git package. Checking the apt documentation tells us that the nameoption only takes a single package, so that won't help. Instead, we need to use a list of items.

Ansible provides the ability to specify a list of items to loop through and apply the task to each. They are specified using the with_items option as part of the task, and our apt task will be updated to look like this:

```
- name: install packages
  apt: name={{ item }} update_cache=yes state=latest
  with_items:
    - apache2
    - git
```

The list of items uses the item variable and will execute the task for each item in the list.

Open apache.yml again.

```
$ vi apache.yml
```

Update the playbook to match the following:

```
---
- hosts: apache
```

```
  sudo: yes

  vars:
    http_port: 80
    domain: example.com

  tasks:

    - name: install packages
      apt: name={{ item }} update_cache=yes state=latest
      with_items:
        - apache2
        - git

    - name: enabled mod_rewrite
      apache2_module: name=rewrite state=present
      notify:
        - restart apache2

    - name: apache2 listen on port {{ http_port }}
      lineinfile: dest=/etc/apache2/ports.conf regexp="^Listen " line="Listen {{ http_port }}"
state=present
      notify:
        - restart apache2

    - name: apache2 virtualhost on port {{ http_port }}
      lineinfile: dest=/etc/apache2/sites-available/000-default.conf regexp="^<VirtualHost \*:"
line="<VirtualHost *:{{ http_port }}>"
      notify:
        - restart apache2

    - name: create virtual host file
      template: src=virtualhost.conf dest=/etc/apache2/sites-available/{{ domain }}.conf

    - name: a2ensite {{ domain }}
      command: a2ensite {{ domain }}
      args:
        creates: /etc/apache2/sites-enabled/{{ domain }}.conf
      notify:
        - restart apache2

    - name: clone basic html template
      git: repo=https://github.com/do-community/ansible-apache-tutorial.git
dest=/var/www/example.com update=no

  handlers:
    - name: restart apache2
      service: name=apache2 state=restarted
```

Save the file and run the playbook.

```
$ ansible-playbook apache.yml --ask-sudo-pass
```

It should install git and successfully clone the repository. You should now see something other than a 404 error when you visit the virtual host from Step 6. Don't forget to check the non virtual host is still returning the default page.

In summary, you now have Git installed and a basic HTML page has been cloned via Git onto your new virtual host. There are still no manual SSH commands required. If you're only after a basic HTML website, and it's in a public Git repository, then you are done!

Conclusion

We have just created an Ansible Playbook to automate the entire process of configuring your host to run the Apache Web Server, with virtual hosts, and a Git repository. All of that has been achieved without needing to log directly into the server, and the best part is that you can run your new Playbook against most Ubuntu servers to achieve the same result.

Note: if your host already has Apache set up and modified, you will most likely need to handle each of the modifications to bring it back to the required state. On the positive side, Ansible will only fix these modifications if they exist, so it's safe to have them in the main Playbook!

Ansible is incredibly powerful and also has a very easy learning curve. You can start off using the basic concepts covered in this tutorial and either stay at this level or learn a lot more to get to the really complicated parts. Either way, you will be able to configure and manage your server(s) without needing to manually login for most, if not all, tasks.

4. Warm up before creating Ansible Roles: Ansible Playbooks to Automate System Configuration on Ubuntu

Introduction

We are going to write a Ansible playbook to setup nginx server.

In the next section we will see how to convert same playbook into Roles.

Exploring a Basic Playbook

Let's look at a basic playbook:

```
---
- hosts: all
  tasks:
    - name: Installs nginx web server
      apt: pkg=nginx state=installed update_cache=true
      notify:
        - start nginx

  handlers:
    - name: start nginx
      service: name=nginx state=started
```

Let's break this down in sections so we can understand how these files are built and what each piece means.

The file starts with:

This is a requirement for YAML to interpret the file as a proper document. YAML allows multiple "documents" to exist in one file, each separated by ---, but Ansible only wants one per file, so this should only be present at the top of the file.

YAML is very sensitive to white-space, and uses that to group different pieces of information together. You should use only spaces and not tabs and you must use consistent spacing for your file to be read correctly. Items at the same level of indentation are considered sibling elements.

Items that begin with a - are considered list items. Items that have the format of key: value operate as hashes or dictionaries. That's pretty much all there is to basic YAML.

YAML documents basically define a hierarchical tree structure with the containing elements further to the left.

On the second line, we have this:

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

This is a list item in YAML as we learned above, but since it is at the left-most level, it is also an Ansible "play". Plays are basically groups of tasks that are performed on a certain set of hosts to allow them to fulfill the function you want to assign to them. Each play must specify a host or group of hosts, as we do here.

Next, we have a set of tasks:

```
---
- hosts: all
  tasks:
    - name: Installs nginx web server
      apt: pkg=nginx state=installed update_cache=true
      notify:
        - start nginx
```

At the top level, we have "tasks:" at the same level as "hosts:". This contains a list (because it starts with a "-") which contains key-value pairs.

The first one, "name", is more of a description than a name. You can call this whatever you would like.

The next key is "apt". This is a reference to an Ansible module, just like when we use the ansible command and type something like:

```
$ ansible -m apt -a 'whatever' all
```

This module allows us to specify a package and the state that it should be in, which is "installed" in our case. The update-cache=true part tells our remote machine to update its package cache (apt-get update) prior to installing the software.

The "notify" item contains a list with one item, which is called "start nginx". This is not an internal Ansible command, it is a reference to a handler, which can perform certain functions when it is called from within a task. We will define the "start nginx" handler below.

```
---
- hosts: droplets
  tasks:
    - name: Installs nginx web server
      apt: pkg=nginx state=installed update_cache=true
      notify:
        - start nginx

  handlers:
    - name: start nginx
      service: name=nginx state=started
```

The "handlers" section exists at the same level as the "hosts" and "tasks". Handlers are just like tasks, but they only run when they have been told by a task that changes have occurred on the client system.

For instance, we have a handler here that starts the Nginx service after the package is installed. The handler is not called unless the "Installs nginx web server" task results in changes to the system, meaning that the package had to be installed and wasn't already there.

We can save this playbook into a file called something like "nginx.yml".

Just for some context, if you were to write this same file in JSON, it might look something like this:

```
[
    {
        "hosts": "droplets",
        "tasks": [
```

```
            {
                "name": "Installs nginx web server",
                "apt": "pkg=nginx state=installed update_cache=true",
                "notify": [
                    "start nginx"
                ]
            }
        ],
        "handlers": [
            {
                "name": "start nginx",
                "service": "name=nginx state=started"
            }
        ]
    }
]
```

As you can see, YAML is much more compact and most people would say more readable.

Running an Ansible Playbook

Once you have a playbook built, you can call it easily using this format:

```
$ ansible-playbook playbook.yml
```

For instance, if we wanted to install and start up Nginx on all of our droplets, we could issue this command:

```
$ ansible-playbook nginx.yml
```

Since the playbook itself specifies the hosts that it should run against (namely, the "droplets" group we created in the last tutorial), we do not have to specify a host to run against.

However, if we would like to filter the host list to only apply to one of those hosts, we can add a flag to specify a subset of the hosts in the file:

```
$ ansible-playbook -l host_subset playbook.yml
```

So if we only wanted to install and run Nginx on our "host3", we could type this:

```
$ ansible-playbook -l host3 nginx.yml
```

Adding Features to the Playbook

Right now our playbook looks like this:

```
---
- hosts: droplets
  tasks:
    - name: Installs nginx web server
      apt: pkg=nginx state=installed update_cache=true
      notify:
        - start nginx

  handlers:
    - name: start nginx
      service: name=nginx state=started
```

It is simple and it works, but all it is doing is installing a piece of software and starting it. That's not very beneficial by itself.

We can start to expand the functionality by adding tasks to our playbook.

Add a Default Index File

We can tell it to transfer a file from our Ansible server onto the host by adding some lines like this:

```
---
- hosts: droplets
  tasks:
    - name: Installs nginx web server
      apt: pkg=nginx state=installed update_cache=true
      notify:
        - start nginx

    - name: Upload default index.html for host
      copy: src=static_files/index.html dest=/usr/share/nginx/www/ mode=0644

  handlers:
    - name: start nginx
      service: name=nginx state=started
```

We can then make a directory called static_files in our current directory and place an index.html file inside.

```
$ mkdir static_files
$ vi static_files/index.html
```

```
Inside of this file, let's just create a basic html structure:
```

```
<html>
  <head>
    <title>This is a sample page</title>
  </head>
  <body>
    <h1>Here is a heading!</h1>
    <p>Here is a regular paragraph.  Wow!</p>
  </body>
</html>
```

Save and close the file.
Now, when we re-run the playbook, Ansible will check each task. It will see that Nginx is already installed on the host, so it will leave it be. It will see the new task section and replace the default index.html file with the one from our server.

Registering Results

When you are installing and configuring services manually, it is almost always necessary to know whether your actions were successful or not. We can cook this functionality into our playbooks by using "register".

For each task, we can optionally register its result (failure or success) in a variable that we can check later on.

When using this functionality, we also have to tell Ansible to ignore errors for that task, since normally it aborts the playbook execution for that host if any trouble happens.

So, if we want to check whether a task has failed or not to decide on subsequent steps, we can use the register functionality.

For instance, we could tell our playbook to upload an index.php file if it exists. If that task fails, we could instead try to upload an index.html file. We will check for the failure condition in the other task because we only want to upload the HTML file if the PHP file fails:

```
---
- hosts: droplets
```

```
tasks:
  - name: Installs nginx web server
    apt: pkg=nginx state=installed update_cache=true
    notify:
    - start nginx
  - name: Upload default index.php for host
    copy: src=static_files/index.php dest=/usr/share/nginx/www/ mode=0644
    register: php
    ignore_errors: True

  - name: Remove index.html for host
    command: rm /usr/share/nginx/www/index.html
    when: php|success

  - name: Upload default index.html for host
    copy: src=static_files/index.html dest=/usr/share/nginx/www/ mode=0644
    when: php|failed

handlers:
  - name: start nginx
    service: name=nginx state=started
```

Note: We have not configured our host to handle PHP files at this time, so even if you did upload a PHP file, it would not be processed correctly.
This new version tries to upload a PHP index file to the host. It registers the success of the operation into a variable called "php".

If this operation was successful, the task to remove the index.html file is run next.

If the operation failed, the index.html file is uploaded instead.

Conclusion

Now, you should have a good handle on how to automate complex tasks using Ansible. This is a basic example of how you can begin to build your configuration library.

Combining host and group definitions as we learned about in the first tutorial, and using available variables to fill in information, we can begin to put together complex computer systems that interact with each other. In a future article, we will discuss how to implement variables into our playbooks and create roles to help manage complex tasks.

5. How to Use Ansible Roles to Abstract your Infrastructure Environment

Introduction

Ansible is an easy to use configuration management system that can assist you in configuring large numbers of servers from a single machine. You can automate complex tasks and easily add machines to your infrastructure without too much trouble.

In previous articles, we discussed how to install and configure Ansible and how to create playbooks to automate system configuration. In this guide, we will discuss how to use Ansible roles to allow you to break up configuration into more modular steps.

We will assume that you have Ansible installed on one Ubuntu 12.04 VPS as we demonstrated in the previous guides. You will also need one or more other computers configured as hosts within the Ansible configuration.

What is an Ansible Role?

You've been exposed to how Ansible can interact with configured clients from the command line with theansible command, and how you can automate configuration with playbooks run through the $ ansible-playbook command. Where do roles fit into this scheme?

Simply put, roles are a further level of abstraction that can be useful for organizing playbooks. As you add more and more functionality and flexibility to your playbooks, they can become unwieldy and difficult to maintain as a single file. Roles allow you to create very minimal playbooks that then look to a directory structure to determine the actual configuration steps they need to perform.

Organizing things into roles also allows you to reuse common configuration steps between different types of servers. This is already possible by "including" other files within a playbook, but with roles, these types of links between files are automatic based on a specific directory hierarchy.

In general, the idea behind roles is to allow you to define what a server is supposed to do, instead of having to specify the exact steps needed to get a server to act a certain way.

Creating Role Framework

In order for Ansible to correctly handle roles, we need to build a directory structure that it can find and understand. We can do this by creating a "roles" directory in our working directory for Ansible.

We're assuming here that you've been using your user's home directory as the Ansible working directory. You should change to whatever actual directory you are keeping your Ansible configuration in.

We are going to create a directory called "roles" where Ansible will look for our roles.

```
$ cd ~
$ mkdir roles
$ cd roles
```

Within this directory, we will define our roles. We will basically create a directory for each role that we will create. Since we are going to replicate our Nginx playbook, let's create an Nginx role:

```
$ mkdir nginx
$ cd nginx
```

Within this directory, we create another set of directories that will help us separate the different sections of a normal playbook. Create these directories now:

```
$ mkdir files handlers meta templates tasks vars
```

These are the directories that will contain all of the code to implement our configuration. You may not use all of the directories, so in real practice, you may not need to create all of these directories.

This is what they are all for:

- files: This directory contains regular files that need to be transferred to the hosts you are configuring for this role. This may also include script files to run.
- handlers: All handlers that were in your playbook previously can now be added into this directory.
- meta: This directory can contain files that establish role dependencies. You can list roles that must be applied before the current role can work correctly.
- templates: You can place all files that use variables to substitute information during creation in this directory.
- tasks: This directory contains all of the tasks that would normally be in a playbook. These can reference files and templates contained in their respective directories without using a path.
- vars: Variables for the roles can be specified in this directory and used in your configuration files.

Within all of the directories but the "files" and "templates", if a file called main.yml exists, its contents will be automatically added to the playbook that calls the role.

Abstracting a Playbook to a Role

For many playbooks, it would make more sense to implement the functionality as a role. We can turn our Nginx playbook from the last article into a role to organize things better.

We should already have the roles/nginx/{subdirectories} structure set up from the last section. Now, we need to create some main.yml files in our structure.

Creating the Tasks main.yml File

We'll start with the tasks subdirectory. Move to that directory now:

```
$ cd ~/roles/nginx/tasks
```
Now, we need to copy the nginx.yml file into this directory:

```
$ cp ~/nginx.yml main.yml
```
Now, we need to edit the main file and remove everything that is not a tasks:

```
$ vi main.yml
```

The file should look like this when you begin:

```
---
- hosts: droplets
  tasks:
    - name: Installs nginx web server
      apt: pkg=nginx state=installed update_cache=true
      notify:
        - start nginx
    - name: Upload default index.php for host
      copy: src=static_files/index.php dest=/usr/share/nginx/www/ mode=0644
      register: php
      ignore_errors: True
    - name: Remove index.html for host
      command: rm /usr/share/nginx/www/index.html
      when: php|success
    - name: Upload default index.html for host
      copy: src=static_files/index.html dest=/usr/share/nginx/www/ mode=0644
      when: php|failed
  handlers:
    - name: start nginx
      service: name=nginx state=started
```

We only want to keep the lines that are red. Furthermore, we can remove the extraneous spaces to the left of our tasks. After our changes, our new tasks/main.yml file will look like this:

```
---
- name: Installs nginx web server
  apt: pkg=nginx state=installed update_cache=true
  notify:
    - start nginx

- name: Upload default index.php for host
  copy: src=static_files/index.php dest=/usr/share/nginx/www/ mode=0644
  register: php
  ignore_errors: True

- name: Remove index.html for host
  command: rm /usr/share/nginx/www/index.html
  when: php|success

- name: Upload default index.html for host
  copy: src=static_files/index.html dest=/usr/share/nginx/www/ mode=0644
  when: php|failed
```

As you can see, this is a lot easier to read in terms of just recognizing the steps that are to be performed.

One additional change that we should make is how we references external files in our configuration. Oursrc lines reference a "static_files" directory. This is unnecessary if we place all of our static files in the "files" subdirectory. Ansible will find them automatically.

When we change those lines, our finished tasks/main.yml file looks like this:

```
---
- name: Installs nginx web server
  apt: pkg=nginx state=installed update_cache=true
  notify:
    - start nginx

- name: Upload default index.php for host
```

```
  copy: src=index.php dest=/usr/share/nginx/www/ mode=0644
  register: php
  ignore_errors: True

- name: Remove index.html for host
  command: rm /usr/share/nginx/www/index.html
  when: php|success

- name: Upload default index.html for host
  copy: src=index.html dest=/usr/share/nginx/www/ mode=0644
  when: php|failed
```

Save and close the file when you are finished.

Creating the Handlers main.yml File

Now that we have the bulk of the playbook in the tasks/main.yml file, we need to move the handlers section into a file located at handlers/main.yml.

Copy the nginx.yml file again, this time into the handlers directory:

```
$ cd ~/roles/nginx/handlers
$ cp ~/nginx.yml main.yml
```

Again, open the file in your text editor:

```
$ vi main.yml
```

The parts that we need to keep are in red again:

```
---
- hosts: droplets
  tasks:
    - name: Installs nginx web server
      apt: pkg=nginx state=installed update_cache=true
      notify:
        - start nginx

    - name: Upload default index.php for host
      copy: src=static_files/index.php dest=/usr/share/nginx/www/ mode=0644
      register: php
      ignore_errors: True

    - name: Remove index.html for host
      command: rm /usr/share/nginx/www/index.html
      when: php|success

    - name: Upload default index.html for host
      copy: src=static_files/index.html dest=/usr/share/nginx/www/ mode=0644
      when: php|failed

  handlers:
    - name: start nginx
      service: name=nginx state=started
```

Remove the whitespace from before the handlers also. In the end, the file should look like this:

```
---
- name: start nginx
  service: name=nginx state=started
```

Save and close the file when you are finished.

Finishing Up

Since our original playbook was very simple, we're almost done.

First, we need to move the index.html page (and the index.php page if you created one) out of the~/static_files directory and put them into the ~/roles/nginx/files directory:

```
$ cp ~/static_files/* ~/roles/nginx/files
```

If our role depended on another role, we could add a file in the meta directory called main.yml. This file might specify that this role depends on a role called "apt".

If our role depended on a role called "apt", the file at ~/roles/nginx/meta/main.yml might look like this:

```
---
dependencies:
  - { role: apt }
```

This would take the "apt" role and place the information from that role before our Nginx information so that our role has its proper dependencies prior to starting.

We said earlier that there is a "vars" directory that can be used to set variables for our role. While it is possible to configure default parameters for a role through a vars/main.yml file, this is usually not recommended, because it makes the details of your configuration reside within the roles hierarchy.

Usually, you want to specify your details outside of the role so that you can easily share the role structure without worrying about leaking information. Also, variables declared within a role are easily overridden by variables in other locations, so they are not very strong to begin with.

By now, you may be wondering why we have to organize our information into directories, when most of our directories only contain a single main.yml file. Why aren't we creating tasks.yml files instead oftasks/main.yml?

The answer is that we are only using the minimum amount of files. The main.yml files are the ones picked up automatically by Ansible, but we can include additional files easily by using the includefunctionality.

If we had an additional task file used to configure SSL for some of our hosts located at tasks/ssl.yml, we could call it like this:

```
. . .
tasks:
  - include: roles/nginx/tasks/ssl.yml
```

Create a Skeleton Playbook

Now that we have configured our role structure, we can call all of the functionality with a very simple playbook.

This allows us to use playbooks to declare what a server is supposed to do, not what steps must happen to make it behave how we want it to.

Outside of the entire role structure, in our working directory (our home directory in this example), we can create a playbook file.

```
cd ~
nano play.yml
```

Inside of this file, we need very little information. First, we have not defined any hosts, so that goes here. Next, we just declare the role we are using:

```
---
- hosts: droplets
  roles:
    - role: nginx
```

Save and close the file. This is our entire playbook. As you can see, it cleans everything up and allows us to concentrate on core functionality. If we had multiple roles configured, we could simply list what different things we want our server to do.

For instance, if we had roles to set up a WordPress server, we might have a playbook that looks like this:

```
---
- hosts: wordpress_hosts
  roles:
    - nginx
    - php
    - mysql
    - wordpress
```

As you can see, this allows us to be very succinct about what we want from a server. Since in the end, we use a playbook to call a role, the command syntax is exactly the same:

```
$ ansible-playbook play.yml
```

Conclusion

Ansible roles are an optional feature to take advantage of, but if you plan on using Ansible extensively, it is highly recommended that you explore this functionality. Not only will it keep your host-level configuration clean and readable, it will also allow you to easily reuse code and implement your changes in a modular fashion.

6. Real time project | Multi tier web application stack deployment using Ansible.

Prereqs

Vagrant and virtualbox installed

Create directories on you local machines
```
# mkdir ansible_proj
# mkdir ansible_proj/Vms
# mkdir ansible_proj/control_repo
```

Spin vagrant enviornment
```
# cd learn_proj/Vms
```

Change IP address of the vm’s from vagrant file
```
# vi Vagrantfile
# vagrant up
```

Assumptions
ansible doc reference for ansible server(control server).
All the commands will be executing from the ansible control machine.
Control machine ssh access(vagrant ssh control).
Super user access on control machine.

SSH key exchange from control node to lb01, app01/02 & db01 will be taken care by Vagrantfile.
Host to ip mapping in control, lb01, app01/02 & db01 will be taken care by Vagrantfile.
Directory sync established from ansible_proj/control_repo to control machine's /home/vagrant/repo through Vagrantfile.

Ansible installation
Ansible package needs to be installed on control machine
Refer ansible doc
http://docs.ansible.com/ansible/intro_installation.html#latest-releases-via-apt-ubuntu

Verify installation
```
# ansible --help
# ansible-playbook --help
# ansible-galaxy --help
```

Setting up ansible repo
Login to control server
```
# vagrant ssh control
# mkdir -p /home/vagrant/repo/ansible
# cd /home/vagrant/repo/ansible
# vi dev (inventory)
+[loadbalancer]
+lb01

+[webserver]
+app01
+app02

+[database]
+db01

+[control]
+control ansible_connection=local
```

Ansible.cfg
Set inventory path in ansible.cfg

Benefit of tracking it in version control system
```
# cd /home/vagrant/repo/ansible
# vi ansible.cfg

+[defaults]
+inventory = ./dev

# ansible --list-hosts all
```

Validate the inventory and connections
```
# ansible --list-hosts all
```

Can use pattern syntax that allow you to select subset from the inventory.

```
# ansible --list-hosts all
# ansible --list-hosts "*"
# ansible --list-hosts loadbalancer
# ansible --list-hosts webserver
# ansible --list-hosts db01
```

```
# ansible --list-hosts database:control
#ansible --list-hosts webserver[0]
# ansible --list-hosts \!control
```

https://docs.ansible.com/ansible/intro_patterns.html

https://docs.ansible.com/ansible/intro_getting_started.html#your-first-commands
https://docs.ansible.com/ansible/ping_module.html
https://docs.ansible.com/ansible/command_module.html

```
# ansible -m ping all
# ansible -m command -a “hostname” all
```

Default module is command
```
# ansible -a “hostname” all
```

All tasks are gonna have return status

Return exit code non zero
```
# ansible -a “/bin/false” all
```

http://docs.ansible.com/ansible/modules_by_category.html

Plays

```
# cd /home/vagrant/repo/ansible
# vi hostname.yml
+ ---
+  - hosts: all
+    tasks:
+      - command: hostname
```

Playbook Execution

```
# ansible-playbook hostname.yml
# vi hostname.yml
---
- hosts: all
    tasks:
+       - name : get server info
-       - command: hostname
+        command: hostname

# ansible-playbook hostname.yml
```

Four pillar of linux application | Principles to setup app on linux

1. Packages’: From repositories(apt or yum) or any other resources
2. Services: init.d or system.d or your own start script
3. System configurations: Files, directories, users, permission, firewall rules etc
4. Config files for the app itself

Playbook intro

http://docs.ansible.com/ansible/modules_by_category.html

Packages

Creating playbook for installing nginx package on loadbalancer

vi loadbalancer.yml

```
---
- hosts: loadbalancer
become: true
  tasks:
    - name: Install nginx
      apt: name=nginx state=present update_cache=yes
```

Creating playbook for installing mysql-server package on database server

```
#  vi database.yml

---
- hosts: database
  become: true
  tasks:
    - name: Install mysql-server
      apt: name=mysql-server state=present update_cache=yes
```

Execute loadbalancer playbook

```
# ansible-playbook loadbalancer.yml
Execute it again
# ansible-playbook loadbalancer.yml

Execute Database playbook
# ansible-playbook database.yml
```

Creating webserver playbook, which covers loops and jinja2 templates
https://docs.ansible.com/ansible/playbooks_loops.html#standard-loops
http://jinja.pocoo.org/

vi webserver.yml

```
---
- hosts: webserver
  become: true
  tasks:
    - name: Install apache2
      apt: name={{item}} state=present update_cache=yes
      with_items:
        - apache2
        - libapache2-mod-wsgi
        - python-pip
        - python-virtualenv
```

Services modules

vi loadbalancer.yml

```
+   - name: Ensure nginx started
+     service: name=nginx state=started enabled=yes
```

Test the nginx service

```
# wget -qO- http://lb01 | less
```

Easier to do with curl, so install curl on the control server with playbook

```
# vi control.yml
```

```
---
- hosts: control
  become: true
  tasks:
    - name: install tools
      apt: name={{item}} state=present update_cache=yes
      with_items:
        - curl



# ansible-playbook control.yml
# curl lb01

Add service module to webserver and database playbooks
# vi webserver.yml

+    - name: Ensure Apache started
+      service: name=apache2 state=started enabled=yes

# vi database.yml
+    - name: Ensure Mysql started
+      service: name=mysql state=started enabled=yes

#  $ ansible-playbook webserver.yml

# ansible-playbook database.yml
```

Operational playbook | Support playbook | Stack Restart

We will need a playbook to manage services of the entire stack or manage cluster services.
vi stackrestart.yml

```
---
# Bring stack down
- hosts: loadbalancer
  become: true
  tasks:
    - service: name=nginx state=stopped

- hosts: webserver
  become: true
  tasks:
    - service: name=apache2 state=stopped

# Restart mysql
- hosts: database
  become: true
  tasks:
    - service: name=mysql state=restarted




# Bring stack up
- hosts: webserver
  become: true
  tasks:
    - service: name=apache2 state=started

- hosts: loadbalancer
  become: true
  tasks:
    - service: name=nginx state=started
```

apache2_module, handlers, notify

We need to setup the apache reciving python application. Python app gonna use mod-wsgi to serve the request. Make sure mod-wsgi is enabled with apache_module.
https://docs.ansible.com/ansible/apache2_module_module.html
https://docs.ansible.com/ansible/playbooks_intro.html#handlers-running-operations-on-change

```
# ansible-playbook webserver.yml
```

```
+     - name: Ensure mod_wsgi enables
+       apache2_module: state=present name=wsgi
+       notify: Restart apache2
+   handlers:
+     - name: Restart apache2
+       service: name=apache2 state=restarted
```

```
# ansible-playbook webserver.yml
```

Files: Copy

Copy the visualapp directory in the ansible directory which contains the python app written in flask.
Use copy module to ship application folder from control server to app01/02 servers.
Our visualapp site file also needs to be copied in /etc/apache2/sites-available directory.
We will enable out visualapp website in next section.
vi webserver.yml

```
+     - name: copy visualapp app source
+       copy: src=visualapp/app/ dest=/var/www/visualapp mode=0755
+       notify: Restart apache

+     - name: copy apache virtual host config
+       copy: src=visualapp/visualapp.conf dest=/etc/apache2/sites-available mode=0755
+       notify: Restart apache
```

ansible-playbook webserver.yml

curl app01
Still shows the default apache site for python app working we need to configure PIP and Virtualenv

Application Modules
To run the python flask application we need to install flask & SQLalchemy python package.
In order to install python packages we will use ‘pip’ which is a python package manager.
But we will not install the python packages directly on the system, we will create a virtual enviornment/container known as virtulenv which will hold all our python packages.

vi webserver.yml

```
+     - name: setup python virtualenv
+       pip: requirements=/var/www/visualapp/requirements.txt virtualenv=/var/www/visualapp/.venv
+       notify: Restart apache2
```

ansible-playbook webserver.yml

Files:File | Activating python site and deactivating apache default site

By default apache will serve a website, also known as apache default page.
We will disable the default website and enable our visualapp website.
In order to disable default website we need to unlink 000-default.conf file located in sites-enabled directory. Next we will create a link from /etc/apache2/sites-available/visualapp.conf to /etc/apache2/sites-enabled/visualapp.conf.

```
# vi webserver.yml

+   - name: de-activate default apache site
+     file: path=/etc/apache2/sites-enabled/000-default.conf state=absent
+     notify: Restart apache

+   - name: activate visualapp apache site
+     file: src=/etc/apache2/sites-available/visualapp.conf dest=/etc/apache2/sites-
enabled/visualapp.conf state=link
+     notify: Restart apache

# ansible-playbook webserver.yml
# curl app01
# curl app02
# curl lb01, will show the default page from nginx, we need to configure it to point to the App servers
```

Templates | Configure lb01 to point to app01 & app02
By default nginx will serve a website, also known as nginx default page.
We will disable the default website and enable a redirect rule which will forward request from nginx to our app servers randomly.

- In order to disable default website we need to unlink default file located in /etc/nginx/sites-enabled/ directory.
- Next we will create a template templates/nginx.conf.j2 as mentioned below and push that template to /etc/nginx/sites-available/visualapp location on lb01.
- Once we push visualapp configuration file in site-available directory we can create a link from /etc/nginx/sites-enabled/visualapp to /etc/nginx/sites-available/visualapp which will enable the nginx site.
- Nginx site is actually a redirect rule which will forward the requests to app01 & app02 randomly

```
# curl lb01
# mkdir templates
# vi templates/nginx.conf.j2
+upstream visualapp {
+{% for server in groups.webserver %}
+    server {{ server }};
+{% endfor %}
+}

+server {
+    listen 80;

+    location / {
+        proxy_pass http://visualapp;
+    }
+}

# vi loadbalancer.yml

+    - name: configure nginx site
+      template: src=templates/nginx.conf.j2 dest=/etc/nginx/sites-available/visualapp
mode=0644
+      notify: restart nginx

+    - name: de-activate default nginx site
+      file: path=/etc/nginx/sites-enabled/default state=absent
+      notify: restart nginx

+    - name: activate visualapp nginx site
+      file: src=/etc/nginx/sites-available/visualapp dest=/etc/nginx/sites-
enabled/visualapp state=link
+      notify: restart nginx

+  handlers:
```

```
+    - name: restart nginx
+      service: name=nginx state=restarted

# ansible-playbook loadbalancer.yml

Test if its working.
# curl lb01
```

Lineinfile | Make db server to listen to on all interface 0.0.0.0.

https://docs.ansible.com/ansible/lineinfile_module.html

Test the connection first
curl app01/db

- Check visualapp.wsgi which has db connection info.
- Check visualapp.py which establishes connection from app to db server
- Login to db server and see myql listening only on local interface
netstat -an
- Open /etc/mysql/my.cnf
With ansible lineinfile module replace “bind-address 127.0.0.1” to bind-address to “bind-address 0.0.0.0”

Login to control server

```
# vi database.yml

+    - name: ensure mysql listening on all ports
+      lineinfile: dest=/etc/mysql/my.cnf regexp=^bind-address
+                  line="bind-address = 0.0.0.0"
+
+      notify: restart mysql

+  handlers:
+    - name: restart mysql
+      service: name=mysql state=restarted

# ansible-playbook database.yml
```

Application Modules: mydql_db, mysql_user | Install python-mysqldb in app and db server, create visualapp db and visualapp user

Install mysql module in apache server
vi webserver.yml

```
  - name: Install apache2
      apt: name={{item}} state=present update_cache=yes
      with_items:
        - apache2
        - libapache2-mod-wsgi
        - python-pip
        - python-virtualenv
  +    - python-mysqldb

# ansible-playbook webserver.yml
```

https://docs.ansible.com/ansible/mysql_db_module.html
https://docs.ansible.com/ansible/mysql_user_module.html

```
# vi database.yml

- hosts: database
   become: true
   tasks:
+    - name: install tools
+      apt: name={{item}} state=present update_cache=yes
+      with_items:
+        - python-mysqldb
+
     - name: install mysql-server
       apt: name=mysql-server state=present update_cache=yes


       lineinfile: dest=/etc/mysql/my.cnf regexp=^bind-address line="bind-address =
0.0.0.0"
       notify: restart mysql

+    - name: create visualapp database
+      mysql_db: name=visualapp state=present
+
+    - name: create visualapp user
+      mysql_user: name=visualapp password=visualapp priv=visualapp.*:ALL host='%'
state=present
+
   handlers:
     - name: restart mysql
       service: name=mysql state=restarted

Execute db playbook
# ansible-playbook database.yml

Test the connection

# curl app01/db
# curl app02/db
# curl lb01/db
# curl lb01/db
# curl lb01/db
# curl lb01/db

Support Playbook 2 – Stack Status: wait_for | Playbook to check the current status of
all the services in stack. Also modifying stackrestart playbook with wait_for module.
```

Create stack_status.yml

```
# vi stack_status.yml
+---
+- hosts: loadbalancer
+  become: true
+  tasks:
+    - name: verify nginx service
+      command: service nginx status
+
+    - name: verify nginx is listening on 80
+      wait_for: port=80 timeout=1
+
+- hosts: webserver
+  become: true
+  tasks:
+    - name: verify apache2 service
+      command: service apache2 status
+
+    - name: verify apache2 is listening on 80
+      wait_for: port=80 timeout=1
+
+- hosts: database
+  become: true
+  tasks:
+    - name: verify mysql service
+      command: service mysql status
+
+    - name: verify mysql is listening on 3306
+      wait_for: port=3306 timeout=1

# ansible-playbook stack_status.yml
```

Modify stackrestart.yml and include wait_for module.
wait_for module is helpful in our stack restart playbook to cleanly bring down and bring up services.

7. When we stop nginx we can wait untill all the connections are cleanly closed(drained) from port 80.
8. When we stop apache we can wait untill it stops responding on port 80.
9. When we start the nginx,apache & mysql we can wait untill it starts responding on their respective ports.

```
# vi stackrestart.yml

  become: true
   tasks:
     - service: name=nginx state=stopped
+    - wait_for: port=80 state=drained

 - hosts: webserver
   become: true
   tasks:
     - service: name=apache2 state=stopped
+    - wait_for: port=80 state=stopped

 # Restart mysql
 - hosts: database
   become: true
   tasks:
     - service: name=mysql state=restarted
+    - wait_for: port=3306 state=started

 # Bring stack up
 - hosts: webserver
   become: true
```

```
   tasks:
     - service: name=apache2 state=started
+    - wait_for: port=80

 - hosts: loadbalancer
   become: true
   tasks:
     - service: name=nginx state=started
+    - wait_for: port=80
```

Support Playbook 2 – Stack Status: uri,register,fall,when |
When we check the status of our stacks services we are blindly trusting that its returning the right content. We just dont want to rely on the service status but actully will check if its returning the right content.
Install python-httplib2 package on loadbalancer(nginx) which is requirment to do the test.
We will install python-httplib2 package on loadbalancer because we are going to do test from the loadbalancer to the app servers.

https://docs.ansible.com/ansible/uri_module.html

```
# vi loadbalancer.yml

 - hosts: loadbalancer
   become: true
   tasks:
+    - name: install tools
+      apt: name={{item}} state=present update_cache=yes
+      with_items:
+        - python-httplib2
+
     - name: install nginx
       apt: name=nginx state=present update_cache=yes
```

```
We will be also doing test from control server so we will install  python-httplib2 on control server also.

# vi control.yml
       apt: name={{item}} state=present update_cache=yes
       with_items:
         - curl
+        - python-httplib2
```

Edit content of stack_status.yml to test the connection from control and loadbalancer(backend).
Connection will be tested by checking the content returned by hitting http://lb01, http://app01, http://app02/, http://lb01,db, http://app01db, http://app02/db

- We will use uri module and get the content of the website.
- Store the content in a varibale using register module.
- Stop the playbook execution if the content in variable is not righ using, fail & when module

```
# vi stack_status.yml

    - name: verify mysql is listening on 3306
      wait_for: port=3306 timeout=1
+
+- hosts: control
+  tasks:
+    - name: verify end-to-end index response
+      uri: url=http://{{item}} return_content=yes
+      with_items: groups.loadbalancer
+      register: lb_index
+
+    - fail: msg="index failed to return content"
+      when: "'Hello, from VisualPath' not in item.content"
+      with_items: "{{lb_index.results}}"
+
+    - name: verify end-to-end db response
+      uri: url=http://{{item}}/db return_content=yes
```

```
+      with_items: groups.loadbalancer
+      register: lb_db
+
+    - fail: msg="db failed to return content"
+      when: "'Database Connected from' not in item.content"
+      with_items: "{{lb_db.results}}"
+
+- hosts: loadbalancer
+  tasks:
+    - name: verify backend index response
+      uri: url=http://{{item}} return_content=yes
+      with_items: groups.webserver
+      register: app_index
+
+    - fail: msg="index failed to return content"
+      when: "'Hello, from VisualPath {{item.item}}!' not in item.content"
+      with_items: "{{app_index.results}}"
+
+    - name: verify backend db response
+      uri: url=http://{{item}}/db return_content=yes
+      with_items: groups.webserver
+      register: app_db
+
+    - fail: msg="db failed to return content"
+      when: "'Database Connected from {{item.item}}!' not in item.content"
+      with_items: "{{app_db.results}}"

# ansible-plabook stack_status.yml
```

Roles:

We have three tier in our app, for every tier we have a seprate playbook.
If we have a another team, they have their application which they want to manage with ansible.
How much of our existing code is reusable to our new team?
We can copy all of our playbooks paste them and the go through each line and edit every place we put visualapp and replace that with new application detail.
That is very tedious and especially we have more application coming down the line.
Also if they have diffrent config like they want to use diffrent ports.
If we want to make changes to our database configuration, we got many places in all our playbooks that require changes. Lot of code duplication thier.

Ansible gives us Roles to solve such problems. Essentially we need to create a folder for every roles which will have sub folders like tasks, files, handlers, vars etc, which holds all the information seprately. For example all the tasks that we have written in our playbooks goes into tasks folder. Now instead of having all the configuration for our tiers mashed up in one playbook we can distribute it into multiple folders or files.

- Encapsulation
- Code Resusablity
- Scalablity
- Ansible galaxy roles

```
# mkdir -p /home/vagrant/repo/ansible-roles
# cp -r /home/vagrant/repo/ansible/* /home/vagrant/repo/ansible-roles/
# cd /home/vagrant/repo/ansible-roles
# mkdir roles
# cd roles

Create skeleton of roles with ansible-galaxy

#  ansible-galaxy init control
#  ansible-galaxy init mysql
#  ansible-galaxy init nginx
#  ansible-galaxy init apache2
#  ansible-galaxy init visualapp_app
```

Need to move content of all the playbooks to their respective roles

1. **control.yml:**
   Move lists of tasks to control role's tasks/main.yml
   Replace tasks section from the control playbook with roles content.

```
# vi control.yml
---
- hosts: control
  become: true
  roles:
    - control
```

Execute control.yml to test.

```
# ansible-playbook control.yml
```

2. **database.yml**:
   Move lists of tasks to mysql role's tasks/main.yml.
   Move mysql start task after my.cnf file change section.
   Move lists of handlers to mysql role's handlers/main.yml.
   Replace tasks & handlers section from the playbook with roles content.

```
---
- hosts: database
  become: true
  roles:
    - mysql
```

3. **loadbalancer.yml**
   Move lists of tasks to nginx role's tasks/main.yml.
   Move lists of handlers to nginx role's handlers/main.yml.
   Move nginx.conf.j2 template to nginx role's templates directory
   Open tasks/main.yml of the nginx role, change templates src path
   from src= templates/nginx.conf.j2 to src=nginx.conf.j2
   Replace tasks & handlers section from the playbook with roles content.

```
---
- hosts: loadbalancer
  become: true
  roles:
     - nginx
```

4. **webserver.yml**
   Move lists of apache tasks to apache role's tasks/main.yml.
   Move apache2 start task to the end of the.
   Move lists of apache handlers to apache role's handlers/main.yml.
   Move lists of app tasks to visualapp_app role's tasks/main.yml.
   Move lists of app handlers to visualapp_app role's handlers/main.yml.
   Copy visualapp app directory to visualapp_app role's files/

   Replace tasks & handlers section from the playbook with roles content.

```
---
- hosts: webserver
  become: true
  roles:
   - apache2
   - visualapp_app
```

Site.yml: include | include all the playbook into site.yml
Create site.yml at the same place where all the playbooks are located

```
# vi site.yml
```

```
---
- include: control.yml
- include: database.yml
- include: webserver.yml
- include: loadbalancer.yml

# ansible-playboook site.yml
```

Variables: facts | Will use fact variable to make mysql listen only on its own ip address and not on 0.0.0.0

Get the list of all the fact variable with setup module
```
# ansible -m setup database
```

We will use ansible_eth0.ipv4.address fact variable in our lineinfile module for the bind-address instead of 0.0.0.0

Edit database role's tasks
```
# vi roles/mysql/tasks/main.yml

 - name: ensure mysql listening on all ports
-  lineinfile: dest=/etc/mysql/my.cnf regexp=^bind-address line="bind-address =
0.0.0.0"
+  lineinfile: dest=/etc/mysql/my.cnf regexp=^bind-address
+              line="bind-address = {{ ansible_eth0.ipv4.address }}"
   notify: restart mysql

 - name: ensure mysql started
```

Execute stack_status.yml
```
# ansible-playbook stack_status.yml
```

We also need to update the wait_for module in stack_status playbook.
By default it will wait for the loopback address 127.0.0.1. Since we updated mysql config to only listen on its ipaddress we need to make change in stack_status playbook as well.

Update stack_status.yml
```
# vi playbooks/stack_status.yml

- name: verify mysql is listening on 3306
-      wait_for: port=3306 timeout=1
+      wait_for: host={{ ansible_eth1.ipv4.address }} port=3306 timeout=1
```

Also update wait_for module in stack_restart playbook.

Update stack_restart.yml
```
# vi playbooks/stack_restart.yml

   become: true
   tasks:
     - service: name=mysql state=restarted
-    - wait_for: port=3306 state=started
+    - wait_for: host={{ ansible_eth1.ipv4.address }} port=3306 state=started
```

Variables: defaults | Using variable in mysql_db & mysql_user module

We have hard coded db config parameter like user, password, dbname etc in the playbook which is not modular, if we want to change a paramater then we need to go over the entire playbook and change every place where we mentioned that parameter like db name.
We will use Role's defaults directory and put all our db config info into variables. We can then refer to these variables in our playbook.

Lowest priorities is for default vars, refer the doc.
https://docs.ansible.com/ansible/playbooks_variables.html#variable-precedence-where-should-i-put-a-variable

roles/mysql/defaults/main.yml

```
 ---
-# defaults file for mysql
+db_name: myapp
+db_user_name: dbuser
+db_user_pass: dbpass
+db_user_host: localhost
```

roles/mysql/tasks/main.yml

```
 - name: ensure mysql started
   service: name=mysql state=started enabled=yes

-- name: create visualapp database
-  mysql_db: name=visualapp state=present
+- name: create database
+  mysql_db: name={{ db_name }} state=present

-- name: create visualapp user
-  mysql_user: name=visualapp password=visualapp priv=visualapp.*:ALL host='%'
state=present
+- name: create user
+  mysql_user: name={{ db_user_name }} password={{ db_user_pass }}
priv={{ db_name }}.*:ALL
+              host='{{ db_user_host }}' state=present
```

Variables: Vars | Multiple ways/places to define vars, refer the doc.

https://docs.ansible.com/ansible/playbooks_variables.html#variable-precedence-where-should-i-put-a-variable

Already defined vars in mysql role's defaults/main.yml which has the last precedence and would be used if vars are nowhere defined. We can override these varibles by defining them while refering to the role in our database.yml playbook
We will define vars in database.yml playbook with a hash.

In this case varible defined in below playbook will have higher precedence than our default variable defined in above section.

```
# vi database.yml
- hosts: database
   become: true
   roles:
-     - mysql
+     - { role: mysql, db_name: visualapp, db_user_name: visualapp, db_user_pass:
visualapp, db_user_host: '%' }
```

Variables: with_dict | Use of with_dict module in nginx.yml playbook.
We can also provide variables to playbooks in list format as mentioned below.
Will provide lists of variables in nginx role's defaults/main.yml. As specified below for reference. Name of the list is "sites".

```
---
# defaults file for nginx
sites:
    myapp:
        frontend: 80
        backend: 80
```

Will refer to those variables with the module with_dict in nginx.yml playbook. As specified below for reference.
with_dict: sites

with_dict will supply variable name in hash/dictionary format, variable names are {{ item.key }} & {{ item.value.frontend }} & {{ item.value.backend }}

```
(item={'value': {u'frontend': 80, u'backend': 80}, 'key': u'myapp'})
```

Update nginx role's defaults/main.yml

```
+sites:
+  myapp:
+    frontend: 80
+    backend: 80
```

Update nginx role's tasks/main.yml

```
 - name: install nginx
   apt: name=nginx state=present update_cache=yes

-- name: configure nginx site
-  template: src=nginx.conf.j2 dest=/etc/nginx/sites-available/visualapp mode=0644
+- name: configure nginx sites
+  template: src=nginx.conf.j2 dest=/etc/nginx/sites-available/{{ item.key }} mode=0644
+  with_dict: sites
   notify: restart nginx

 - name: de-activate default nginx site
   file: path=/etc/nginx/sites-enabled/default state=absent
   notify: restart nginx

-- name: activate visualapp nginx site
-  file: src=/etc/nginx/sites-available/visualapp dest=/etc/nginx/sites-enabled/visualapp
state=link
+- name: activate nginx sites
+  file: src=/etc/nginx/sites-available/{{ item.key }} dest=/etc/nginx/sites-enabled/{{ item.key
}} state=link
+  with_dict: sites
```

```
   notify: restart nginx
```

Update nginx role's templates/nginx.conf.j2

```
-upstream visualapp {
+upstream {{ item.key }} {
 {% for server in groups.webserver %}
-    server {{ server }};
+    server {{ server }}:{{ item.value.backend }};
 {% endfor %}
 }

 server {
-    listen 80;
+    listen {{ item.value.frontend }};

     location / {
-        proxy_pass http://visualapp;
+        proxy_pass http://{{ item.key }};
     }
 }
```

Selective Removal: shell, register, with_items, when

When we configure nginx, we find there is already a default website that we need to unlink.
/etc/nginx/sites-enabled/default linked to /etc/nginx/sites-available/default

We need to unlink all the websites except ours visualapp.
From nginx role's tasks/main.yml we will find all the nginx enabled sites
(soft link in /etc/nginx/sites-enabled) and unlink those which we do not want now
(visualapp -> /etc/nginx/sites-available/visualapp)

```
# vi roles/nginx/tasks/main.yml
-- name: de-activate default nginx site
-  file: path=/etc/nginx/sites-enabled/default state=absent
+- name: get active sites
+  shell: ls -1 /etc/nginx/sites-enabled
+  register: active
+
+- name: de-activate sites
+  file: path=/etc/nginx/sites-enabled/{{ item }} state=absent
+  with_items: active.stdout_lines
+  when: item not in sites
   notify: restart nginx

 - name: activate nginx sites
```

Variables continued | will push visualapp.wsgi as a template from visualapp_app role
We have replaced hardcoded DB values in our playbook as variables.
We also have db config parameters hardcoded in visualapp.wsgi file from our python flask app.
We will push visualapp.wsgi as a template and put all our db varibales into that template.

Move visualapp.wsgi to visualapp_app role's templates directory

```
# mv roles/visualapp_app/files/visualapp/app/visualapp.wsgi
roles/visualapp_app/templates/visualapp.wsgi.j2
```

```
Update the visualapp.wsgi.j2 with db variables names.

activate_this = '/var/www/visualapp/.venv/bin/activate_this.py'
execfile(activate_this, dict(__file__=activate_this))

import os
+os.environ['DATABASE_URI'] = 'mysql://{{ db_user }}:{{ db_pass }}@{{ groups.database[0] }}/
{{ db_name }}'

import sys
sys.path.insert(0, '/var/www/visualapp')

from visualapp import app as application

Refer the template visualapp.wsgi.j2 in visualapp_app role's tasks/main.yml

# vi roles/visualapp_app/tasks/main.yml

   copy: src=visualapp/app/ dest=/var/www/visualapp mode=0755
   notify: restart apache2


+- name: copy visualapp.wsgi
+  template: src=visualapp.wsgi.j2 dest=/var/www/visualapp/visualapp.wsgi mode=0755
+  notify: restart apache2
+
 - name: copy apache virtual host config
   copy: src=visualapp/visualapp.conf dest=/etc/apache2/sites-available mode=0755
   notify: restart apache2







Put the variables name in webserver.yml playbook where we refer the nginx role

# vi webserver.yml
  become: true
   roles:
     - apache2
-    - visualapp_app
+    - { role: visualapp_app, db_user: visualapp, db_pass: visualapp, db_name: visualapp }

# ansible-playbook webserver.yml
```

Variables: vars_files, group_vars | Use group_vars/all file to specify variables for database

Database variables are referred in two places
visualapp.wsgi.j2 template from visualapp_app role &
tasks/main.yml from mysql role.

We will use group_vars/all file to specify below variables
db_name: visualapp
db_user: visualapp
db_pass: visualapp

```
# mkdir group_vars
# vi group_vars/all

+---
+db_name: visualapp
+db_user: visualapp
+db_pass: visualapp
```

Remove db variables from websever.yml playbook

```
# vi webserver.yml
```

```
   become: true
   roles:
     - apache2
-    - { role: visualapp_app, db_user: visualapp, db_pass: visualapp, db_name:
visualapp }
+    - visualapp_app
```

Refer db variables from group_vars/all file inside database.yml playbook also called as variable routing. Variable routing can de defined as one variable pointing to another variable.
E:g db_user_name: “{{ db_user }}”

```
# vi database.yml

 - hosts: database
   become: true
   roles:
-    - { role: mysql, db_name: visualapp, db_user_name: visualapp, db_user_pass:
visualapp, db_user_host: '%' }
+    - role: mysql
+      db_user_name: "{{ db_user }}"
+      db_user_pass: "{{ db_pass }}"
+      db_user_host: '%'
```

Variables: vautl | encrypting db_pass variables value with vault

https://docs.ansible.com/ansible/playbooks_vault.html

Instead of having all file for variables, we will create all directory and move our variables in all directory.

```
# cd group_vars
# mv all vars
# mkdir all
# mv vars all
# export EDITOR=vim
```

Change directory to all directory and create vault. Vault file should be present in all directory for our scenario.

```
# cd all
# ansible-vault create vault
Give a vault password and put below mentioned content.

+ ---
+ vault_db_pass : admin123

Refer to vault_db_pass in all/vars file

# vi vars
+---
+db_name: visualapp
+db_user: visualapp
+db_pass: "{{ vault_db_pass }}"
```

Execute database.yml playbook, you should get some error like below

```
# ansible-playbook database.yml
```

ERROR! Decryption failed
ERROR! A vault password must be specified

Either you can give –ask-vault-pass option and execute the playbook which will ask you the vaultpassword or you can use a file where you specify vault password.

“vaultpass” is our vault password.

```
# echo "vaultpass" > ~/.vault_pass.txt
# chmod 0600 ~/.vault_pass.txt
# vi ansible.cfg
 [defaults]
 inventory = ./dev
+vault_password_file = ~/.vault_pass.txt
```

Execute database playbook to test it
```
# ansible-playbook database.yml
```

Login to db01 instance and login to mysql database to verify.
```
# ssh db01
# mysql -h localhost -u visualapp -p
```

Advanced Execution
Advanced Execution: gather_facts | By disabling gather_facts we can save the execution time.

```
Time the execution with time command for site.yml, stack_status and stack_restart.yml
# time ansible-plabook site.yml
# time ansible-plabook stack_status.yml
# time ansible-plabook stack_restart.yml

Disable gather_facts for all the plays except dbserver because we use fact varible for dbservers
play.
ansible_eth1.ipv4.address

control.yml

 ---
 - hosts: control
   become: true
+  gather_facts: false
   roles:
     - control

loadbalancer.yml

 ---
 - hosts: loadbalancer
   become: true
+  gather_facts: false
   roles:
```

```
     - nginx
hostname.yml

 ---
   - hosts: all
+    gather_facts: false
     tasks:
       - name: get server hostname
         command: hostname

stack_restart.yml

 # Bring stack down
 - hosts: loadbalancer
   become: true
+  gather_facts: false
   tasks:
     - service: name=nginx state=stopped
     - wait_for: port=80 state=drained


 - hosts: webserver
   become: true
+  gather_facts: false
   tasks:
     - service: name=apache2 state=stopped
     - wait_for: port=80 state=stopped

 # Bring stack up
 - hosts: webserver
   become: true
+  gather_facts: false
   tasks:
     - service: name=apache2 state=started
     - wait_for: port=80


 - hosts: loadbalancer
   become: true
+  gather_facts: false
   tasks:
     - service: name=nginx state=started
     - wait_for: port=80

stack_status.yml

 ---
 - hosts: loadbalancer
   become: true
+  gather_facts: false
   tasks:
     - name: verify nginx service
       command: service nginx status



 - hosts: webserver
   become: true
+  gather_facts: false
   tasks:
     - name: verify apache2 service
       command: service apache2 status

       wait_for: host={{ ansible_eth0.ipv4.address }} port=3306 timeout=1


 - hosts: control
+  gather_facts: false
   tasks:
     - name: verify end-to-end index response
       uri: url=http://{{item}} return_content=yes

       with_items: "{{lb_db.results}}"


 - hosts: loadbalancer
+  gather_facts: false
   tasks:
```

```
     - name: verify backend index response
       uri: url=http://{{item}} return_content=yes

webserver.yml

 ---
 - hosts: webserver
   become: true
+  gather_facts: false
   roles:
     - apache2



Time the execution again with time command to verify.
```

Extracting Repetitive Tasks: cache_valid_time

Open site.yml and add below mentioned play

```
 ---
+- hosts: all
+  become: true
+  gather_facts: false
+  tasks:
+    - name: update apt cache
+      apt: update_cache=yes cache_valid_time=86400
+
 - include: control.yml
 - include: database.yml
 - include: webserver.yml
```

Remove update_cache=yes parameter from all the tasks/main.yml of all the roles.

```
# vi roles/mysql/tasks/main.yml
E:g
-  apt: name=mysql-server state=present update_cache=yes
+  apt: name=mysql-server state=present
```

Limiting Execution by Hosts : limit

```
# ansible-playbook site.yml –limit app01
```

Limiting Execution by Tasks : tags

We can also select particular task or tasks by tagging them and then using tag name while executing playbooks or site.yml.

```
# vi roles/control/tasks/main.yml
   with_items:
     - apache2
     - libapache2-mod-wsgi
+  tags: [ 'packages' ]
```

List the available tags.

```
#  $ ansible-playbook site.yml –list-tags
```

Execute it to verify

```
# ansible-playbook site.yml –tags “packages”
```

We can also skip the specific tags and execute the rest.
$ ansible-playbook site.yml –skip-tags “packages”

Tag all the tasks as per our 4 principles of app deployment on Linux systems
[‘packages’]
[‘service’]
[‘system’]
[‘configure’]

```
# vi roles/apache2/tasks/main.yml

   with_items:
     - apache2
     - libapache2-mod-wsgi
+  tags: [ 'packages' ]


 - name: ensure mod_wsgi enabled
   apache2_module: state=present name=wsgi
   notify: restart apache2
+  tags: [ 'system' ]


 - name: de-activate default apache site
   file: path=/etc/apache2/sites-enabled/000-default.conf state=absent
   notify: restart apache2
+  tags: [ 'system' ]


 - name: ensure apache2 started
   service: name=apache2 state=started enabled=yes
+  tags: [ 'service' ]

# vi roles/control/tasks/main.yml

   with_items:
     - curl
     - python-httplib2
+  tags: [ 'packages' ]

# vi roles/visualapp_app/tasks/main.yml

     - python-pip
     - python-virtualenv
     - python-mysqldb
+  tags: [ 'packages' ]


 - name: copy visualapp app source
   copy: src=visualapp/app/ dest=/var/www/visualapp mode=0755
   notify: restart apache2
+  tags: [ 'configure' ]


 - name: copy visualapp.wsgi
   template: src=visualapp.wsgi.j2 dest=/var/www/visualapp/visualapp.wsgi mode=0755
   notify: restart apache2
+  tags: [ 'configure' ]


 - name: copy apache virtual host config
   copy: src=visualapp/visualapp.conf dest=/etc/apache2/sites-available mode=0755
   notify: restart apache2
+  tags: [ 'configure' ]


 - name: setup python virtualenv
   pip: requirements=/var/www/visualapp/requirements.txt virtualenv=/var/www/visualapp/.venv
   notify: restart apache2
+  tags: [ 'system' ]


 - name: activate visualapp apache site
   file: src=/etc/apache2/sites-available/visualapp.conf dest=/etc/apache2/sites-
enabled/visualapp.conf state=link
   notify: restart apache2
+  tags: [ 'configure' ]

# vi roles/mysql/tasks/main.yml
```

```
   apt: name={{item}} state=present
   with_items:
     - python-mysqldb
+  tags: [ 'packages' ]

 - name: install mysql-server
   apt: name=mysql-server state=present
+  tags: [ 'packages' ]

 - name: ensure mysql listening on all ports
   lineinfile: dest=/etc/mysql/my.cnf regexp=^bind-address
               line="bind-address = {{ ansible_eth0.ipv4.address }}"
   notify: restart mysql
+  tags: [ 'configure' ]

 - name: ensure mysql started
   service: name=mysql state=started enabled=yes
+  tags: [ 'service' ]

 - name: create database
   mysql_db: name={{ db_name }} state=present
+  tags: [ 'configure' ]

 - name: create user
   mysql_user: name={{ db_user_name }} password={{ db_user_pass }} priv={{ db_name }}.*:ALL
               host='{{ db_user_host }}' state=present
+  tags: [ 'configure' ]

# vi roles/nginx/tasks/main.yml

   apt: name={{item}} state=present
   with_items:
     - python-httplib2
+  tags: [ 'packages' ]

 - name: install nginx
   apt: name=nginx state=present
+  tags: [ 'packages' ]

 - name: configure nginx sites
   template: src=nginx.conf.j2 dest=/etc/nginx/sites-available/{{ item.key }} mode=0644
   with_dict: sites
   notify: restart nginx
+  tags: [ 'configure' ]

 - name: get active sites
   shell: ls -1 /etc/nginx/sites-enabled
   register: active
+  tags: [ 'configure' ]

 - name: de-activate sites
   file: path=/etc/nginx/sites-enabled/{{ item }} state=absent
   with_items: active.stdout_lines
   when: item not in sites
   notify: restart nginx
+  tags: [ 'configure' ]

 - name: activate nginx sites
   file: src=/etc/nginx/sites-available/{{ item.key }} dest=/etc/nginx/sites-enabled/{{ item.key
}} state=link
   with_dict: sites
   notify: restart nginx
+  tags: [ 'configure' ]

 - name: ensure nginx started
   service: name=nginx state=started enabled=yes
+  tags: [ 'service' ]

# vi site.yml
```

```
   tasks:
     - name: update apt cache
       apt: update_cache=yes cache_valid_time=86400
+      tags: [ 'packages' ]

 - include: control.yml
 - include: database.yml.
```

Save execution time by skipping packages tags
time $ ansible-playbook site.yml --skip-tags “packages”

Troubleshooting, Testing & Validation ordering problem

Make sure to have mysql service start task after you changed the mysql configuration.
If there is some typo or bad config in my.cnf, notify mysql restart is going to stop mysql and while starting it will fail. If you run the playbook again first it will hit mysql start task which will fail to start the mysql service and your play is not going to reach to the task where it pushes new my.cnf file to fix the issue. Ordering is very important in such scenario.

- name: Install mysql-server
  apt: name=mysql-server state=present

- name: ensure mysql listening on all ports

```
  lineinfile: dest=/etc/mysql/my.cnf regexp=^bind-address
              line="bind-address = {{ansible_eth1.ipv4.address}}"

  notify: restart mysql

- name: Ensure Mysql started
  service: name=mysql state=started enabled=yes
```

Jumping to Specific tasks: list-tasks, step, start-at-task

https://docs.ansible.com/ansible/playbooks_startnstep.html

```
# ansible-playbook site.yml –step
```

Select y for yes, n for no and c to continue with the execution without asking the prompt.

List all the tasks from the playbook and select the task from where you want to start the execution.

```
# ansible-playbook site.yml –list-tasks

# ansible-playbook site.yml –start-at-task “copy visualapp app source”
```

Retrying failed hosts

```
# ansible-playbook site.yml –limit @/home/ansible/site.retry
```

Syntax-Check & Dry-Run: syntax-check, check

```
# ansible-playbook –syntax-check site.yml
# ansible-playbook --check site.yml
```