Writing Ansible Playbooks

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

## 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 the vars 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 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 the var 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](http://jinja.pocoo.org/docs/dev/) 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 14.04 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](https://github.com/erikaheidi/cfmgmt/tree/master/ansible). The folder also contains a Vagrantfile that lets you test the playbook in a simplified setup, using a virtual machine managed by [Vagrant](https://vagrantup.com/).

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.