Mastering

ANSIBLE

Section 02

Using Ansible with Vagrant for fast provisioning

Integrating Ansible with Vagrant

- In the last section we introduced Vagrant as a provisioning platform that works on top of a virtualization platform (like VirtualBox). You saw how easy it was to fire up multiple machines at the same time, change their network configuration and so on.

- Additionally, and as a provisioning platform of it own, Vagrant can be integrated with CM tools like Chef, Puppet, Salt and of course Ansible.

- Open the Vagrantfile and add the following lines to "client" machine configuration part:

```
client.vm.provision "ansible" do |ansible|
  ansible.playbook = "myplaybook.yml"
end
```

- What you've just did now is that you instructed Vagrant to bootstrap "client" machine and configure it using Ansible. Ansible in turn will be using the *playbook* that you specified: myplaybook.yml

- A playbook is nothing by a YAML file that contains instructions for Ansible to work. In our example, this file should exist in the shared directory between the Vagrant machine and the host. This is - by default - the directory where the Vagrantfile exists, and it is referred to from inside the machine as /vagrant. But what would myplaybook.yml contain?

The YAML format

- YAML stands for YAML Ain't Markup Language. Yes, it is one of those recursive acronyms like GNU Not Unix (what GNU stands for!).
- It is an international format used for data serialization. You can think of it as the counterpart of JSON.
- A YAML document can start with triple dashes (`---`) and end with triple dots (...).
- The key/value pairs can be nested as long as they are in the same indentation level. For example:
```
packages:
  mandatory:
    name: vim
    description: Vi Improved
```
- It often contains a list of items (called a hash or dictionary). The first line contains the dictionary name followed by a colon. Then the list items are written on the subsequent lines each on a line starting with a dash and a space. You can think of lists as arrays in JSON For example:
```
servers:
  - db
  - web
```
- It can also contain key: value. A colon and a space separates the key and the value. For example:
```
db: role=database vendor=mysql
```
- You can also quote the values if you want to escape special characters like colons.
- And always remember: **never use tabs in a YAML file only spaces are allowed**

The myplaybook.yml

- Now that you know the basics of YAML, let's write our myplaybook.yml playbook. Open the file in the directory containing the Vagrantfile. Using your favorite editor, write the following:

```
- hosts: all
  become: yes
  tasks:
   - name: Ensure that Apache webserver is installed
     yum: name=httpd state=present
   - name: Ensure that the service is running and persistent
     service: name=httpd state=started enabled=yes
```

- Before we delve into the details of this playbook and see what it does, let's first see how this compares to a JSON file (if you are more familiar with JSON):

Ok, and in JSON

- Although JSON is not used with Ansible, some people who are not familiar with YAML find YAML intimidating at first. So here is the JSON representation of the previous YAML text to help you understand the syntax:

```
[
  {
      "hosts"     : "all",
      "become"    : "yes",
      "tasks"     : [
          {
              "name" : "Ensure that Apache webserver is installed",
              "yum"  : "name=httpd state=present"
          },
          {
              "name"     : "Ensure that the service is running and persistent",
              "service"  : "name=httpd state=started enabled=yes"
          }
      ]
  }
]
```

Parsing myplaybook.yml

- Let's have a look at what each line of the playbook that we've just written does:
  - `- hosts: all` instructs Ansible to run on all hosts. Notice that Vagrant is the client here so it is using its own version of /etc/ansible/hosts. In our case it is only one VM.
  - `become: yes` because we are using commands that require root privileges, this line instructs Ansible to run the playbook using sudo.
  - `tasks:` this starts the array (or list or dictionary) that contains the tasks that will be run on each machine in the hosts (in our case it's only one).
  - `- name:` the name of the task. You should write a descriptive name so that the playbook will be well documented.
  - `yum: name=httpd state=present`. Yum is one of Ansible's many modules that are used to work with the host. This is the Red Hat package manager. The `name` parameter here is mandatory. It specifies the name of the package as it appears in the repository. The `state` parameter may be `present` (to ensure that the package is installed) or `absent` (to ensure that it is not installed and uninstall it if necessary).
  - `service: name=httpd state=started enabled=yes`. Again `service` is another Ansible module. It controls the service status and behavior. The required parameter here is `name` which specifies the name of the service as viewed on the OS. The `state` parameter defines the state in which the service should be (`started`, `stopped`, `restarted`, `reloaded`). You have to specify at least a `state` or an `enabled` parameter with the `service` module. The enabled parameter controls whether or not the service should start when the system boots.

Let's play

- Ok now everything is in place, we need to instruct Vagrant to provision the machine using Ansible.
- Since the machine has already been provisioned before (when you issued vagrant up), you need to inform Vagrant that it needs to run the provisioning task again with the new instructions.
- This can be done with either reloading the machine with `--provision` flag: `vagrant reload client --provision`, or if you don't want to reboot the machine you can just issue `vagrant provision client`.
- Now observe the output at the CLI. Ansible will give you the name of each task as it is doing it. After a few moments you'll find the name of the host (client) with the result of the run: `ok=3 changed=2 unreachable=0 failed=0`
- Time to check the results of the command, login to the machine using vagrant ssh client and run the following command: `rpm -qa | grep http`. You should typically find `httpd` and `httpd-tools` packages installed.

I could've done that using a shell script

- Sure you could've done that. Actually Vagrant supports shell scripts as a provisioning method just like Ansible, Puppet, Chef and Salt. But let's see why that won't be the best option to handle the task.

- First, you want to make sure that Apache web server is not installed before you attempt to run yum. Otherwise you will have to wait till yum queries the available repositories for the latest metadata. You could do something like this:
```
if ! rpm -qa | grep httpd >/dev/null; then yum -y install httpd; fi
```

- The problem with the above approach is that rpm -qa | grep httpd will also match httpd-tools. You can work around this by modifying the script to be as follows:
```
if  ! rpm -qa | grep httpd | grep -v httpd-tools >/dev/null ; then yum -y install httpd; fi
```

- But what if you have multiple packages that happen to be having "httpd" in their name? This means more and more grep -v commands. Of course their could be more elegant solutions for this problem using bash but it will always mean writing and maintaining more code. Additionally, you should inform the user with what the script is doing and also add comments whenever necessary. So the above script could look as follows:
```
# Checking for the existence of httpd
echo "Checking whether or not httpd is already installed"
if  ! rpm -qa | grep httpd | grep -v httpd-tools >/dev/null; then
    echo 'Installing httpd'
    #Installing httpd
    yum -y install httpd
else
    echo 'httpd package is already installed'
fi
```

- Compare this to the couple of lines that you added to the playbook earlier to see the difference.