Table of Contents

Ansible tutorial

This tutorial presents ansible step-by-step. You'll need to have a (virtual or physical) machine to act as an ansible node. A vagrant environment is provided for going through this tutorial.

Ansible is a configuration management software that let's you control and configure nodes from another machine. What makes it different from other management software is that ansible uses (potentially existing) SSH infrastructure, while others (chef, puppet, ...) need a specific PKI infrastructure to be set-up.

Ansible also emphasises push mode, where configuration is pushed from a master machine (a master machine is only a machine where you can SSH to nodes from) to nodes, while most other CM typically do it the other way around (nodes pull their config at times from a master machine).

This mode is really interesting since you do not need to have a 'publicly' accessible 'master' to be able to configure remote nodes: it's the nodes that need to be accessible (we'll see later that 'hidden' nodes can pull their configuration too!), and most of the time they are.

Prerequisites for Ansible

You need the following python modules on your machine (the machine you run ansible on)

- python-yaml
- python-jinja2

On Debian/Ubuntu run: `sudo apt-get install python-yaml python-jinja2 python-paramiko python-crypto`

We're also assuming you have a keypair in your ~/.ssh directory.

Installing Ansible

From source

Ansible devel branch is always usable, so we'll run straight from a git checkout. You might need to install git for this (`sudo apt-get install git` on Debian/Ubuntu).

```
git clone git://github.com/ansible/ansible.git
cd ./ansible
```

At this point, we can load the ansible environment:

```
source ./hacking/env-setup
```

From a deb package

When running from an installed package, this is absolutely not necessary. If you prefer running from a debian package ansible, provides a `make target` to build it. You need a few packages to build the deb:

```
sudo apt-get install make fakeroot cdbs python-support
git clone git://github.com/ansible/ansible.git
cd ./ansible
make deb
sudo dpkg -i ../ansible_1.1_all.deb (version may vary)
```

We'll assume you're using the deb packages in the rest of this tutorial.

Cloning the tutorial

```
git clone https://github.com/leucos/ansible-tuto.git
cd ansible-tuto
```

Using Vagrant with the tutorial

It's highly recommended to use Vagrant to follow this tutorial. If you don't have it already, setting up should be quite easy and is described in step-00/README.md.

If you wish to proceed without Vagrant (not recommended!), go straight to step-01/README.md.

Contents

- command or action: ansible module like just a shell command. Intro in step-02.
- task: it's combine an action (a module and its arguments) with a name and optionally some other keywords (like looping directives).
- playbook: an yaml file contains roles executed in sequence, and eventually individual tasks. Intro in step-04.
- role: an organisational unit grouping tasks together in order to install a piece of software. Intro in step-12.

Just in case you want to skip to a specific step, here is a topic table of contents.

Contributing

Thanks to all people who have contributed to this tutorial:

- Aladin Jaermann
- Alexis Gallagher
- Alice Ferrazzi
- Alice Pote
- Amit Jakubowicz
- Arbab Nazar
- Atilla Mas
- Ben Visser
- Benny Wong

- Bernardo Vale
- Chris Schmitz
- dalton
- Daniel Howard
- David Golden
- Eric Corson
- Eugene Kalinin
- Hartmut Goebel
- Jelly Robot
- Justin Garrison
- Karlo
- Marchenko Alexandr
- mxxcon
- Patrick Pelletier
- Pierre-Gilles Levallois
- Ruud Kamphuis
- [torenware] (https://github.com/torenware)
- Victor Boivie

(and sorry if I forgot anyone)

I've been using Ansible almost since it's birth, but I learned a lot in the process of writing it. If you want to jump in, it's a great way to learn, feel free to add your contributions.

The chapters being written live in the writing branch.

If you have ideas on topics that would require a chapter, please open a PR.

I'm also open on pairing for writing chapters. Drop me a note if you're interested.

If you make changes or add chapters, please fill the `test/expectations` file and run the tests (`test/run.sh`). See the `test/run.sh` file for (a bit) more information.

When adding a new chapter (e.g. `step-NN`), please issue:

```
cd step-99
ln -sf ../step-NN/{hosts,roles,site.yml,group_vars,host_vars} .
```

For typos, grammar, etc... please send a PR for the master branch directly.

Thank you!

Ansible tutorial

To make the tutorial self-contained, a Vagrant file is provided. Vagrant makes it easy to bootstrap barebones virtual machines with VirtualBox.

Installing Vagrant

In order to run Vagrant, you need:

- VirtualBox installed
- Ruby installed (should be on your system already)
- Vagrant 1.1+ installed (see http://docs.vagrantup.com/v2/installation/index.html).

This should be all it takes to set up Vagrant.

Now bootstrap your virtual machines with the following command. Note that you do not need to download any "box" manually. This tutorial already includes a `Vagrantfile` to get you up and running, and will get one for you if needed.

`vagrant up`

and go grab yourself a coffee (note that if you use vagrant-hostmaster, you'll need to type your password since it needs to sudo as root).

If something goes wrong, refer to Vagrant's Getting Started Guide.

Cautionary tale about NetworkManager

On some systems, NetworkManager will take over `vboxnet` interfaces and mess everything up. If you're in this case, you should prevent NetworkManager from trying to autoconfigure `vboxnet` interfaces. Just edit `/etc/NetworkManager/NetworkManager.conf` (or whatever the NetworkManager config is on your system) and add in section `[keyfile]`:

```
unmanaged-devices=mac:MAC_OF_VBOXNET0_IF;mac:MAC_OF_VBOXNET1_IF;...
```

Then destroy Vagrant machines, restart NetworkManager and try again.

Adding your SSH keys on the virtual

machines

To follow this tutorial, you'll need to have your keys in VMs root's `authorized_keys`. While this is not absolutely necessary (Ansible can use sudo, password authentication, etc...), it will make things way easier.

Ansible is perfect for this and we will use it for the job. However I won't explain what's happening for now. Just trust me.

```
ansible-playbook -c paramiko -i step-00/hosts step-00/setup.yml --ask-pass --become
```

When asked for password, enter *vagrant*. If you get "Connections refused" errors, please check the firewall settings of your machine.

To polish things up, it's better to have an ssh-agent running, and add your keys to it (`ssh-add`).

NOTE: We are assuming that you're using Ansible version v2 on your local machine. If not you should upgrade ansible to v2 before using this repository

To check your ansible version use the command `ansible --version`. The output should be similar to the above:

```
$ ansible --version
  ansible 2.0.0.2
    config file = /etc/ansible/ansible.cfg
    configured module search path = Default w/o overrides
```

Now head to the first step in step-01.

Ansible tutorial

Inventory

Before continuing, you need an inventory file. The default place for such a file is `/etc/ansible/hosts`. However, you can configure ansible to look somewhere else, use an environment variable (`ANSIBLE_HOSTS`), or use the `-i` flag in ansible commands an provide the inventory path.

We've created an inventory file for you in the directory that looks like this:

```
host0.example.org ansible_host=192.168.33.10 ansible_user=root
host1.example.org ansible_host=192.168.33.11 ansible_user=root
host2.example.org ansible_host=192.168.33.12 ansible_user=root
```

`ansible_host` is a special *variable* that sets the IP ansible will use when trying to connect to this host. It's not necessary here if you use the vagrant-hostmaster gem. Also, you'll have to change the IPs if you have set up your own virtual machines with different addresses.

`ansible_user` is another special *variable* that tells ansible to connect as this user when using ssh. By default ansible would use your current username, or use another default provided in ~/.ansible.cfg (`remote_user`).

Testing

Now that ansible is installed, let's check everything works properly.

```
ansible -m ping all -i step-01/hosts
```

What ansible will try to do here is just executing the `ping` module (more on modules later) on each host.

The output should look like this:

```
host0.example.org | success >> {
    "changed": false,
    "ping": "pong"
}

host1.example.org | success >> {
    "changed": false,
    "ping": "pong"
}

host2.example.org | success >> {
    "changed": false,
    "ping": "pong"
}
```

Good! All 3 hosts are alive and kicking, and ansible can talk to them.

Now head to next step in directory step-02.

Ansible tutorial

Talking with nodes

Now we're good to go. Let's play with the command we saw in the previous chapter: `ansible`. This command is the first one of three that ansible provides which interact with nodes.

Doing something useful

In the previous command, `-m ping` means "use module *ping*". This module is one of many available with ansible. `ping` module is really simple, it doesn't need any arguments. Modules that take arguments pass them via `-a` switch. Let's see a few other modules.

Shell module

This module lets you execute a shell command on the remote host:

```
ansible -i step-02/hosts -m shell -a 'uname -a' host0.example.org
```

Output should look like:

```
host0.example.org | success | rc=0 >>
Linux host0.example.org 3.2.0-23-generic-pae #36-Ubuntu SMP Tue Apr 10 22:19:09 UTC 20
12 i686 i686 i386 GNU/Linux
```

Easy!

Copy module

No surprise, with this module you can copy a file from the controlling machine to the node. Lets say we want to copy our `/etc/motd` to `/tmp` of our target node:

```
ansible -i step-02/hosts -m copy -a 'src=/etc/motd dest=/tmp/' host0.example.org
```

Output should look similar to:

```
host0.example.org | success >> {
    "changed": true,
    "dest": "/tmp/motd",
    "group": "root",
    "md5sum": "d41d8cd98f00b204e9800998ecf8427e",
    "mode": "0644",
    "owner": "root",
    "size": 0,
    "src": "/root/.ansible/tmp/ansible-1362910475.9-246937081757218/motd",
    "state": "file"
}
```

Ansible (more accurately *copy* module executed on the node) replied back a bunch of useful information in JSON format. We'll see how that can be used later.

We'll see other useful modules below. Ansible has a huge module list that covers almost anything you can do on a system. If you can't find the right module, writing one is pretty easy (it doesn't even have to be Python, it just needs to speak JSON).

Many hosts, same command

Ok, the above stuff is fun, but we have many nodes to manage. Let's try that on other hosts too.

Lets say we want to get some facts about the node, and, for instance, know which Ubuntu version we have deployed on nodes, it's pretty easy:

```
ansible -i step-02/hosts -m shell -a 'grep DISTRIB_RELEASE /etc/lsb-release' all
```

`all` is a shortcut meaning 'all hosts found in inventory file'. It would return:

```
host1.example.org | success | rc=0 >>
DISTRIB_RELEASE=12.04

host2.example.org | success | rc=0 >>
DISTRIB_RELEASE=12.04

host0.example.org | success | rc=0 >>
DISTRIB_RELEASE=12.04
```

Many more facts

That was easy. However, It would quickly become cumbersome if we wanted more information (ip addresses, RAM size, etc...). The solution comes from another really handy module (weirdly) called `setup` : it specializes in node's *facts* gathering.

Try it out:

```
ansible -i step-02/hosts -m setup host0.example.org
```

replies with lots of information:

```
"ansible_facts": {
    "ansible_all_ipv4_addresses": [
        "192.168.0.60"
    ],
    "ansible_all_ipv6_addresses": [],
    "ansible_architecture": "x86_64",
    "ansible_bios_date": "01/01/2007",
    "ansible_bios_version": "Bochs"
    },
    ---snip---
    "ansible_virtualization_role": "guest",
    "ansible_virtualization_type": "kvm"
},
"changed": false,
"verbose_override": true
```

It's been truncated for brevity, but you can find many interesting bits in the returned data. You may also filter returned keys, in case you're looking for something specific.

For instance, let's say you want to know how much memory you have on all your hosts, easy with `ansible -i step-02/hosts -m setup -a 'filter=ansible_memtotal_mb' all` :

```
host2.example.org | success >> {
    "ansible_facts": {
        "ansible_memtotal_mb": 187
    },
    "changed": false,
    "verbose_override": true
}

host1.example.org | success >> {
    "ansible_facts": {
        "ansible_memtotal_mb": 187
    },
    "changed": false,
    "verbose_override": true
}

host0.example.org | success >> {
    "ansible_facts": {
        "ansible_memtotal_mb": 187
    },
    "changed": false,
    "verbose_override": true
}
```

Notice that hosts replied in different order compared to the previous output. This is because ansible parallelizes communications with hosts!

BTW, when using the setup module, you can use `*` in the `filter=` expression. It will act like a shell glob.

Selecting hosts

We saw that `all` means 'all hosts', but ansible provides a lot of other ways to select hosts:

- `host0.example.org:host1.example.org` would run on host0.example.org and host1.example.org
- `host*.example.org` would run on all hosts starting with 'host' and ending with '.example.org' (just like a shell glob too)

There are other ways that involve groups, we'll see that in step-03.

Ansible tutorial

Grouping hosts

Hosts in inventory can be grouped arbitrarily. For instance, you could have a `debian` group, a `web-servers` group, a `production` group, etc...

```
[debian]
host0.example.org
host1.example.org
host2.example.org
```

This can even be expressed shorter:

```
[debian]
host[0:2].example.org
```

If you wish to use child groups, just define a `[groupname:children]` and add child groups in it. For instance, let's say we have various flavors of linux running, we could organize our inventory like this:

```
[ubuntu]
host0.example.org

[debian]
host[1:2].example.org

[linux:children]
ubuntu
debian
```

Grouping of course, leverages configuration mutualization.

Setting variables

You can assign variables to hosts in several places: inventory file, host vars files, group vars files, etc...

I usually set most of my variables in group/host vars files (more on that later). However, I often use some variables directly in the inventory file, such as `ansible_host` which sets the IP address for the host. Ansible by default resolves hosts' name when it attempts to connect via SSH. But when you're bootstrapping a host, it might not have its definitive ip address yet. `ansible_host` comes in handy here.

When using `ansible-playbook` command (not the regular `ansible` command), variables can also be set with `--extra-vars` (or `-e`) command line switch. `ansible-playbook` command will be covered in the next step.

`ansible_port` , as you can guess, has the same function regarding the ssh port ansible will try to connect at.

```
[ubuntu]
host0.example.org ansible_host=192.168.0.12 ansible_port=2222
```

Ansible will look for additional variables definitions in group and host variable files. These files will be searched in directories `group_vars` and `host_vars` , below the directory where the main inventory file is located.

The files will be searched by name. For instance, using the previously mentioned inventory file, `host0.example.org` variables will be searched in those files:

- `group_vars/linux`
- `group_vars/ubuntu`
- `host_vars/host0.example.org`

It doesn't matter if those files do not exist, but if they do, ansible will use them.

Now that we know the basics of modules, inventories and variables, let's explore the real power of Ansible with playbooks.

Head to step-04.

Ansible tutorial

Ansible playbooks

Playbook concept is very simple: it's just a series of ansible commands (tasks), like the ones we used with the `ansible` CLI tool. These tasks are targeted at a specific set of hosts/groups.

The necessary files for this step should have appeared magically and you don't even have to type them.

Apache example (a.k.a. Ansible's "Hello World!")

We assume we have the following inventory file (let's name it `hosts`):

```
[web]
host1.example.org
```

and all hosts are debian-like.

Note: remember you can (and in our exercise we do) use `ansible_host` to set the real IP of the host. You can also change the inventory and use a real hostname. In any case, use a non-critical machine to play with! In the real hosts file, we also have `ansible_user=root` to cope with potential different ansible default configurations.

Lets build a playbook that will install apache on machines in the `web` group.

```
- hosts: web
  tasks:
    - name: Installs apache web server
      apt: pkg=apache2 state=installed update_cache=true
```

We just need to say what we want to do using the right ansible modules. Here, we're using the apt module that can install debian packages. We also ask this module to update the package cache.

We also added a name for this task. While this is not necessary, it's very informative when the playbook runs, so it's highly recommended.

All in all, this was quite easy!

You can run the playbook (lets call it `apache.yml`):

```
ansible-playbook -i step-04/hosts -l host1.example.org step-04/apache.yml
```

Here, `step-04/hosts` is the inventory file, `-l` limits the run only to `host1.example.org` and `apache.yml` is our playbook.

When you run the above command, you should see something like:

```
PLAY [web] *********************

GATHERING FACTS *********************
ok: [host1.example.org]

TASK: [Installs apache web server] *********************
changed: [host1.example.org]

PLAY RECAP *********************
host1.example.org              : ok=2    changed=1    unreachable=0    failed=0
```

Note: You might see a cow passing by if you have `cowsay` installed. You can get rid of it with `export ANSIBLE_NOCOWS="1"` if you don't like it.

Let's analyse the output one line at a time.

```
PLAY [web] *********************
```

Ansible tells us it's running the play on hosts `web`. A play is a suite of ansible instructions related to a host. If we'd have another `-host: blah` line in our playbook, it would show up too (but after the first play has completed).

```
GATHERING FACTS *********************
ok: [host1.example.org]
```

Remember when we used the `setup` module? Before each play, ansible runs it on necessary hosts to gather facts. If this is not required because you don't need any info from the host, you can just add `gather_facts: no` below the host entry (same level as `tasks:`).

```
TASK: [Installs apache web server] *********************
changed: [host1.example.org]
```

Next, the real stuff: our (first and only) task is run, and because it says `changed`, we know that it changed something on `host1.example.org`.

```
PLAY RECAP ********************
host1.example.org              : ok=2    changed=1    unreachable=0    failed=0
```

Finally, ansible outputs a recap of what happened: two tasks have been run and one of them changed something on the host (our apache task, setup module doesn't change anything).

Now let's try to run it again and see what happens:

```
$ ansible-playbook -i step-04/hosts -l host1.example.org step-04/apache.yml

PLAY [web] ********************

GATHERING FACTS ********************
ok: [host1.example.org]

TASK: [Installs apache web server] *********************
ok: [host1.example.org]

PLAY RECAP ********************
host1.example.org              : ok=2    changed=0    unreachable=0    failed=0
```

Now changed is '0'. This is absolutely normal and is one of the core feature of ansible: the playbook will act only if there is something to do. It's called *idempotency*, and means that you can run your playbook as many times as you want, you will always end up in the same state (well, unless you do crazy things with the `shell` module of course, but this is beyond ansible's control).

Refining things

Sure our playbook can install apache server, but it could be a bit more complete. It could add a virtualhost, ensure apache is restarted. It could even deploy our web site from a git repository. Lets "make it so"

Head to next step in step-05.

Ansible tutorial

Refining apache setup

We've installed apache, now lets set up our virtualhost.

Refining the playbook

We need just one virtualhost on our server, but we want to replace the default one with something more specific. So we'll have to remove the current (presumably `default`) virtualhost, send our virtualhost, activate it and restart apache.

Let's create a directory called `files`, and add our virtualhost configuration for host1.example.org, which we'll call `awesome-app`:

```
<VirtualHost *:80>
  DocumentRoot /var/www/awesome-app

  Options -Indexes

  ErrorLog /var/log/apache2/error.log
  TransferLog /var/log/apache2/access.log
</VirtualHost>
```

Now, a quick update to our apache playbook and we're set:

```
- hosts: web
  tasks:
    - name: Installs apache web server
      apt: pkg=apache2 state=installed update_cache=true

    - name: Push default virtual host configuration
      copy: src=files/awesome-app dest=/etc/apache2/sites-available/awesome-app mode=0
640

    - name: Disable the default virtualhost
      file: dest=/etc/apache2/sites-enabled/default state=absent
      notify:
        - restart apache

    - name: Disable the default ssl virtualhost
      file: dest=/etc/apache2/sites-enabled/default-ssl state=absent
      notify:
        - restart apache

    - name: Activates our virtualhost
      file: src=/etc/apache2/sites-available/awesome-app dest=/etc/apache2/sites-enabl
ed/awesome-app state=link
      notify:
        - restart apache

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

Here we go:

```
$ ansible-playbook -i step-05/hosts -l host1.example.org step-05/apache.yml

PLAY [web] *********************

GATHERING FACTS *********************
ok: [host1.example.org]

TASK: [Installs apache web server] *********************
ok: [host1.example.org]

TASK: [Push default virtual host configuration] *********************
changed: [host1.example.org]

TASK: [Disable the default virtualhost] *********************
changed: [host1.example.org]

TASK: [Disable the default ssl virtualhost] *********************
changed: [host1.example.org]

TASK: [Activates our virtualhost] *********************
changed: [host1.example.org]

NOTIFIED: [restart apache] *********************
changed: [host1.example.org]

PLAY RECAP *********************
host1.example.org              : ok=7    changed=5    unreachable=0    failed=0
```

Pretty cool! Well, thinking about it, we're getting ahead of ourselves here. Shouldn't we check that the config is ok before restarting apache? This way we won't end up interrupting the service if our configuration file is incorrect.

Lets do that in step-06.

Ansible tutorial

Restarting when config is correct

We've installed apache, pushed our virtualhost and restarted the server. But what if we wanted the playbook to restart the server only if the config is correct? Let's do that.

Bailing out when things go wrong

Ansible has a nifty feature: it will stop all processing if something goes wrong. We'll take advantage of this feature to stop our playbook if the config file is not valid.

Let's change our `awesome-app` virtual host configuration file and break it:

```
<VirtualHost *:80>
  RocumentDoot /var/www/awesome-app

  Options -Indexes

  ErrorLog /var/log/apache2/error.log
  TransferLog /var/log/apache2/access.log
</VirtualHost>
```

As said, when a task fails, processing stops. So we'll ensure that the configuration is valid before restarting the server. We also start by adding our virtualhost *before* removing the default virtualhost, so a subsequent restart (possibly done directly on the server) won't break apache.

Note that we should have done this in the first place. Since we ran our playbook already, the default virtualhost is already deactivated. Nevermind: this playbook might be used on other innocent hosts, so let's protect them.

```
- hosts: web
  tasks:
    - name: Installs apache web server
      apt: pkg=apache2 state=installed update_cache=true

    - name: Push future default virtual host configuration
      copy: src=files/awesome-app dest=/etc/apache2/sites-available/ mode=0640

    - name: Activates our virtualhost
      command: a2ensite awesome-app

    - name: Check that our config is valid
      command: apache2ctl configtest

    - name: Deactivates the default virtualhost
      command: a2dissite default

    - name: Deactivates the default ssl virtualhost
      command: a2dissite default-ssl
      notify:
        - restart apache

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

Here we go:

```
$ ansible-playbook -i step-06/hosts -l host1.example.org step-06/apache.yml

PLAY [web] ********************

GATHERING FACTS ********************
ok: [host1.example.org]

TASK: [Installs apache web server] *********************
ok: [host1.example.org]

TASK: [Push future default virtual host configuration] *********************
changed: [host1.example.org]

TASK: [Activates our virtualhost] *********************
changed: [host1.example.org]

TASK: [Check that our config is valid] *********************
failed: [host1.example.org] => {"changed": true, "cmd": ["apache2ctl", "configtest"],
"delta": "0:00:00.045046", "end": "2013-03-08 16:09:32.002063", "rc": 1, "start": "201
3-03-08 16:09:31.957017"}
stderr: Syntax error on line 2 of /etc/apache2/sites-enabled/awesome-app:
Invalid command 'RocumentDoot', perhaps misspelled or defined by a module not included
in the server configuration
stdout: Action 'configtest' failed.
The Apache error log may have more information.

FATAL: all hosts have already failed -- aborting

PLAY RECAP ********************
host1.example.org              : ok=4    changed=2    unreachable=0    failed=1
```

As you can see since `apache2ctl` returns with an exit code of 1 when it fails, ansible is aware of it and stops processing. Great!

Mmmh, not so great in fact... Our virtual host has been added anyway. Any subsequent apache restart will complain about our config and bail out. So we need a way to catch failures and revert back.

Let's do that in step-07.

Ansible tutorial

Using conditionals

We've installed apache, pushed our virtualhost and restarted the server. But we want to revert things to a stable state if something goes wrong.

Reverting when things go wrong

A word of warning: there's no magic here. The previous error was not ansible's fault. It's not a backup system, and it can't rollback all things. It's your job to make sure your playbooks are safe. Ansible just doesn't know how to revert the effects of `a2ensite awesome-app`.

But if we care to do it, it's well within our reach.

As said, when a task fails, processing stops... unless we accept failure (and we should). This is what we'll do: continue processing if there is a failure but only to revert what we've done.

```
- hosts: web
  tasks:
    - name: Installs apache web server
      apt: pkg=apache2 state=installed update_cache=true

    - name: Push future default virtual host configuration
      copy: src=files/awesome-app dest=/etc/apache2/sites-available/ mode=0640

    - name: Activates our virtualhost
      command: a2ensite awesome-app

    - name: Check that our config is valid
      command: apache2ctl configtest
      register: result
      ignore_errors: True

    - name: Rolling back - Restoring old default virtualhost
      command: a2ensite default
      when: result|failed

    - name: Rolling back - Removing our virtualhost
      command: a2dissite awesome-app
      when: result|failed

    - name: Rolling back - Ending playbook
      fail: msg="Configuration file is not valid. Please check that before re-running
the playbook."
      when: result|failed

    - name: Deactivates the default virtualhost
      command: a2dissite default

    - name: Deactivates the default ssl virtualhost
      command: a2dissite default-ssl
      notify:
        - restart apache

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

The `register` keyword records output from the `apache2ctl configtest` command (exit status, stdout, stderr, ...), and `when: result|failed` checks if the registered variable (`result`) contains a failed status.

Here we go:

```
$ ansible-playbook -i step-07/hosts -l host1.example.org step-07/apache.yml

PLAY [web] ********************

GATHERING FACTS ********************
ok: [host1.example.org]

TASK: [Installs apache web server] *********************
ok: [host1.example.org]

TASK: [Push future default virtual host configuration] *********************
ok: [host1.example.org]

TASK: [Activates our virtualhost] *********************
changed: [host1.example.org]

TASK: [Check that our config is valid] *********************
failed: [host1.example.org] => {"changed": true, "cmd": ["apache2ctl", "configtest"],
"delta": "0:00:00.051874", "end": "2013-03-10 10:50:17.714105", "rc": 1, "start": "201
3-03-10 10:50:17.662231"}
stderr: Syntax error on line 2 of /etc/apache2/sites-enabled/awesome-app:
Invalid command 'RocumentDoot', perhaps misspelled or defined by a module not included
in the server configuration
stdout: Action 'configtest' failed.
The Apache error log may have more information.
...ignoring

TASK: [Rolling back - Restoring old default virtualhost] *********************
changed: [host1.example.org]

TASK: [Rolling back - Removing our virtualhost] *********************
changed: [host1.example.org]

TASK: [Rolling back - Ending playbook] *********************
failed: [host1.example.org] => {"failed": true}
msg: Configuration file is not valid. Please check that before re-running the playbook
.

FATAL: all hosts have already failed -- aborting

PLAY RECAP *********************
host1.example.org              : ok=7    changed=4    unreachable=0    failed=1
```

Seemed to work as expected. Let's try to restart apache to see if it really worked:

```
$ ansible -i step-07/hosts -m service -a 'name=apache2 state=restarted' host1.example.
org
```

```
host1.example.org | success >> {
    "changed": true,
    "name": "apache2",
    "state": "started"
}
```

Ok, now our apache is safe from misconfiguration here.

While this sounds like a lot of work, it isn't. Remember you can use variables almost everywhere, so it's easy to make this a general playbook for apache, and use it everywhere to deploy your virtualhosts. Write it once, use it everywhere. We'll do that in step 9 but for now, let's deploy our web site using git in step-08.

Ansible tutorial

Deploying our website from git

We've installed apache, pushed our virtualhost and restarted the server safely. Now we'll use the git module to deploy our application.

The git module

Well, this is a kind of break. Nothing necessarily new here. The `git` module is just another module. But we'll try it out just for fun. And we'll be familiar with it when it comes to `ansible-pull` later on.

Our virtualhost is set, but we need a few changes to finish our deployment. First, we're deploying a PHP application. So we need to install the `libapache2-mod-php5` package. Second, we have to install `git` since the git module (used to clone our application's git repository) uses it.

We could do it like this:

```
...
- name: Installs apache web server
  apt: pkg=apache2 state=installed update_cache=true

- name: Installs php5 module
  apt: pkg=libapache2-mod-php5 state=installed

- name: Installs git
  apt: pkg=git state=installed
...
```

but Ansible provides a more readable way to write this. Ansible can loop over a series of items, and use each item in an action like this:

```
- hosts: web
  tasks:
    - name: Updates apt cache
      apt: update_cache=true

    - name: Installs necessary packages
      apt: pkg={{ item }} state=latest
```

```
      with_items:
        - apache2
        - libapache2-mod-php5
        - git

    - name: Push future default virtual host configuration
      copy: src=files/awesome-app dest=/etc/apache2/sites-available/ mode=0640

    - name: Activates our virtualhost
      command: a2ensite awesome-app

    - name: Check that our config is valid
      command: apache2ctl configtest
      register: result
      ignore_errors: True

    - name: Rolling back - Restoring old default virtualhost
      command: a2ensite default
      when: result|failed

    - name: Rolling back - Removing out virtualhost
      command: a2dissite awesome-app
      when: result|failed

    - name: Rolling back - Ending playbook
      fail: msg="Configuration file is not valid. Please check that before re-running
the playbook."
      when: result|failed

    - name: Deploy our awesome application
      git: repo=https://github.com/leucos/ansible-tuto-demosite.git dest=/var/www/awes
ome-app
      tags: deploy

    - name: Deactivates the default virtualhost
      command: a2dissite default

    - name: Deactivates the default ssl virtualhost
      command: a2dissite default-ssl
      notify:
        - restart apache

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

Here we go:

```
$ ansible-playbook -i step-08/hosts -l host1.example.org step-08/apache.yml

PLAY [web] ********************

GATHERING FACTS ********************
ok: [host1.example.org]

TASK: [Updates apt cache] *******************
ok: [host1.example.org]

TASK: [Installs necessary packages] *******************
changed: [host1.example.org] => (item=apache2,libapache2-mod-php5,git)

TASK: [Push future default virtual host configuration] *******************
changed: [host1.example.org]

TASK: [Activates our virtualhost] *******************
changed: [host1.example.org]

TASK: [Check that our config is valid] *******************
changed: [host1.example.org]

TASK: [Rolling back - Restoring old default virtualhost] *******************
skipping: [host1.example.org]

TASK: [Rolling back - Removing out virtualhost] *******************
skipping: [host1.example.org]

TASK: [Rolling back - Ending playbook] *******************
skipping: [host1.example.org]

TASK: [Deploy our awesome application] *******************
changed: [host1.example.org]

TASK: [Deactivates the default virtualhost] *******************
changed: [host1.example.org]

TASK: [Deactivates the default ssl virtualhost] *******************
changed: [host1.example.org]

NOTIFIED: [restart apache] *******************
changed: [host1.example.org]

PLAY RECAP ********************
host1.example.org              : ok=10   changed=8    unreachable=0    failed=0
```

You can now browse to http://192.168.33.11, and it should display a kitten, and the server hostname.

Note the `tags: deploy` line allows you to execute just a part of the playbook. Let's say you push a new version for your site. You want to speed up and execute only the part that takes care of deployment. Tags allows you to do it. Of course, "deploy" is just a string, it doesn't have any specific meaning and can be anything. Let's see how to use it:

```
$ ansible-playbook -i step-08/hosts -l host1.example.org step-08/apache.yml -t deploy
X11 forwarding request failed on channel 0

PLAY [web] ********************

GATHERING FACTS ********************
ok: [host1.example.org]

TASK: [Deploy our awesome application] ********************
changed: [host1.example.org]

PLAY RECAP ********************
host1.example.org              : ok=2    changed=1    unreachable=0    failed=0
```

Ok, let's deploy another web server in step-09.

Ansible tutorial

Adding another Webserver

We have one web server. Now we want two.

Updating the inventory

Since we have big expectations, we'll add another web server and a load balancer we'll configure in the next step. But let's complete the inventory now.

```
[web]
host1.example.org ansible_host=192.168.33.11 ansible_user=root
host2.example.org ansible_host=192.168.33.12 ansible_user=root

[haproxy]
host0.example.org ansible_host=192.168.33.10 ansible_user=root
```

Remember we're specifying `ansible_host` here because the host has a different IP than expected (or can't be resolved). You could add these hosts in your `/etc/hosts` and not have to worry, or use real host names (which is what you would do in a classic situation).

Building another web server

We didn't do all this work for nothing. Deploying another web server is dead simple:

```
$ ansible-playbook -i step-09/hosts step-09/apache.yml

PLAY [web] *********************

GATHERING FACTS *********************
ok: [host2.example.org]
ok: [host1.example.org]

TASK: [Updates apt cache] *********************
ok: [host1.example.org]
ok: [host2.example.org]

TASK: [Installs necessary packages] *********************
ok: [host1.example.org] => (item=apache2,libapache2-mod-php5,git)
```

```
changed: [host2.example.org] => (item=apache2,libapache2-mod-php5,git)

TASK: [Push future default virtual host configuration] ********************
ok: [host1.example.org]
changed: [host2.example.org]

TASK: [Activates our virtualhost] *********************
changed: [host2.example.org]
changed: [host1.example.org]

TASK: [Check that our config is valid] *********************
changed: [host2.example.org]
changed: [host1.example.org]

TASK: [Rolling back - Restoring old default virtualhost] *********************
skipping: [host1.example.org]
skipping: [host2.example.org]

TASK: [Rolling back - Removing out virtualhost] *********************
skipping: [host1.example.org]
skipping: [host2.example.org]

TASK: [Rolling back - Ending playbook] *********************
skipping: [host1.example.org]
skipping: [host2.example.org]

TASK: [Deploy our awesome application] *********************
ok: [host1.example.org]
changed: [host2.example.org]

TASK: [Deactivates the default virtualhost] *********************
changed: [host1.example.org]
changed: [host2.example.org]

TASK: [Deactivates the default ssl virtualhost] *********************
changed: [host2.example.org]
changed: [host1.example.org]

NOTIFIED: [restart apache] *********************
changed: [host1.example.org]
changed: [host2.example.org]

PLAY RECAP *********************
host1.example.org              : ok=10   changed=5    unreachable=0    failed=0
host2.example.org              : ok=10   changed=8    unreachable=0    failed=0
```

All we had to do was remove `-l host1.example.org` from our command line. Remember `-l` is a switch that limits the playbook run on specific hosts. Now that we don't limit anymore, it will run on all hosts where the playbook is intended to run on (i.e. `web`).

If we had other servers in group `web` but wanted to limit the playbook to a subset, we could have used, for instance: `-l firsthost:secondhost:...`.

Now that we have this nice farm of web servers, let's turn it into a cluster by putting a load balancer in front of them in step-10.

Ansible tutorial

Templates

We'll use the `haproxy` as loadbalancer. Of course, install is just like we did for apache. But now configuration is a bit more tricky since we need to list all web servers in haproxy's configuration. How can we do that?

HAProxy configuration template

Ansible uses Jinja2, a templating engine for Python. When you write Jinja2 templates, you can use any variable defined by Ansible.

For instance, if you want to output the inventory_name of the host the template is currently built for, you just can write `{{ inventory_hostname }}` in the Jinja template.

Or if you need the IP of the first ethernet interface (which ansible knows thanks to the `setup` module), you just write: `{{ ansible_eth1['ipv4']['address'] }}` in your template.

Jinja2 templates also support conditionals, for-loops, etc...

Let's make a `templates/` directory and create a Jinja template inside. We'll call it `haproxy.cfg.j2`. We use the `.j2` extension by convention, to make it obvious that this is a Jinja2 template, but this is not necessary.

```
global
    daemon
    maxconn 256

defaults
    mode http
    timeout connect 5000ms
    timeout client 50000ms
    timeout server 50000ms

listen cluster
    bind {{ ansible_eth1['ipv4']['address'] }}:80
    mode http
    stats enable
    balance roundrobin
{% for backend in groups['web'] %}
    server {{ hostvars[backend]['ansible_hostname'] }} {{ hostvars[backend]['ansible_e
th1']['ipv4']['address'] }} check port 80
{% endfor %}
    option httpchk HEAD /index.php HTTP/1.0
```

We have many new things going on here.

First, `{{ ansible_eth1['ipv4']['address'] }}` will be replaced by the IP of the load balancer on eth1.

Then, we have a loop. This loop is used to build the backend servers list. It will loop over every host listed in the `[web]` group (and put this host in the `backend` variable). For each of the hosts it will render a line using host's facts. All hosts' facts are exposed in the `hostvars` variable, so it's easy to access another host variables (like its hostname or in this case IP).

We could have written the host list by hand, since we have only 2 of them. But we're hoping that the server will be very successful, and that we'll need a hundred of them. Thus, adding servers to the configuration or swapping some out boils down to adding or removing hosts from the `[web]` group.

HAProxy playbook

We've done the most difficult part of the job. Writing a playbook to install and configure HAproxy is a breeze:

```
- hosts: haproxy
  tasks:
    - name: Installs haproxy load balancer
      apt: pkg=haproxy state=installed update_cache=yes

    - name: Pushes configuration
      template: src=templates/haproxy.cfg.j2 dest=/etc/haproxy/haproxy.cfg mode=0640 o
wner=root group=root
      notify:
        - restart haproxy

    - name: Sets default starting flag to 1
      lineinfile: dest=/etc/default/haproxy regexp="^ENABLED" line="ENABLED=1"
      notify:
        - restart haproxy

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

Looks familiar, isn't it? The only new module here is `template`, which has the same arguments as `copy`. We also restrict this playbook to the group `haproxy`.

And now... let's try this out. Since our inventory contains only hosts necessary for the cluster, we don't need to limit the host list and can even run both playbooks. Well, to tell the truth, we must run both of them at the same time, since the haproxy playbook requires facts *from* the two webservers. In step-11 we'll show how to avoid this.

```
$ ansible-playbook -i step-10/hosts step-10/apache.yml step-10/haproxy.yml

PLAY [web] ********************

GATHERING FACTS ********************
ok: [host1.example.org]
ok: [host2.example.org]

TASK: [Updates apt cache] *********************
ok: [host1.example.org]
ok: [host2.example.org]

TASK: [Installs necessary packages] *********************
ok: [host1.example.org] => (item=apache2,libapache2-mod-php5,git)
ok: [host2.example.org] => (item=apache2,libapache2-mod-php5,git)

TASK: [Push future default virtual host configuration] *********************
ok: [host2.example.org]
ok: [host1.example.org]

TASK: [Activates our virtualhost] *********************
changed: [host1.example.org]
```

```
changed: [host2.example.org]

TASK: [Check that our config is valid] ********************
changed: [host1.example.org]
changed: [host2.example.org]

TASK: [Rolling back - Restoring old default virtualhost] *********************
skipping: [host1.example.org]
skipping: [host2.example.org]

TASK: [Rolling back - Removing out virtualhost] *********************
skipping: [host1.example.org]
skipping: [host2.example.org]

TASK: [Rolling back - Ending playbook] *********************
skipping: [host1.example.org]
skipping: [host2.example.org]

TASK: [Deploy our awesome application] *********************
ok: [host2.example.org]
ok: [host1.example.org]

TASK: [Deactivates the default virtualhost] *********************
changed: [host1.example.org]
changed: [host2.example.org]

TASK: [Deactivates the default ssl virtualhost] *********************
changed: [host2.example.org]
changed: [host1.example.org]

NOTIFIED: [restart apache] *********************
changed: [host2.example.org]
changed: [host1.example.org]

PLAY RECAP *********************
host1.example.org              : ok=10   changed=5    unreachable=0    failed=0
host2.example.org              : ok=10   changed=5    unreachable=0    failed=0



PLAY [haproxy] *********************

GATHERING FACTS *********************
ok: [host0.example.org]

TASK: [Installs haproxy load balancer] *********************
changed: [host0.example.org]

TASK: [Pushes configuration] *********************
changed: [host0.example.org]

TASK: [Sets default starting flag to 1] *********************
changed: [host0.example.org]
```

```
NOTIFIED: [restart haproxy] *********************
changed: [host0.example.org]

PLAY RECAP ********************
host0.example.org              : ok=5    changed=4    unreachable=0    failed=0
```

Looks good. Now head to http://192.168.33.10/ and see the result. Your cluster is deployed!

you can even peek at HAProxy's statistics at http://192.168.33.10/haproxy?stats.

Now on to the next chapter about "Variables again", in step-11.

Ansible tutorial

Variables again

So we've setup our loadbalancer, and it works quite well. We grabbed variables from facts and used them to build the configuration. But Ansible also supports other kinds of variables. We already saw `ansible_host` in inventory, but now we'll use variables defined in `host_vars` and `group_vars` files.

Fine tuning our HAProxy configuration

HAProxy usually checks if the backends are alive. When a backend seems dead, it is removed from the backend pool and HAproxy doesn't send requests to it anymore.

Backends can also have different weights (between 0 and 256). The higher the weight, the higher number of connections the backend will receive compared to other backends. It's useful to spread traffic more appropriately if nodes are not equally powerful.

We'll use variables to configure all these parameters.

Group vars

The check interval will be set in a group_vars file for haproxy. This will ensure all haproxies will inherit from it.

We just need to create the file `group_vars/haproxy` below the inventory directory. The file has to be named after the group you want to define the variables for. If we wanted to define variables for the web group, the file would be named `group_vars/web`.

```
haproxy_check_interval: 3000
haproxy_stats_socket: /tmp/sock
```

The name is arbitrary. Meaningful names are recommended of course, but there is no required syntax. You could even use complex variables (a.k.a. Python dict) like this:

```
haproxy:
    check_interval: 3000
    stats_socket: /tmp/sock
```

This is just a matter of taste. Complex vars can help group stuff logically. They can also, under some circumstances, merge subsequently defined keys (note however that this is not the default ansible behaviour). For now we'll just use simple variables.

Hosts vars

Hosts vars follow exactly the same rules, but live in files under `host_vars` directory.

Let's define weights for our backends in `host_vars/host1.example.com` :

```
haproxy_backend_weight: 100
```

and `host_vars/host2.example.com` :

```
haproxy_backend_weight: 150
```

If we'd define `haproxy_backend_weight` in `group_vars/web` , it would be used as a 'default': variables defined in `host_vars` files overrides varibles defined in `group_vars` .

Updating the template

The template must be updated to use these variables.

```
global
    daemon
    maxconn 256
{% if haproxy_stats_socket %}
    stats socket {{ haproxy_stats_socket }}
{% endif %}

defaults
    mode http
    timeout connect 5000ms
    timeout client 50000ms
    timeout server 50000ms

listen cluster
    bind {{ ansible_eth1['ipv4']['address'] }}:80
    mode http
    stats enable
    balance roundrobin
{% for backend in groups['web'] %}
    server {{ hostvars[backend]['ansible_hostname'] }} {{ hostvars[backend]['ansible_e
th1']['ipv4']['address'] }} check inter {{ haproxy_check_interval }} weight {{ hostvar
s[backend]['haproxy_backend_weight'] }} port 80
{% endfor %}
    option httpchk HEAD /index.php HTTP/1.0
```

Note that we also introduced an `{% if ...` block. This block enclosed will only be rendered if the test is true. So if we define `haproxy_stats_socket` somewhere for our loadbalancer (we might even use the `--extra-vars="haproxy_stats_sockets=/tmp/sock"` at the command line), the enclosed line will appear in the generated configuration file (note that the suggested setup is highly insecure!).

Let's go:

```
ansible-playbook -i step-11/hosts step-11/haproxy.yml
```

Note that, while we could, it's not necessary to run the apache playbook since nothing changed, but we had to cheat a bit for that. Here is the updated haproxy playbook :

```
- hosts: web
- hosts: haproxy
  tasks:
    - name: Installs haproxy load balancer
      apt: pkg=haproxy state=installed update_cache=yes

    - name: Pushes configuration
      template: src=templates/haproxy.cfg.j2 dest=/etc/haproxy/haproxy.cfg mode=0640 o
wner=root group=root
      notify:
        - restart haproxy

        - name: Sets default starting flag to 1
          lineinfile: dest=/etc/default/haproxy regexp="^ENABLED" line="ENABLED=1"
          notify:
            - restart haproxy

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

See? We added an empty play for web hosts at the top. It does nothing. But it's here because it will trigger facts gathering on hosts in group `web`. This is required because the haproxy playbook needs to pick facts from hosts in this group. If we don't do this, ansible will complain saying that `ansible_eth1` key doesn't exist.

Now on to the next chapter about "Migrating to Roles!", in step-12.

Ansible tutorial

Migrating to roles!

Now that our playbook is done, let's refactor everything! We'll replace our plays with roles. Roles are just a new way of organizing files but bring interesting features. I won't go into great lengths here, since they're listed in Ansible's documentation, but my favorite is probably roles dependencies: role B can depend on another role A. Thus, when applying role B, role A will automatically be applied too. We'll see this in the next chapter, but for now, let's refactor our playbook to use roles.

Roles structures

Roles add a bit of "magic" to Ansible: they assume a specific file organization. While there is a suggested layout regarding roles, you can organize things the way you want using includes. However, role's conventions help building modular playbooks, and housekeeping will be much simpler. Rubyists would call this "convention over configuration".

The file layout for roles looks like this:

```
roles
  |
  |_some_role
       |
       |_defaults
       |   |
       |   |_main.yml
       |   |_...
       |
       |_files
       |   |
       |   |_file1
       |   |_...
       |
       |_handlers
       |   |
       |   |_main.yml
       |   |_some_other_file.yml
       |   |_ ...
       |
       |_meta
       |   |
       |   |_main.yml
       |   |_some_other_file.yml
       |   |_ ...
       |
       |_tasks
       |   |
       |   |_main.yml
       |   |_some_other_file.yml
       |   |_ ...
       |
       |_templates
       |   |
       |   |_template1.j2
       |   |_...
       |
       |_vars
           |
           |_main.yml
           |_some_other_file.yml
           |_ ...
```

Quite simple. The files named `main.yml` are not mandatory. However, when they exist, roles will add them to the play automatically. You can use this file to include other tasks, handlers, ... in the play. We'll see that in a minute.

Note that there is also a `vars` and a `meta` directory. `vars` is used when you want to put a bunch of variables regarding the roles. However, I don't like setting vars in roles (or plays) directly. I think variables belong to configuration, while plays are the structure. In other

words, I see plays and roles as a factory, and data as inputs to this factory. So I really prefer to have "data" (e.g. variables) outside roles and play. This way, I can share my roles more easily, without worrying about exposing too much about my servers. But that's just a personal preference. Ansible just lets you do it the way you want.

But you have some vars that you hardly want to change. For instance, if you have a role for nginx that pulls the .deb package from a PPA, you might want to add the PPA address in `vars/main.yml`. It is something that you *can* configure, but that will be mostly static 99% of the time. Using `vars` will let you pull out this information out of your role, making it more generic. But really, this is a matter of taste.

However, for real vars (e.g. things you would like to use in a configuration file generated by a template), you can set defaults for roles, and this is a recommended practice. Using sane defaults ensures your role always work. For instance, you could set the number of pre-forked servers for your apache server. The best place to put the defaults is... you guessed it, the `defaults` directory.

The `meta` directory is where you can add dependencies, and it's really a neat feature. We'll see that later.

Note that roles sit in the `roles` directory, which is also cool since it will reduce top level ansible playbook clutter. But you can configure Ansible to use an alternate directory to store role (see `roles_path` variable in `ansible.cfg`). This way you can setup a 'central place ' for all your roles, and use them in all your playbooks.

Creating the Apache role

Ok, now that we know the required layout, we can create our apache role from our apache playbook.

The steps required are really simple:

- create the roles directory and apache role layout
- extract the apache handler into `roles/apache/handlers/main.yml`
- move the apache configuration file `awesome-app` into `roles/apache/files/`
- create a role playbook

Creating the role layout

This is what has been done to convert step-11 apache files into a role:

```
mkdir -p step-12/roles/apache/{tasks,handlers,files}
```

Now we need to copy the tasks from `apache.yml` to `main.yml`, so this file looks like this:

```
- name: Updates apt cache
  apt: update_cache=true

- name: Installs necessary packages
  apt: pkg={{ item }} state=latest
  with_items:
    - apache2
    - libapache2-mod-php5
    - git

...

- name: Deactivates the default ssl virtualhost
  command: a2dissite default-ssl
  notify:
    - restart apache
```

The file is not fully reproduced, but it is exactly the content of `apache.yml` between `tasks:` and `handlers:`.

Note that we also have to remove references to `files/` and `templates/` directories in tasks. Since we're using the roles structure, Ansible will look for them in the right directories.

Extracting the handler

We can extract the handlers part and create `step-12/roles/apache/handlers/main.yml`:

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

Moving the configuration file

As simple as:

```
cp step-11/files/awesome-app step-12/roles/apache/files/
```

At this point, the apache role is fully working, but we need a way to invoke it.

Create a role playbook

Let's create a top level playbook that we'll use to map hosts and host groups to roles. We'll call it `site.yml`, since our goal is to have our site-wide configuration in it. While we're at it, we'll include `haproxy` in it too:

```
- hosts: web
  roles:
    - { role: apache }

- hosts: haproxy
  roles:
    - { role: haproxy }
```

That wasn't too hard.

Now let's create the haproxy role:

```
mkdir -p step-12/roles/haproxy/{tasks,handlers,templates}
cp step-11/templates/haproxy.cfg.j2 step-12/roles/haproxy/templates/
```

then extract the handler, and remove reference to `templates/`.

We can try out our new playbook with:

```
ansible-playbook -i step-12/hosts step-12/site.yml
```

If eveything goes well, we should end up with a happy "PLAY RECAP" like this one:

```
host0.example.org          : ok=5    changed=2    unreachable=0 failed=0
host1.example.org          : ok=10   changed=5    unreachable=0 failed=0
host2.example.org          : ok=10   changed=5    unreachable=0 failed=0
```

You may have noticed that running all roles in site.yml can take a long time. What if you only wanted to push changes to web? This is also easy, with the limit flag:

```
ansible-playbook -i step-12/hosts -l web step-12/site.yml
```

This concludes our migration to roles. It was quite easy, and adds a bunch of features to our playbook that we'll use in a future step.

In step-13, we will see how we can use tags to select which parts of our playbook we want to run.

The end

At this point, you can try building up everything from scratch, to see if you can properly provision your cluster with your playbook.

Fire in the hole!

```
vagrant destroy -f
vagrant up
ansible-playbook -c paramiko -i step-00/hosts step-00/setup.yml --ask-pass --sudo
```

(you might need to wait a little for the network to come up before running the last command).

All the preceeding commands are just here to set-up our test environment. Deploying on the blank machines just requires one line:

```
ansible-playbook -i step-99/hosts step-99/site.yml
```

Just one command to rule them all: you have your cluster, can add nodes ad nauseam, tune settings, ... all this can be extended at will with more variables, other plays, etc...

The end

Ok, seems we're done with our tutorial. Hope you enjoyed playing with Ansible, and felt the power of this new tool.

Now go straight to Ansible website, dive in the docs, check references, skim through playbooks, chat on freenode in #ansible, and foremost, have fun!