Ansible for Automation!

With <3 from SymfonyCasts

Chapter 1: Hello Ansible!

Welcome automation lovers!!! You've come to the right place, because this tutorial is all about Ansible! And boy, it's just a *lot* of fun. Imagine if we could take control of an army of tiny, but mighty robots... simply by writing a YAML file. Think of the servers we could launch! The infrastructure we could build! The code we could deploy! The laundry we could wash! Well, actually, these Ansible robots are sorta virtual robots, so yes to automating things like server setup... but probably not to automating your laundry. However, I *do* hope one of you proves me wrong!

To be the best robot overlord you can be, download the course code from this page and code along with me! After unzipping the file, you'll find a start/ directory, which will have the same code that you see here. Follow the README.md file to get your project up and running. Well actually, setting up the project isn't *that* important, because we will ultimately use Ansible to do that! But if you want to see things working now follow that file. The last step will be to go into your terminal, move into the directory, and run

```
$ php bin/console server:run
```

to launch the built-in PHP web server. In your browser, open up http://localhost:8000. Introducing, our new side project here at KnpUniversity: MooTube! Yes, far too many cows are out of shape and need some serious exercise! So in case this whole tutorial business falls through, we'll turn to bovine fitness: a subscription-based service to keep cows in tip-top shape.

The App & Our Mission!

The first version of our app is done actually! But there are two problems. First, we'd love to have a way to easily boot up development servers complete with PHP, a web server and anything else we need to get our app working. And second, it would be *bananas* if we could have an automated way to *deploy* code to Amazon EC2 servers. Well, that's *exactly* what we're going to do.

Our project is a Symfony application... but if you're not used to Symfony, that's *no* problem. From a server-perspective, the app requires a few interesting things:

111 lines | src/AppBundle/Controller/DefaultController.php

```php
... lines 1 - 9
class DefaultController extends Controller
{
... lines 12 - 14
    public function indexAction()
    {
        $videos = $this->getVideoRepository()
            ->findAll();
        $tags = $this->getUniqueOrderedTags($videos);

        // Redis cache
        try {
            if ($this->getRedisClient()->exists('total_video_uploads_count')) {
                $totalVideoUploadsCount = $this->getRedisClient()->get('total_video_uploads_count');
            } else {
                $totalVideoUploadsCount = $this->countTotalVideoUploads();
                $this->getRedisClient()->set('total_video_uploads_count', $totalVideoUploadsCount, 'ex', 60); // 60s
            }
... lines 29 - 35
        } catch (ConnectionException $e) {
            $totalVideoUploadsCount = $this->countTotalVideoUploads();
            $totalVideoViewsCount = $this->countTotalVideoViews();
        }

        return $this->render('default/index.html.twig', [
            'videos' => $videos,
            'tags' => $tags,
            'totalVideoUploadsCount' => $totalVideoUploadsCount,
            'totalVideoViewsCount' => $totalVideoViewsCount,
        ]);
    }
... lines 48 - 109
}
```

First, it needs a database connection to load the video info. And second, it uses Redis to cache a few things. So when we boot up our servers, that stuff needs to be there.

Introducing Ansible

There's a good chance you've setup a server before. I know I've setup a *ton*, and it's always the same, manual, error-prone, confusing process: SSH into the server, manually run a bunch of commands, have errors, Google things, run more commands, and edit a bunch of files.

Ansible... kinda does the same thing. When you execute something with Ansible, it ultimately SSH's onto the server and runs some commands. Ansible is said to be "agentless", which just means that you don't need to install anything on the target server. As long as you can SSH to a server, you can unleash your Ansible robot army onto it. That's pretty cool.

But Ansible is even more interesting than that. When you execute an Ansible *task* - that's what they're called - it is *idempotent*... well, usually it is. Idempotency is an obscure - but cool - word to mean that Ansible tasks don't just dumbly run a command. What they *really* do is *guarantee* that the server finishes in a specific *state*. For example, if we tell Ansible to create a directory, it doesn't necessarily mean that it will run a mkdir command. Instead, it means that Ansible will just make sure that the directory exists - only creating it if necessary.

This idea of guranteeing a "state" - like "this directory must exist" - is much more powerful than randomly running commands over SSH. These tasks also send back JSON info about what happened, which we can use to tweak and influence what happens next.

Installing Ansible

Ok, let's start playing already! First, we need to install Ansible of course! Since I'm on a Mac, I've already installed it with: brew install robotarmy. I mean,

```
$ brew install ansible
```

If you're on different system, check out the Ansible docs for your install instructions. Unfortunately, if you're using Windows, you *can't* use Ansible. Well, you can't natively. If you're virtualizing a Linux machine or are using Windows 10 with the Linux subsystem, then you can install Ansible there.

Once you've got it, run

```
$ ansible --version
```

to make sure you have at least version 2.0.

Ok team, let's boot up our robot army!

Chapter 2: Modules

We need to talk about 2 important words in Ansible: modules and hosts.

Ansible comes built with a *ton* of things called *modules*: small programs that do some work on the server. Most of the time, instead of saying:

> Ansible! Execute this command!

you'll say:

> Ansible! Execute this module and allow it to run whatever commands it needs to get its job done.

For example, if you want to install something on an Ubuntu server, instead of running sudo apt-get install php7.1, you'll use the apt module. Need to edit a file? There's a module for that too.

And when you execute a module, you'll of course always execute that module on one or more servers. These are called hosts. So, in Ansible language, we'll say:

> I want to run this module on these hosts.

Running your First Module

Ok, terminology garbage done. Let's do something! The simplest way to execute a module is from the command line. ansible localhost means that - for now - we're going to run this module against our local machine. Then, -m command to run the "command" module - the simplest module in Ansible that allows you to... well... just run a command directly on the server. Then, add -a "/bin/echo 'Hello Ansible'" to pass that as an argument to the command module:

```
$ ansible localhost -m command -a "/bin/echo 'Hello Ansible'"
```

Try it! We see some output and... Hello Ansible! Congrats! You just ran your first module: command. In this case, we can even remove the -m option:

```
$ ansible localhost -a "/bin/echo 'Hello Ansible'"
```

The command module is so fundamental, it's the *default* module... if we don't pass one.

Hey! Let's try another module!

```
$ ansible localhost -m ping
```

You can probably guess what this does: ping is a small program that makes sure that we can contact the server.

What other modules can we use? Flip back to your browser and Google for "Ansible modules". They have a few pages, like "modules by category". But let's go to the full All Modules page.

Woh! If we're commanding a robot army, we just found out that a lot of our robots already know how to do a *ton* of stuff. Yes! This is all free functionality!

The composer Module

One of these modules should look pretty interesting to PHP developers: the composer module. Instead of executing Composer commands manually on the server, you can use this.

For example, back on the command line, if you setup your project like I did, then your vendor/ directory is populated with files.

Let's kill them! Be reckless by running:

```
$ rm -rf vendor/
```

Now, take the composer module for a test drive:

```
$ ansible localhost -m composer
```

Woh! It fails!

> Missing required arguments: working_dir.

The Ansible module documentation is pretty awesome: each page lists all of that module's options, their default value and whether or not they're *required*, like working_dir. And below, they usually have a bunch of really nice examples.

In this case, to pass the option, add -a "working_dir=./" to point to *this* directory, since the module will run on *this* machine. We also need to pass no_dev=false. That's just another option for this module - and we'll talk about what it does in a little while.

Ok, try that!

```
$ ansible localhost -m composer -a "working_dir=./ no_dev=false"
```

Woohoo! It looks like it's working! My terminal tab even shows the crazy things it's doing behind the scenes.

Tip

If it doesn't work on your machine, no big deal. This module requires you to have PHP and Composer installed. Soon, we'll *guarantee* that these are installed on a fresh Ubuntu machine.

Once it's done... boom! We see the full output of everything that happened.

The "changed" Status

AND, we see one *really* important thing: it says "changed true" and the output is yellow. The module detected that this command made a *change* to the server. Run the module again:

```
$ ansible localhost -m composer -a "working_dir=./ no_dev=false"
```

Woh! Now the output is green and it says "changed: false".

That is one of the most important superpowers of modules: not only do they make something happen on your server, they're able to detect whether or not the server *changed* by running the module. This will be important: later, we can trigger different actions based on whether or not a module did or did not actually make any changes.

But how the heck did Ansible know that the server didn't change the second time? That cleverness is actually built into each module. The composer module is smart enough to know that nothing changed based on the *output* of the command - the fact that it said:

> Nothing to install or update

By the way, now that we're rocking Ansible, we could have cleared the vendor directory via the command module:

```
$ ansible localhost -a "rm -rf vendor/"
```

To prove that worked, run the composer module again:

```
$ ansible localhost -m composer -a "working_dir=./ no_dev=false"
```

Ha! Back to "changed: true" with yellow output.

Next, let's talk about how we organize which servers we're running Ansible against. Because obviously, we don't want to run against localhost forever!

Chapter 3: Hosts & the Inventory File

When we run ansible, we see a few warnings on top:

> Host file not found

with a path to a host file somewhere on your system. Then it says:

> provided hosts list is empty, only localhost is available.

It turns out, at first, we can *only* execute ansible against one host: localhost. If you want to start running against any other server, you need to create a host configuration file. You can either do this in a global hosts file - in the location described in the warning - or you can create a file right inside your project. That's the way I like to do it!

In your project, create a new directory called ansible. And inside, make a new hosts.ini file.

The *smallest* thing you need to configure a host is... just the IP address: 127.0.0.1:

```
2 lines | ansible/hosts.ini
127.0.0.1
```

We'll keep running things against our local machine for a bit longer.

As *soon* as you have this, you can use *this* as your host: ansible 127.0.0.1 -m ping. To tell Ansible about the new hosts file, add -i ansible/hosts.ini. It's -i because the hosts file is known as your *inventory*. Try it!

```
$ ansible 127.0.0.1 -m ping -i ansible/hosts.ini
```

Setting Host Variables

Ah! It fails! I keep telling you that Ansible works by connecting over SSH and then running commands. Well, technically, that's not 100% true: you can actually configure Ansible to connect to your server in different ways, though you'll almost always use SSH. The most common exception is when you're working on your *local* machine - you don't need to connect via SSH at all!

To tell Ansible that this is a local connection, in your hosts.ini file, after the IP address, add ansible_connection=local:

```
3 lines | ansible/hosts.ini
127.0.0.1 ansible_connection=local
... lines 2 - 3
```

There's also a docker connection type if you're getting nerdy with Docker.

Try that ping again!

```
$ ansible 127.0.0.1 -m ping -i ansible/hosts.ini
```

Got it!

This little change is actually *really* important. By saying ansible_connection=local, we are setting a *variable* inside of Ansible. And as we build out more complex Ansible configuration, this idea of setting and using variables will become more important. As you'll see, you can set more variables for each host, which will let us change behavior on a host-by-host basis.

In this case, ansible_connection is a built-in variable that Ansible uses when it connects. We're simply changing it first.

Host Groups

So right now, we have just *one* host. But eventually, you might have *many* - like 5 web server hosts, 2 database hosts and a Redis host. One common practice is to *group* your hosts. Let me show you: at the top, add a group called [local], with our one host below it:

4 lines | ansible/hosts.ini

```
[local]
127.0.0.1 ansible_connection=local
... lines 3 - 4
```

As soon as we do that, instead of using the IP address in the command, we can use the group name:

```
$ ansible local -m ping -i ansible/hosts.ini
```

That will run the module against *all* hosts inside of the local group... which is just one right now. Boring! Let's add another! Below the first, add localhost with ansible_connection=local:

5 lines | ansible/hosts.ini

```
[local]
127.0.0.1 ansible_connection=local
localhost ansible_connection=local
... lines 4 - 5
```

This is a little silly, but it shows how this works. Run the command now!

```
$ ansible local -m ping -i ansible/hosts.ini
```

Yes! It runs the ping module twice: once on each server. If you needed to setup 10 web servers... well, you can imagine how *awesome* this could be.

And actually, there's a special option - --list-hosts that can show you all of the hosts in that group:

```
$ ansible local --list-hosts -i ansible/hosts.ini
```

Ok, remove the localhost line:

4 lines | ansible/hosts.ini

```
[local]
127.0.0.1 ansible_connection=local
... lines 3 - 4
```

Time to start executing things against a *real* server.

Chapter 4: Vagrant <3's Ansible

Our first big goal is to see if we can use Ansible to setup, or provision, an entire server that can run our MooTube app. To setup an empty VM, we'll use Vagrant with VirtualBox.

So before we start, make sure that you have VirtualBox installed: it's different for each OS, but should - hopefully - be pretty easy. Next, make sure you have Vagrant: it has a nice installer for every OS. On my Mac, surprise! I installed Vagrant via brew.

As long as you can type:

```
$ vagrant -v
```

you're good to go.

Vagrantfile Setup!

If you're new to Vagrant, it's a tool that helps create and boot different virtual machines, usually via VirtualBox behind the scenes. It works by reading a configuration file... which we don't have yet. Let's generate it!

```
$ vagrant init ubuntu/trusty64
```

That just created a file - Vagrantfile - at the root of our project that will boot a VM using an ubuntu/trusty64 image. That's not the newest version of Ubuntu, but it works really well. You're free to use a different one, but a few commands or usernames and passwords might be different!

Tip

If you want to use the latest Ubuntu 18.04 LTS release, you'll need a few tweaks:

1) Change the VM box in Vagrantfile to:

```
# Vagrantfile
Vagrant.configure("2") do |config|
  # ...
  config.vm.box = "ubuntu/bionic64"
  # ...
```

Or ubuntu/xenial64 in case you're interested in Ubuntu 16.04 LTS release.

2) Ubuntu 18.04/16.04 requires a private SSH key to be specified instead of a simple password to login via SSH - Ansible can't log in into the server using just a username and password pair. Also, the Ansible user should be set to vagrant. You can specify all this information in the hosts.ini file for the VirtualBox host:

```
# ...
[vb]
192.168.33.10 ansible_user=vagrant ansible_ssh_private_key_file=./.vagrant/machines/default/virtualbox/private_key
# ...
```

Make sure to uncomment private_network configuration as we did below in this code block to be able to connect to the 192.168.33.10 IP.

3) Notice, that Ubuntu 18.04/16.04 has the new pre-installed Python 3. In case you have an error related to Python interpreter, specify the path to its binary explicitly as:

```
# ...
[vb]
192.168.33.10 ansible_user=vagrant ansible_ssh_private_key_file=./.vagrant/machines/default/virtualbox/private_key
ansible_python_interpreter=/usr/bin/python3
# ...
```

4) Ubuntu 18.04/16.04 doesn't come with aptitude pre-installed, so you will need to install it first if you want to use the safe upgrade option for installed packages - we will talk about it later in this course. Just add one new task to your playbook before upgrading:

```
# ansible/playbook.yml
---
- hosts: vb
  # ...
  tasks:
    # ...
    - name: Install aptitude
      become: true
      apt:
        name: aptitude

    - name: Upgrade installed packages
      become: true
      apt:
        upgrade: safe
    # ...
```

Boot that VM!

With that file in place, let's boot the VM!

```
$ vagrant up
```

Then... go make a sandwich! Or run around outside! Unless you've run this command before, it'll need to download the Ubuntu image... which is pretty huge. So go freshen up your cup of coffee and come back.

Thanks to the power of video, we'll zoom to the end! Zoooom!

When it finishes, make sure you can SSH into it:

```
$ vagrant ssh
```

With any luck, you'll step right into your brand new, basically empty, but totally awesome, Ubuntu virtual machine.

By the way, Vagrant stores some info in a .vagrant directory. In a real project, you'll probably want to add this to your .gitignore file:

19 lines | .gitignore

```
... lines 1 - 17
18  /.vagrant/
```

Setup an External IP Address

Our goal is to have Ansible talk to this new VM. For that, we need a dependable IP address for the VM. Check out the Vagrantfile that was generated automatically for us: it has a section about a "private network". Uncomment that!

```
72 lines | Vagrantfile
... lines 1 - 7
Vagrant.configure("2") do |config|
... lines 9 - 26
  # Create a private network, which allows host-only access to the machine
  # using a specific IP.
  config.vm.network "private_network", ip: "192.168.33.10"
... lines 30 - 72
```

This will let us talk to the VM via 192.168.33.10.

For that to take effect, run:

```
$ vagrant reload
```

Tip

If you're inside the VM, for stepping out of it simply run:

```
$ exit
```

That should take just a minute or two. Perfect! And now we can ping that IP!

```
$ ping 192.168.33.10
```

Configuring the new Ansible Host

The VM represents a new host. And that means we need to add it to our hosts file! In hosts.ini, let's keep the local group and add another called vb, for VirtualBox. Under there, add the IP: 192.168.33.10:

```
7 lines | ansible/hosts.ini
... lines 1 - 3
[vb]
192.168.33.10
... lines 6 - 7
```

We know that as soon as we make this change, we should be able to use the vb host:

```
$ ansible vb -m ping -i ansible/hosts.ini
```

That should work, right? It fails! The ping module does a bit more than just a ping, and in this case, it's detecting that Ansible can't SSH into the machine. The reason is that we haven't specified a username and password or key to use for SSH.

Configuring SSH Properly

To see what I mean, try SSH'ing manually - the machine is setup with a vagrant user:

```
$ ssh vagrant@192.168.33.10
```

Woh! Our first error looks awesome!

> WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!

You may or may not get this error. Since I've used Vagrant in this same way in the past, it's telling me that last time I SSH'ed to this IP address, it was a different machine! We know that's ok - there's nothing nefarious happening. To fix it, I just need to find line 210 of my known_hosts file and remove the old fingerprint. I'll do that and save. Try it again:

```
$ ssh vagrant@192.168.33.10
```

It saves the fingerprint and then asks for a password. The password for the image that we're using is vagrant. That's pretty standard, but it might be different if you're using a different image.

Tip

If you still see an error like:

> vagrant@192.168.33.10: Permission denied (publickey).

It seems like the server requires a private SSH key file to be specified. Try to specify it as an identity file:

```
$ ssh vagrant@192.168.33.10 -i ./.vagrant/machines/default/virtualbox/private_key
```

We're inside! So, how can we tell Ansible to SSH with username vagrant and password vagrant? The answer is... not surprising! These are two more variables in your hosts inventory file: ansible_user set to vagrant and ansible_ssh_pass=vagrant:

7 lines | ansible/hosts.ini

```
... lines 1 - 3
[vb]
192.168.33.10 ansible_user=vagrant ansible_ssh_pass=vagrant
... lines 6 - 7
```

Try the ping again:

```
$ ansible vb -m ping -i ansible/hosts.ini
```

Tip

If you still can't SSH into the Vagrant box with Ansible using a simple username/password pair and continue getting an error like:

```
192.168.33.10 | FAILED! => {
    "msg": "to use the 'ssh' connection type with passwords, you must install the sshpass program"
}
```

Try to specify the SSH private key instead of password. For this, change the line to:

```
# ...
[vb]
192.168.33.10 ansible_user=vagrant ansible_ssh_private_key_file=./.vagrant/machines/default/virtualbox/private_key
# ...
```

Eureka! But, quick note about the SSH password. If this weren't just a local VM, we might not want to store the password in plain text. Instead, you can use a private key for authentication, or use the Ansible "vault" - a cool feature that lets us *encrypt* secret things, like passwords. More on that later.

But for now, our setup is done! We have a VM, and Ansible can talk to it. Next, we need to create a *playbook* that's capable of setting up the VM.

Chapter 5: Create your Playbook!

At this point, we can execute any module on our hosts... but we're still doing it manually from the command line. If that's all Ansible could do, it would... kinda suck.

Nope, what I *really* want is to be able to create a big config file that describes *all* of the modules that we need to execute to get an entire server set up: like installing PHP, installing MySQL, installing Nginx and then configuring everything.

This is done in something called a *playbook*: a YAML-formatted file that contains these instructions. In the ansible directory, create our playbook: playbook.yml. For now, just leave it empty.

To run the playbook, instead of using the ansible command, use ansible-playbook. Point that at your playbook and keep passing -i ansible/hosts.ini. Try it!

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

Error!

> Playbook must be a list of plays

Your First Play and Task

Ah, ok, this is a preview of some Ansible terminology. Let's start filling in the playbook. At the top, every YAML file in Ansible starts with three dashes (---):

```
6 lines | ansible/playbook.yml
---
... lines 2 - 6
```

It's not really important and doesn't meaning anything... it's just a YAML standard. Below, add host: all:

```
6 lines | ansible/playbook.yml
---
- hosts: all
... lines 3 - 6
```

Then, indent two spaces, add tasks, indent again, and add ping: ~:

```
6 lines | ansible/playbook.yml
---
- hosts: all

  tasks:
    - ping: ~
```

If you're new to YAML, you *do* need to be careful: the spacing and indentation are important. Try it:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

Sweet! It runs the ping module against *all* hosts: our local machine and the VM.

Ok, back to the terminology we saw in the error message. A playbook contains plays. In this case we have just *one* play that will run on all hosts. Later, if we added *another* host line with its own tasks below it, that would be a *second* play. So, a playbook contains plays, a play contains tasks and each task executes a module. In this case we're executing the ping module.

To run the play against only *one* group, there are two options. First, at the command line, you can pass -l vb:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini -l vb
```

But a better way is to configure this *inside* of your playbook. This play will be responsible for setting up an entire server... so we don't want to run it against our local machine... ever. Change the all to vb:

6 lines | ansible/playbook.yml

```yaml
---
- hosts: vb

  tasks:
    - ping: ~
```

Now try the command, but without the -l option:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

Yes! Just one ping.

Ok, let's install stuff!

Chapter 6: Install Stuff: The apt Module

Now we are dangerous! With the playbook setup, we can add more and more tasks that use more and more modules. One of the most useful modules is called apt - our best friend for installing things via apt-get on Debian or Ubuntu.

We're going to install a *really* important utility called cowsay. I already have it installed locally, so let's try it:

```
$ cowsay "I <3 Ansible"
```

OMG!

Tip

If you have cowsay installed locally, make sure to run export ANSIBLE_NOCOWS=1. Otherwise, Ansible will use cowsay for its output, which is hilarious, but a bit distracting.

Since this is absolutely necessary on any server that runs MooTube, let's add a second task to install it. Usually, I give my tasks a bit more structure, with a name that mentions how important this is. Below, add the module you want to use: apt:

10 lines | ansible/playbook.yml

```
---
- hosts: vb

  tasks:
    - ping: ~

    - name: Install cowsay - it's probably important
      apt:
... lines 9 - 10
```

If you check out the apt module docs, you'll see that *it* has an option called name, which is the package that we want to install. To pass this option to the module, indent on the next line, and add name: cowsay:

10 lines | ansible/playbook.yml

```
---
- hosts: vb

  tasks:
    - ping: ~

    - name: Install cowsay - it's probably important
      apt:
        name: cowsay
```

Done!

Run the playbook!

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

sudo: Using become

The ping works, but the install fails! Check out the error:

> Could not open lock file. Unable to lock the administration directory, are you root?

Of course! Ansible doesn't automatically run things with sudo. When a task *does* need sudo, it needs another option: become: true:

11 lines | ansible/playbook.yml

```
---
- hosts: vb

  tasks:
... lines 5 - 6
    - name: Install cowsay - it's probably important
      become: true
      apt:
        name: cowsay
```

This means that we want to *become* the super user. In our VM, the vagrant user can sudo without typing their password. But if that's not your situation, you *can* configure the password.

Try it again!

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

This time... it works! And notice, it says "changed" because it *did* install cowsay. Now try it again:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

Ah, hah! The second time it just says "Ok" with changed=0. Because remember: the module doesn't just dumbly run apt-get! Its real job is to guarantee that cowsay is in an installed "state".

Oh, and if you're Googling about Ansible, you might see become: yes. In Ansible, whenever you need a Boolean value like true or false, Ansible allows you to say "yes" or "no". Don't get surprised by that: "yes" means true and "no" means false. Use whichever you like!

Time to get our system setup for real, with PHP, Nginx and other goodies!

Chapter 7: apt Package Upgrades & Requirements

The apt module is the key to getting *so* much stuff installed. But, it can do more than that. When we first boot our server from the Ubuntu image, what guarantees that our packages aren't completely out of date? Nothing! In fact, I bet a lot of packages *are* old and need upgrading.

Check out the apt module options. See update_cache? That's equivalent to running apt-get update, which downloads the latest package lists from the Ubuntu repositories. We definitely need that. Then after, to actually upgrade the packages, we can use the upgrade option.

update_cache: Updating the Repositories Cache

Head back to your playbook and add a new task to update the APT package manager repositories cache. Add become: true, use the apt module, and set update_cache to yes:

21 lines | ansible/playbook.yml

```yaml
---
- hosts: vb

  tasks:
    - ping: ~

    - name: Update APT package manager repositories cache
      become: true
      apt:
        update_cache: yes
... lines 11 - 21
```

Remember, yes and true mean the same thing.

upgrade: Upgrading Packages

Cool! Copy that task to create the next one: upgrade the existing packages. Now, set upgrade to dist:

21 lines | ansible/playbook.yml

```yaml
---
- hosts: vb

  tasks:
    - ping: ~

    - name: Update APT package manager repositories cache
      become: true
      apt:
        update_cache: yes

    - name: Upgrade installed packages
      become: true
      apt:
        upgrade: dist
... lines 16 - 21
```

There are a few possible values for upgrade - some upgrade more aggressively than others.

Find your terminal and run that playbook!

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

The first time you do this... it might take awhile - like several minutes. So go get some coffee and bother your co-workers. And it makes sense that it's slow: our server probably *is* out of date, so this is doing a lot of work. Thanks to the power of TV, we'll fast-forward.

Yes! The "Upgrade installed packages" task says "changed". It *did* upgrade some stuff!

Module Requirements: Installing Aptitude (if needed)

Head back to the docs: one of the other values for the upgrade option is safe, which I kind of like because it's a bit more conservative than dist. When we use safe, it uses aptitude, instead of apt-get. That's important, because not all Ubuntu images come with aptitude installed out-of-the-box.

In fact, scroll up a bit. The apt module has a "Requirements" section. Interesting... It says that the host - meaning the virtual machine in our case - needs python-apt, which our VM has, and aptitude to be installed for things to work. So far, we think of modules as standalone workers that take care of everything for us... and that's mostly true. But sometimes, modules have *requirements*. And it's up to us to make sure those requirements are met *before* using the module.

Open a new terminal tab and SSH into the VM with:

```
$ vagrant ssh
```

Let's see if aptitude is installed:

```
$ aptitude
```

It opens! So it *is* installed. Hit "q" to quit.

In this case, the requirement is already met out-of-the-box. But in other situations, in fact, in older versions of this Ubuntu image, you may need to add a task to install aptitude via the apt module.

But now, just set upgrade to safe:

21 lines | ansible/playbook.yml

```yaml
---
- hosts: vb

  tasks:
... lines 5 - 11
    - name: Upgrade installed packages
      become: true
      apt:
        upgrade: safe
... lines 16 - 21
```

Then, try the playbook again to make sure it's still happy!

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

It didn't make any changes... but it *is* still happy! We rock!

With repositories cache and packages upgraded, we can go crazy and install everything we need.

Installing git

So let's add a task to install the Git version control system: we'll use it to "deploy" our code. Like always, become: true, use the apt module, and use name: git:

26 lines | ansible/playbook.yml

```yaml
---
- hosts: vb

  tasks:
... lines 5 - 21
    - name: Install Git VCS
      become: true
      apt:
        name: git
```

I'll move this below cowsay - but the order between these doesn't matter.

Try this out:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

It *looks* like it worked. How could we know? Move over to the terminal tab that's already SSH'ed into the VM. Run git --version. Yes!

Using state: latest

Back on the apt docs, there's an option called state with values latest, absent, present or build-dep. This shouldn't be surprising: this module is a lot smarter than simply running apt-get install: the module helps guarantee that a package is in a specific *state*, like "present" or "absent"... if you wanted to make sure that a package was *not* installed.

Add state to our task, set to latest:

27 lines | ansible/playbook.yml

```yaml
---
- hosts: vb

  tasks:
... lines 5 - 21
    - name: Install Git VCS
      become: true
      apt:
        name: git
        state: latest
```

Now, instead of simply making sure that the package is *present* on the system, it will make sure that it's upgraded to the *latest* available version. This setting is a bit more aggressive - so do what's best for you.

Try the playbook again!

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

Ok, it didn't make any changes: git is already at the latest version... which makes sense... because we just installed it a minute ago. But in the future, when a new version comes out, our playbook will grab it.

Ok, let's get PHP 7 installed!

Chapter 8: PHP 7, Nginx & MySQL

Our app is written in PHP... so... we should probably get that installed. Copy the git block, paste it, and change it to php5-cli:

33 lines | ansible/playbook.yml

```
---
- hosts: vb

  tasks:
... lines 5 - 27
    - name: Install PHP CLI
      become: true
      apt:
        name: php5-cli
        state: latest
```

Run that playbook!

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

And yea... I know PHP 5 is *ancient*. But our Ubuntu distribution doesn't support version 7. But don't worry, we're going to upgrade to 7 in a minute by using a custom repository. It's going to be awesome - so stay with me.

While the playbook is running, change over to your terminal tab and make sure you're still SSH'ed into the VM. Of course, if we try php -v right now... it doesn't work. But as *soon* as Ansible finishes, try it again:

```
$ php -v
```

Yea! Version 5.5.9. Now, let's kill this ancient version of PHP and go to 7.

Using a Custom apt Repository

Here's the deal: if you research how to install PHP 7 on this version of Ubuntu, you'll learn about a third-party repository called ppa:ondrej/php. If we can add this to our apt *sources* - usually done by running a few commands - we'll be in business.

And of course... there's a module for that! It's not apt, it's apt_repository. It doesn't have any requirements, and the options look pretty easy - just set repository to the one we want to use.

Let's do it! In the playbook, above the PHP task, add a new one: Add PHP 7 Personal Package Archive repository. Use the apt_repository module and set repository to: ppa:ondrej/php:

38 lines | ansible/playbook.yml

```yaml
---
- hosts: vb

  tasks:
... lines 5 - 27
    - name: Add PHP 7 PPA repository
      become: true
      apt_repository:
        repo: 'ppa:ondrej/php'

    - name: Install PHP CLI
... lines 34 - 38
```

And *now*, we can install php7.1-cli:

38 lines | ansible/playbook.yml

```yaml
---
- hosts: vb

  tasks:
... lines 5 - 32
    - name: Install PHP CLI
... line 34
      apt:
        name: php7.1-cli
... lines 37 - 38
```

Run it!

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

Love it... no errors, and 2 tasks changed. Over in the VM, try:

```
$ php -v
```

Oh, sweet PHP 7.1 goodness.

Install Nginx

We're on a role - so let's take care of installing a few more things, like Nginx! Add the new task: Install Nginx web server. I'm putting this above the PHP install, but it doesn't matter. Add the normal become: true, apt and install nginx with state: latest:

44 lines | ansible/playbook.yml

```
---
- hosts: vb

  tasks:
... lines 5 - 27
    - name: Install Nginx web server
      become: true
      apt:
        name: nginx
        state: latest

    - name: Add PHP 7 PPA repository
... lines 35 - 44
```

Run it!

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

And by the way, in the real world... where you can't fast-forward through Ansible, running the playbook takes awhile. So, you might want to add multiple tasks before testing things.

Ok, worked again! In theory, Nginx is now installed and running! Switch over to the VM and try to hit our server:

```
$ curl localhost
```

Hey hey! We are rocking! Sure, we still need to add a lot of Nginx configuration, but it *is* running.

And we should be able to see this page from our host machine. Find your browser and go to http://192.168.33.10. Hey again Nginx!

Now eventually, we're going to access the site via mootube.l from our host machine. To handle that, head to your terminal on your *main* machine - so not the VM - and edit your /etc/hosts file. Inside, anywhere, add 192.168.33.10 mootube.l. Save that!

Back at the browser test it: http://mootube.l. Got it!

Install MySQL

Before we move on, let's check one more thing off our list: MySQL. Copy the Nginx configuration to Install MySQL DB Server. Set it to the mysql-server package:

50 lines | ansible/playbook.yml

```
---
- hosts: vb

  tasks:
... lines 5 - 33
    - name: Install MySQL DB server
      become: true
      apt:
        name: mysql-server
        state: latest

    - name: Add PHP 7 PPA repository
... lines 41 - 50
```

Run Ansible again:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

Now, in a bigger infrastructure, you might want to keep your database on a separate server, or host, in Ansible language.

As we've seen from our playbook, each *play* is for a specific host. And there's nothing stopping us from having one host for our web server, with Nginx and PHP, and another host for our database, where we install the MySQL server. But for our example, we'll keep them all together.

Ok! Now MySQL should be installed. Move to your VM terminal tab, and try to connect to the local server:

```
$ mysql -u root
```

And we are in! Type exit to get out.

We've got Nginx, MySQL and PHP installed. But, PHP isn't ready yet... we're missing a bunch of PHP extension and our php.ini file needs some changes. Let's crush that!

Chapter 9: Extensions, php.ini & lineinfile

PHP, MySQL and Nginx: check, check and check! But we've *only* installed php7.1-cli. An in reality, we need a lot more than that! What about php7.1-mysql or php7.1-fpm? Yep, we need those friendly extensions... and a few others.

Looping: with_items

Wonderfully, on Ubuntu, these are all installed via apt-get. We *could* copy and paste the php7.1-cli task over and over and over again for each package. Or, to level-up our Ansible-awesomeness, we can loop!

Let's see how: change the task's name to Install PHP packages:

56 lines | ansible/playbook.yml

```
---
- hosts: vb

  tasks:
... lines 5 - 44
    - name: Install PHP packages
... lines 46 - 56
```

Then, instead of php7.1-cli, add the very cryptic "{{ item }}":

56 lines | ansible/playbook.yml

```
---
- hosts: vb

  tasks:
... lines 5 - 44
    - name: Install PHP packages
... line 46
      apt:
        name: "{{ item }}"
... lines 49 - 56
```

Finish it with a new with_items key *after* the apt module. This gets a big list of the stuff we want: php7.1-cli, php7.1-curl, ice cream, php7.1 -fpm, php7.1-intl, a pony and php7.1-mysql:

56 lines | ansible/playbook.yml

```
---
- hosts: vb

  tasks:
... lines 5 - 44
    - name: Install PHP packages
      become: true
      apt:
        name: "{{ item }}"
        state: latest
      with_items:
        - php7.1-cli
        - php7.1-curl
        - php7.1-fpm
        - php7.1-intl
        - php7.1-mysql
```

Tip

Using a loop in apt module is deprecated and will be removed in version 2.11. Instead of using the loop and specifying name: {{ item }}, you can pass an array to the name key and specify the items like this:

```
---
- hosts: vb

  tasks:
    # ...
    - name: Install PHP packages
      become: true
      apt:
        name:
          - php7.1-cli
          - php7.1-curl
          - php7.1-fpm
          - php7.1-intl
          - php7.1-mysql
        state: latest
```

If we need more goodies later, we can add them. Flip over to your terminal and try it!

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

Using Jinja

While we wait, let's check out the code we just wrote. For the *first* time, we're seeing Ansible's templating language in action! Yep, whenever you see {{ }}, you're writing Jinja code - a Python templating language... which - guess what - is more or less identical to Twig. Win!

In this case, we're opening up Jinja to print a variable called item. That works because Ansible has this nice with_items loop feature. And notice, this is *not* special to the apt module - it'll work anywhere.

Oh, and those quotes *are* important:

```
56 lines | ansible/playbook.yml
---
- hosts: vb

  tasks:
... lines 5 - 44
    - name: Install PHP packages
... line 46
      apt:
        name: "{{ item }}"
... lines 49 - 56
```

Quoting is usually optional in YAML. But if a value starts with {{, it's mandatory.

Head back to the terminal. Yes! Celebrate! PHP extensions installed! I'll move to my third tab where I've already run vagrant ssh to get into the VM. Check for the MySQL extension:

```
$ php -i | grep mysql
```

See that PDO MySQL stuff? That proves it worked!

Re-run that command again and look for timezone:

```
$ php -i | grep timezone
```

Tweaking php.ini settings with lineinfile

Hmm, it says date.timezone no value, which means that it is *not* set in php.ini. Since PHP 7.0, that's not a *huge* deal - in PHP 5 this caused an annoying warning. But, I still want to make sure it's set.

Question number 1 is... where the heck is my php.ini file? Answer, run:

```
$ php --ini
```

There it is /etc/php/7.1/cli/php.ini. Open that up in vim and hit /timezone, enter, to find that setting:

```
[Date]
; Defines the default timezone used by the date functions
; http://php.net/date.timezone
;date.timezone =
```

Ok, it's commented-out right now. We want Ansible to uncomment that line and set it to UTC. Quit with Escape, :q, enter.

So how can we tell Ansible to make a change *right* in the middle of a file? Of course, Ansible has a module *just* for that! Search for the "Ansible lineinfile" module. Ah, ha!

> Ensure a particular line is in a file, or replace an existing line

Let's check out the options! The only required one is path - the file we need to change. Then, we can use the regexp option to find the target line and line as the value to replace it with.

Before we do this, look back at the path option. It says that *before* Ansible 2.3, this was called dest, destfile or name instead of path. What version do we have? Find out:

```
$ ansible --version
```

We're on 2.1! So instead of path, we need to use dest. This is something to watch out for... because at the time of this recording, Ansible 2.3 isn't even released yet! For some reason, Ansible always shows the docs for its latest, unreleased version.

Let's rock! Add the new task: Set date.timezone for CLI. Add become: true and use the lineinfile module:

70 lines | ansible/playbook.yml

```yaml
---
- hosts: vb

  tasks:
... lines 5 - 56
    - name: Set date.timezone for CLI
      become: true
      lineinfile:
... lines 60 - 70
```

For options, pass it dest: /etc/php/7.1/cli/php.ini and regexp: date.timezone =:

70 lines | ansible/playbook.yml

```yaml
---
- hosts: vb

  tasks:
... lines 5 - 56
    - name: Set date.timezone for CLI
      become: true
      lineinfile:
        dest: /etc/php/7.1/cli/php.ini
        regexp: "date.timezone ="
... lines 62 - 70
```

We're not leveraging any regex here: this will simply find a line that contains date.timezone =. Finally, add line: date.timezone = UTC:

70 lines | ansible/playbook.yml

```yaml
---
- hosts: vb

  tasks:
... lines 5 - 56
    - name: Set date.timezone for CLI
      become: true
      lineinfile:
        dest: /etc/php/7.1/cli/php.ini
        regexp: "date.timezone ="
        line: "date.timezone = UTC"
... lines 63 - 70
```

With the line option, the *entire* line will be replaced - not just the part that was matched from the regexp option. That means the comment at the beginning of the line *will* be removed.

Tip

There is also an ini_file module, which makes modifying .ini files even easier. For an example, see: http://bit.ly/knpu-ini-module

Now, copy that entire task and paste it. In Ubuntu, there are 2 different php.ini files: rename this one to

Set date.timezone for FPM. Change the dest path from cli/ to fpm/. That's the correct path inside the VM:

```
70 lines | ansible/playbook.yml
---
- hosts: vb

  tasks:
... lines 5 - 63
    - name: Set date.timezone for FPM
      become: true
      lineinfile:
        dest: /etc/php/7.1/fpm/php.ini
        regexp: "date.timezone ="
        line: "date.timezone = UTC"
```

Run it!

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

Before that finishes, flip back to the VM and check the timezone setting:

```
$ php -i | grep timezone
```

No value. Then... once it finishes... try it again:

```
$ php -i | grep timezone
```

Got it! UTC! I'll open up my php.ini to be sure... and...

```
[Date]
; Defines the default timezone used by the date functions
; http://php.net/date.timezone
date.timezone = UTC
```

Yes! The line was perfectly replaced.

Say hello to lineinfile: your Swiss army knife for updating configuration files.

Chapter 10: git & Variables

With PHP setup, it's time to actually pull down our code onto the server. Let's think about what this process will look like. First, we need to create a directory. Then we need to clone our code with Git. Well there are several ways to get code onto a machine - but using Git is a really nice option.

The file Module

Let's create that directory. Not surprisingly, Ansible has module for this. Search for the "Ansible file module". Yes! This helps set attributes on files, symlinks and directories. If you need to create any of these or set permissions, this is your friend.

The most important option is path, but there are few other we'll need, like state, where you choose what type of "thing" the path should be, and also, owner, group and mode for permissions goodness.

You know the drill: create a new task: Create a project directory and set its permissions. Use become: true, use file and set path to, how about, /var/www/project. The state should be directory and add owner: vagrant and group: vagrant:

79 lines | ansible/playbook.yml

```
---
- hosts: vb

  tasks:
... lines 5 - 70
    - name: Create project directory and set its permissions
      become: true
      file:
        path: "/var/www/project"
        state: directory
        owner: "vagrant"
        group: "vagrant"
... lines 78 - 79
```

This will let our SSH user write these files.

Tip

In some setups, you might want to have your web-server user - e.g. www-data - be the owner if this directory. Then, you can use become: www-data on future calls to become that user.

Oh, and set recurse: true - in case /var/www doesn't exist, it'll create that!

79 lines | ansible/playbook.yml

```
---
- hosts: vb

  tasks:
... lines 5 - 70
    - name: Create project directory and set its permissions
      become: true
      file:
        path: "/var/www/project"
        state: directory
        owner: "vagrant"
        group: "vagrant"
        recurse: yes
```

Referencing Variables

But don't try this yet! Thanks to our hosts.ini setup, we've told Ansible that we want to SSH as the user vagrant. We did that by overriding a built-in variable called ansible_user. Well, guess what? We can *reference* that same variable in our playbook! Instead of hardcoding vagrant, use Jinja: {{ ansible_user }}. Repeat that next to group:

79 lines | ansible/playbook.yml

```
---
- hosts: vb

  tasks:
... lines 5 - 70
    - name: Create project directory and set its permissions
... line 72
      file:
... lines 74 - 75
        owner: "{{ ansible_user }}"
        group: "{{ ansible_user }}"
... lines 78 - 79
```

Start up your playbook!

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

The git Module

While we're waiting, let's move to the next step: cloning our code via Git. Unfortunately, Ansible does *not* have a Git module to help us... bah! Just kidding, it totally does! Search for the "Ansible git module" to find it.

The module *does* have Git as a requirement - but we already installed that. So our job is pretty simple: pass it the repo we want to clone, and the dest-ination we want to clone to.

Do it! Go to http://github.com/symfony/symfony-standard. To start, we'll pull down the Symfony Standard Edition *instead* of our MooTube code. But that's just temporary. I'll click the "Clone" button and copy the URL.

Now, add a task: Checkout Git repository:

```yaml
85 lines | ansible/playbook.yml
---
- hosts: vb

  tasks:
... lines 5 - 79
    - name: Checkout Git repository
... lines 81 - 85
```

We do *not* need become: true because we *own* the destination directory. Go straight to git, then repo set to the URL we just copied. For dest, put /var/www/project. Add force: yes: that'll discard uncommitted changes if there are any:

```yaml
85 lines | ansible/playbook.yml
---
- hosts: vb

  tasks:
... lines 5 - 79
    - name: Checkout Git repository
      git:
        repo: https://github.com/symfony/symfony-standard.git
        dest: "/var/www/project"
        force: yes
```

Head back to your terminal. Sweet! The directory *was* created! In the VM, the /var/www/project directory is empty.

Creating a Variable

Before we run the new git task, I want to solve *one* last thing: we have duplication! The directory name - /var/www/project is in *two* places:

```yaml
85 lines | ansible/playbook.yml
---
- hosts: vb

  tasks:
... lines 5 - 70
    - name: Create project directory and set its permissions
... line 72
      file:
        path: "/var/www/project"
... lines 75 - 79
    - name: Checkout Git repository
      git:
... line 82
        dest: "/var/www/project"
... lines 84 - 85
```

Lame!

Well, good news: in addition to overriding variables - like ansible_user - we can create *new* variables.

Go all the way to the top of the file - right below hosts - though the order doesn't matter. Add a new vars: key, then below, set symfony_root_dir: /var/www/project:

```
88 lines | ansible/playbook.yml
---
- hosts: vb

  vars:
    symfony_root_dir: /var/www/project
... lines 6 - 88
```

Copy that new variable name and use it *just* like before: {{ symfony_root_dir }}. Repeat that next to dest:

```
88 lines | ansible/playbook.yml
---
- hosts: vb

  vars:
    symfony_root_dir: /var/www/project

  tasks:
... lines 8 - 73
    - name: Create project directory and set its permissions
... line 75
      file:
        path: "{{ symfony_root_dir }}"
... lines 78 - 82
    - name: Checkout Git repository
      git:
... line 85
        dest: "{{ symfony_root_dir }}"
... lines 87 - 88
```

Ok, *now* I'm happy. Move over and try the playbook!

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

Looks good! Check the directory in the VM:

```
$ ls /var/www/project
```

Got it! The code is here, but it's not working yet: we need to install our Composer dependencies!

Chapter 11: Installing Composer & the script Module

With our project cloned, the next step is obvious: use Composer to install our dependencies! Actually, we used the Composer module earlier from the command line. Google again for "Ansible Modules" and find the "All Modules" page. I'll find the composer module and open that in a new tab.

This module is really easy... except for one problem. Under Requirements, it says that Composer already needs to be installed. We have *not* done that yet... and, unfortunately, it can't be installed with apt-get.

Installing Composer Programmatically?

So how *do* you install it? Check out https://getcomposer.org and click "Download".

Normally, we just paste these lines into our terminal and celebrate! But... there's this problematic fine print at the bottom:

> Do not redistribute the install code. It will change for every version of the install.

Huh. Composer includes a bit of built-in security: a sha hash to make sure that the installer hasn't been tampered with. If we tried to use these 4 commands in Ansible, it would work... for awhile. But next time the installer is updated, and that sha changed... it would *stop* working.

What to do? Check out that how to install Composer programmatically link. Eureka: a shell script that will safely download the latest version of Composer. The end result is a composer.phar file wherever we run this script from.

Installing Composer with script

Our mission is clear: somehow, execute this shell script via Ansible. But before we do that, near the top, add one new task: Install low-level utilities:

109 lines | ansible/playbook.yml

```yaml
---
- hosts: vb
... lines 3 - 6
  tasks:
... lines 8 - 24
    - name: Install low-level utilities
... lines 26 - 109
```

Here, use the apt module and the with_items syntax to install zip and unzip:

109 lines | ansible/playbook.yml

```yaml
---
- hosts: vb
... lines 3 - 6
  tasks:
... lines 8 - 24
    - name: Install low-level utilities
      become: true
      apt:
        name: "{{ item }}"
      with_items:
        - zip
        - unzip
... lines 32 - 109
```

Without these, Composer will run *really* slowly and you'll blame Jordi when you should be thanking him.

Now, back to our main job: how can we execute a script on a host? Why, with... the script module of course!

> Runs a local script on a remote node after transferring it

Neato! We just point it at a local script, and it takes care of the rest. Go copy the script and, in our ansible directory, create a new scripts directory and a new file called install_composer.sh. Paste the code there:

17 lines | ansible/scripts/install_composer.sh

```
#!/bin/sh

EXPECTED_SIGNATURE=$(wget -q -O - https://composer.github.io/installer.sig)
php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"
ACTUAL_SIGNATURE=$(php -r "echo hash_file('SHA384', 'composer-setup.php');")

if [ "$EXPECTED_SIGNATURE" != "$ACTUAL_SIGNATURE" ]
then
    >&2 echo 'ERROR: Invalid installer signature'
    rm composer-setup.php
    exit 1
fi

php composer-setup.php --quiet
RESULT=$?
rm composer-setup.php
exit $RESULT
```

Back in the playbook, at the bottom, create a new task: Download Composer:

109 lines | ansible/playbook.yml

```
---
- hosts: vb
... lines 3 - 6
  tasks:
... lines 8 - 96
    - name: Download Composer
... lines 98 - 109
```

Use the script module. Then, the easiest way to use this is to literally put the script filename on the same line as the module name: script: scripts/install_composer.sh:

109 lines | ansible/playbook.yml

```
---
- hosts: vb
... lines 3 - 6
  tasks:
... lines 8 - 96
    - name: Download Composer
      script: scripts/install_composer.sh
... lines 99 - 109
```

Actually, every module can be used with a one-line syntax like this... but since line breaks are pretty cheap these days, I usually organize things a bit more.

Thanks to this task, we'll have a new composer.phar file in our home directory, which is where this task - well, all tasks - are running. But that's not enough: we need to move this to /usr/local/bin/composer.

Moving Composer Globally

Create another task: Move Composer globally. This time, use become: true and use the command module:

```yaml
109 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 6
  tasks:
... lines 8 - 99
    - name: Move Composer globally
      become: true
... lines 102 - 109
```

In your browser, go find the command module. Like with script, command has a short syntax. We'll say:
command: mv composer.phar to /usr/local/bin/composer:

```yaml
109 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 6
  tasks:
... lines 8 - 99
    - name: Move Composer globally
      become: true
      command: mv composer.phar /usr/local/bin/composer
... lines 103 - 109
```

If you're a little surprised that I'm using the command module instead of some built-in file or move module... me too! In general, you should always look for a built-in module first: they're always more powerful than using command. But sometimes, like with moving files, command *is* the right tool for the job.

Add *one* more task to make sure the file is executable: "Set Permissions on Composer" with become: true:

```yaml
109 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 6
  tasks:
... lines 8 - 103
    - name: Set permissions on Composer
      become: true
... lines 106 - 109
```

Remember, Ansible is all about *state*. The job of the file module isn't really to *create* files or symlinks. Instead, it's to make sure that they *exist* and have the right permissions. In this case, we're going to take advantage of the mode option to guarantee that the file is executable.

In the playbook, use the file module, set path to /usr/local/bin/composer and mode to "a+x" to guarantee that all users have executable permission:

```yaml
109 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 6
  tasks:
... lines 8 - 103
    - name: Set permissions on Composer
      become: true
      file:
        path: /usr/local/bin/composer
        mode: "a+x"
```

Oh, and make sure the file you created is install_composer.sh.

Time to give this a try. Find your main machine's terminal and run the playbook!

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

Back on the VM, I'm in my home directory. And right now, it's empty. But if you're really fast, you can see the installation script doing its work. There it is: composer-setup.phar, composer-temp.phar, composer.phar and then it's gone once our task moves it. Yes!

And finally, we can type composer. Let's install some dependencies already!

Chapter 12: Installing Composer Deps

Ok, let's install our Composer dependencies already! Go back to the Ansible composer module for reference. Then, find your playbook and add a new task with a poetic and flowery name: "Install Composer's dependencies":

113 lines | ansible/playbook.yml

```
---
- hosts: vb
... lines 3 - 6
  tasks:
... lines 8 - 109
    - name: Install Composer's dependencies
... lines 111 - 113
```

Ok, boring name, but clear! Use the composer module, and set the one required option - working_dir - to {{ symfony_root_dir }}:

113 lines | ansible/playbook.yml

```
---
- hosts: vb
... lines 3 - 6
  tasks:
... lines 8 - 109
    - name: Install Composer's dependencies
      composer:
        working_dir: "{{ symfony_root_dir }}"
```

Hey, that variable is coming in handy!

Run that playbook!

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

It's running... running, installing Composer's dependencies and... explosion! Ah! So much red! Run!

Then... come back. Let's see what's going on. It *looks* like it was downloading stuff... if we move to /var/www/project on the VM and ls vendor/, yep, it *was* populated.

The problem was later - when one of Symfony's post-install tasks ran:

> Fatal error: Uncaught exception, SensioGeneratorBundle does not exist.

Oh yea. By default, the composer module runs composer like this:

```
$ composer install --no-dev
```

This means that your require-dev dependencies from composer.json are *not* installed:

78 lines | composer.json

```
{
... lines 2 - 35
    "require-dev": {
        "sensio/generator-bundle": "^3.0",
        "symfony/phpunit-bridge": "^3.0",

        "doctrine/data-fixtures": "^1.1",
        "hautelook/alice-bundle": "^1.3"
    },
... lines 43 - 76
}
```

If you're deploying to production, you may want that: it gives you a slight performance boost. But in a Symfony 3 application, it makes things blow up! You *can* fix this by setting an environment variable... and we *will* do that later.

But, since this is a development machine, we probably *do* want the dev dependencies. To fix that, in the playbook, set no_dev to no:

114 lines | ansible/playbook.yml

```
---
- hosts: vb
... lines 3 - 6
  tasks:
... lines 8 - 109
    - name: Install Composer's dependencies
      composer:
        working_dir: "{{ symfony_root_dir }}"
        no_dev: no
```

Try the playbook now.

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

This time, I personally guarantee it'll work. In fact, I'm *so* confident, that if it doesn't work this time, I'll buy you a beer or your drink of choice if we meet in person. Yep, it's *definitely* going to work - I've never been so sure of anything in my entire life.

Ah! No! It blew up again! Find the culprit!

> Attempted to load class "DOMDocument" from the global namespace.

Uh oh. I skipped past something I shouldn't have. When you download a new Symfony project, you can make sure your system is setup by running:

```
$ php bin/symfony_requirements
```

> Your system is not ready to run Symfony projects.

Duh! The message - about the SimpleXML extension - means that we're missing an extension! In our playbook, find the task where we install PHP. Add another extension: php7.1-xml:

```
115 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 6
  tasks:
... lines 8 - 55
    - name: Install PHP packages
... lines 57 - 60
      with_items:
... lines 62 - 66
        - php7.1-xml
... lines 68 - 115
```

Run that playbook - hopefully - one last time:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

Ya know, this is the *great* thing about Ansible. Sure, we might have forgotten to install an extension. But instead of installing it manually and forgetting all about it next time, it now lives permanently in our playbook. We'll never forget it again.

Phew! It worked! Go back to the VM and check out requirements again:

```
$ php bin/symfony_requirements
```

We're good! And most importantly, we can boot up our Symfony app via the console:

```
$ php bin/console
```

Our app is working! And there's just one last big step to get things running: configure NGINX with PHP-FPM and point it at our project. Let's go!

Chapter 13: Nginx Configuration & Ansible Templates

Nginx is still using its generic, boring, but very polite default HTML page:

> You're welcome for using Nginx!

It's time to point Nginx at *our* code! Google for "Symfony Nginx" to find a documentation page about Configuring a Web Server.

Scroll down to the Nginx section: it has a great block of Nginx config that'll work with our app. Here's the trick: we somehow need to put this configuration up onto the server *and* customize it. How can we do that?

Ansible *does* have a module for copying files from your local machine to the remote host. But in this case, there's a different module that's even *more* perfect: the template module. It says:

> Templates a file out to a remote server

In normal-person English, this means that it executes a file through Jinja - allowing us to print dynamic variables - and *then* copies it up to the remote server. I *love* this module.

Create a templates/ directory and inside, a symfony.conf file. Paste in the raw Nginx configuration:

55 lines | ansible/templates/symfony.conf

```
server {
    server_name domain.tld www.domain.tld;
    root /var/www/project/web;

    location / {
        # try to serve file directly, fallback to app.php
        try_files $uri /app.php$is_args$args;
    }
    # DEV
    # This rule should only be placed on your development environment
    # In production, don't include this and don't deploy app_dev.php or config.php
    location ~ ^/(app_dev|config)\.php(/|$) {
        fastcgi_pass unix:/var/run/php/php7.1-fpm.sock;
        fastcgi_split_path_info ^(.+\.php)(/.*)$;
        include fastcgi_params;
        # When you are using symlinks to link the document root to the
        # current version of your application, you should pass the real
        # application path instead of the path to the symlink to PHP
        # FPM.
        # Otherwise, PHP's OPcache may not properly detect changes to
        # your PHP files (see https://github.com/zendtech/ZendOptimizerPlus/issues/126
        # for more information).
        fastcgi_param SCRIPT_FILENAME $realpath_root$fastcgi_script_name;
        fastcgi_param DOCUMENT_ROOT $realpath_root;
    }
    # PROD
    location ~ ^/app\.php(/|$) {
        fastcgi_pass unix:/var/run/php/php7.1-fpm.sock;
        fastcgi_split_path_info ^(.+\.php)(/.*)$;
        include fastcgi_params;
        # When you are using symlinks to link the document root to the
        # current version of your application, you should pass the real
        # application path instead of the path to the symlink to PHP
        # FPM.
        # Otherwise, PHP's OPcache may not properly detect changes to
        # your PHP files (see https://github.com/zendtech/ZendOptimizerPlus/issues/126
        # for more information).
        fastcgi_param SCRIPT_FILENAME $realpath_root$fastcgi_script_name;
        fastcgi_param DOCUMENT_ROOT $realpath_root;
        # Prevents URIs that include the front controller. This will 404:
        # http://domain.tld/app.php/some-path
        # Remove the internal directive to allow URIs like this
        internal;
    }

    # return 404 for all other php files not matching the front controller
    # this prevents access to other php files you don't want to be accessible.
    location ~ \.php$ {
      return 404;
    }

    error_log /var/log/nginx/project_error.log;
    access_log /var/log/nginx/project_access.log;
}
```

Now, a few things need to change in here. Like, server_name should be mootube.l and root should point to the correct directory. Actually, root is already correct by chance... but we can do better!

Variables in your Template

All variables available in your playbook are available in a template. So at the top of our playbook, let's add a few more: server_name: mootube.l and symfony_web_dir set to {{ symfony_root_dir }}/web:

137 lines | ansible/playbook.yml

```yaml
---
- hosts: vb

  vars:
    server_name: mootube.l
    symfony_root_dir: /var/www/project
    symfony_web_dir: "{{ symfony_root_dir }}/web"
... lines 8 - 137
```

The web directory is our document root.

Let's put those variables to work! First, use {{ server_name }} and then set the root to {{ symfony_web_dir }}:

55 lines | ansible/templates/symfony.conf

```
server {
    server_name {{ server_name }};
    root {{ symfony_web_dir }};
... lines 4 - 53
}
```

At the bottom, tweak the logs paths - instead of the word project, use {{ server_name }}. Do it for the access_log too:

55 lines | ansible/templates/symfony.conf

```
server {
... lines 2 - 51
    error_log /var/log/nginx/{{ server_name }}_error.log;
    access_log /var/log/nginx/{{ server_name }}_access.log;
}
```

Oh, and while we're here, see those php5-fpm references? Change them to php7.1-fpm.sock in both places. FPM will already be configured to put its sock file here:

55 lines | ansible/templates/symfony.conf

```
server {
... lines 2 - 11
    location ~ ^/(app_dev|config)\.php(/|$) {
        fastcgi_pass unix:/var/run/php/php7.1-fpm.sock;
... lines 14 - 24
    }
    # PROD
    location ~ ^/app\.php(/|$) {
        fastcgi_pass unix:/var/run/php/php7.1-fpm.sock;
... lines 29 - 43
    }
... lines 45 - 53
}
```

Using the template Module

Let's add our task to use the template: "Add Symfony config template to the Nginx available sites". Add become: true and use the template module:

```
137 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 8
  tasks:
... lines 10 - 46
    - name: Add Symfony config template to the Nginx available sites
      become: true
      template:
... lines 50 - 137
```

The two important options are dest and src. Set src to, well templates/symfony.conf, and dest to /etc/nginx/sites-available/{{ server_name }}.conf:

```
137 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 8
  tasks:
... lines 10 - 46
    - name: Add Symfony config template to the Nginx available sites
      become: true
      template:
        src: templates/symfony.conf
        dest: "/etc/nginx/sites-available/{{ server_name }}.conf"
... lines 52 - 137
```

Enabling the Nginx Site

Cool! Once it's in that directory, we need to enable it... which means we need to create a symbolic link from sites-enabled to that file in sites-available. And we already know the *perfect* module for this: file!

Add the new task: "Enable Symfony config template from Nginx available sites". We still need become: true and use the file module:

```
137 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 8
  tasks:
... lines 10 - 46
    - name: Add Symfony config template to the Nginx available sites
... lines 48 - 52
    - name: Enable Symfony config template from Nginx available sites
      become: true
      file:
... lines 56 - 137
```

This time, for src, copy the sites-available line from dest above. For dest, just change it to sites-enabled:

```
137 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 8
  tasks:
... lines 10 - 46
    - name: Add Symfony config template to the Nginx available sites
... lines 48 - 52
    - name: Enable Symfony config template from Nginx available sites
      become: true
      file:
        src: "/etc/nginx/sites-available/{{ server_name }}.conf"
        dest: "/etc/nginx/sites-enabled/{{ server_name }}.conf"
        state: link
... lines 59 - 137
```

To create the symbolic link, use state: link. Earlier we created a directory with state: directory.

Updating /etc/hosts

The *last* thing I'll do - which is optional - is to add a task named "Add enabled Nginx site /etc/hosts". I'll use lineinfile to modify /etc/hosts and look for a line that contains {{ server_name }}. Then, I'll replace it with 127.0.0.1 {{ server_name }}:

```
137 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 8
  tasks:
... lines 10 - 52
    - name: Enable Symfony config template from Nginx available sites
... lines 54 - 59
    - name: Add enabled Nginx site to /etc/hosts
      become: true
      lineinfile:
        dest: /etc/hosts
        regexp: "{{ server_name }}"
        line: "127.0.0.1 {{ server_name }}"
... lines 66 - 137
```

This will guarantee that we have a /etc/hosts entry for mootube.l that points to 127.0.0.1 inside the VM. It's not a big deal - but it'll let us refer to mootube.l in the VM.

Of course, the first time this runs, it will *not* find a line in that file that matches mootube.l. But that's no problem: lineinfile will guarantee that the line value exists by adding it at the bottom.

Before you try this, make sure you have sites-available and sites-enabled:

137 lines | ansible/playbook.yml

```
---
- hosts: vb
... lines 3 - 8
  tasks:
... lines 10 - 46
    - name: Add Symfony config template to the Nginx available sites
... line 48
      template:
... line 50
        dest: "/etc/nginx/sites-available/{{ server_name }}.conf"
... line 52
    - name: Enable Symfony config template from Nginx available sites
... line 54
      file:
        src: "/etc/nginx/sites-available/{{ server_name }}.conf"
        dest: "/etc/nginx/sites-enabled/{{ server_name }}.conf"
... lines 58 - 137
```

Ok, run it!

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

Woohoo! No errors! Let's check it out! First, the file *does* live in sites-available/. And yea, awesome! The variables inside worked. Our sites-enabled directory *does* have the link, and /etc/hosts has our new mootube.l line at the bottom:

```
# /etc/hosts
# ...
127.0.0.1 mootube.l
```

Wow, we just killed it!

So, in theory, we should be able to go to our browser and refresh http://mootube.l to see our site. Refresh! Um.... we still see the same, boring - but polite - Nginx Configuration page.

Why? Some of you are probably screaming the answer at your computer... and also scaring your co-workers. It's because, we have *not* restarted or reloaded Nginx. Yea, this is easy to do manually - but come on! We need to teach our playbook to know when Nginx - and also PHP FPM - need to be restarted. We can do that with... handlers!

Chapter 14: Handlers: For Handling Serious Biz

Nginx *is* setup! But we need to restart or at least reload Nginx for our site to, ya know, *actually* work. Surprise! There's a module for that... called service. This module is all business: give it the name of the service and the state you want, like started or restarted.

So we should add a new task to restart Nginx, right? Wait, no! Ansible has a different option... a more *awesome* option.

Hello Handlers

At the bottom of the playbook, add a new section called handlers:

```
155 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 8
  tasks:
... lines 10 - 142
  handlers:
... lines 144 - 155
```

A handler is written *just* like any task: give it a name - "Restart Nginx" - use become: true, and then choose the module we want: service. Set the name to nginx and state to restarted:

```
155 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 8
  tasks:
... lines 10 - 142
  handlers:
    - name: Restart Nginx
      become: true
      service:
        name: nginx
        state: restarted
... lines 149 - 155
```

Tip

We could also just reload Nginx - that's enough for most changes.

Here's the deal with handlers: unlike tasks, they are *not* automatically called. Nope, instead, you find a task - like the task where we create the symbolic link to sites-enabled and, at the bottom, add notify: Restart Nginx:

```
155 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 8
  tasks:
... lines 10 - 54
    - name: Enable Symfony config template from Nginx available sites
... lines 56 - 60
      notify: Restart Nginx
... lines 62 - 155
```

Now, when this task runs, it will "notify" the "Restart Nginx" handler so that it will execute. Actually, that's kind of a lie - but just go with it for now.

There are a few reasons why this is better as a handler than a task. First, handlers run *after* all of the tasks, and if multiple tasks notify the same handler, that handler only runs once. Cool! I'll mention a second advantage to handlers in a minute.

Change over to our local machine's terminal and run the playbook!

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

We're expecting Ansible to execute all of the tasks... and *then* call the handler at the end. Wait! There's nothing at the end: it did *not* call the handler!

To prove it, refresh the browser: yep, we're still staring at the boring Nginx test page.

Handler and Changed State

So... this didn't work... and this is the *second* reason why handlers are great! A handler *only* runs if the task that's notifying it *changed*. If you look at the output, not surprisingly, almost every task says "Ok"... which means that they did *not* make a change to the server. The only two that *did* change are related to Composer... and honestly... those aren't really changing the server either. They just aren't smart enough - *yet* - to correctly report that they have not made a change.

The important one in our case is "Enable Symfony Config". The symbolic link was already there, so it did *not* change and so it did *not* notify the handler.

So let's delete that link and see what happens! In the VM, run:

```
$ sudo rm /etc/nginx/sites-enabled/mootube.l.conf
```

Try the playbook now!

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

Watch for the "changed" state... got it! And... yes! Running Handlers: Restart Nginx. *Now* try your browser. Woh! Ok, 502 bad gateway. Ya know, I'm going to say that's progress: Nginx *did* restart... but now something else is busted! Before we put on our fancy debuggin' hat, let's add the "Restart Nginx" notify everywhere else it's needed.

For example, if we decide to change the template, we need Nginx to restart, or reload if you prefer:

```yaml
155 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 8
  tasks:
... lines 10 - 47
    - name: Add Symfony config template to the Nginx available sites
... lines 49 - 52
      notify: Restart Nginx
... lines 54 - 155
```

Up further, when we *first* install Nginx, if this changes because of a new Nginx version, we also want to restart:

```yaml
155 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 8
  tasks:
... lines 10 - 40
    - name: Install Nginx web server
... lines 42 - 45
      notify: Restart Nginx
... lines 47 - 155
```

Restarting PHP-FPM

The *other* thing we might need to restart is PHP-FPM, like when we update php.ini. At the bottom, copy the handler and make a new one called "Restart PHP-FPM". Then, just replace the name - nginx with php7.1-fpm:

```yaml
155 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 8
  tasks:
... lines 10 - 142
  handlers:
... lines 144 - 149
    - name: Restart PHP-FPM
      become: true
      service:
        name: php7.1-fpm
        state: restarted
```

Copy the name of the handler.

We *definitely* need to run this if any PHP extensions are installed or updated. And also if we update php.ini:

```
155 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 8
  tasks:
... lines 10 - 80
    - name: Install PHP packages
... lines 82 - 92
      notify: Restart PHP-FPM
... lines 94 - 101
    - name: Set date.timezone for FPM
... lines 103 - 107
      notify: Restart PHP-FPM
... lines 109 - 155
```

Beautiful! Since we have *not* restarted php-fpm yet, I'll go to my VM so we can make one of these tasks change. Open php.ini:

```
$ sudo vim /etc/php/7.1/fpm/php.ini
```

I'll search for timezone and set this back to an empty string:

```
; ...
[Date]
; Defines the default timezone used by the date functions
; http://php.net/date.timezone
date.timezone =
; ...
```

Now, re-run the playbook!

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

Watch for it.... yes! Restart PHP-FPM.

Let's try it! Refresh the browser! Bah! Still a 502 bad gateway!? That's bad news. Debuggin' time! In the VM, tail the log:

```
$ tail /var/log/nginx/mootube.l_error.log
```

Ah ha! It doesn't see our socket file! That's because I messed up! The *true* socket path is /var/run/php/php7.1-fpm.sock.

Easy fix: in symfony.conf, add /php in both places:

```
55 lines | ansible/templates/symfony.conf
server {
... lines 2 - 11
    location ~ ^/(app_dev|config)\.php(/|$) {
        fastcgi_pass unix:/var/run/php/php7.1-fpm.sock;
... lines 14 - 24
    }
    # PROD
    location ~ ^/app\.php(/|$) {
        fastcgi_pass unix:/var/run/php/php7.1-fpm.sock;
... lines 29 - 43
    }
... lines 45 - 53
}
```

Start up the playbook again!

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

This is a *perfect* example of why handlers are so cool! Since the template task is "changed", Nginx should be restarted. That's *exactly* how it should work.

Try the browser now! Ah, 500 error! Again, I'm counting that as progress!

Tail the log once more:

```
$ tail /var/log/nginx/mootube.l_error.log
```

Ah, so Symfony is unable to create the cache directory. That's an easy... but interesting fix. Let's do it next.

Chapter 15: Cache Permissions

Our app is a *big* 500 error because Symfony can't write to its cache directory.

This is an easy fix... well mostly. Let's start with the easy part: if we 777 the var/ directory, we should be good.

Add a new task at the end: "Fix var directory permissions":

```
163 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 9
  tasks:
... lines 11 - 143
    - name: Fix var directory permissions
... lines 145 - 163
```

To refer to the var/ directory, I'll create another variable: symfony_var_dir set to {{ symfony_root_dir }}/var:

```
163 lines | ansible/playbook.yml
---
- hosts: vb

  vars:
... line 5
    symfony_root_dir: /var/www/project
... line 7
    symfony_var_dir: "{{ symfony_root_dir }}/var"
... lines 9 - 163
```

Back at the bottom, use the file module, set the path to the new variable, and state to directory:

```
163 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 9
  tasks:
... lines 11 - 143
    - name: Fix var directory permissions
      file:
        path: "{{ symfony_var_dir }}"
        state: directory
... lines 148 - 163
```

That'll create the directory if it doesn't exist, but, it should. Then, they key part: mode: 0777 and recurse: yes:

```
163 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 9
  tasks:
... lines 11 - 143
    - name: Fix var directory permissions
      file:
        path: "{{ symfony_var_dir }}"
        state: directory
        mode: 0777
        recurse: yes
... lines 150 - 163
```

Tip

If you're going to 777 your var/ directory, make sure that you've uncommented the umask calls in app.php, app_dev.php and bin/console:

```
26 lines | web/app.php
<?php
... lines 2 - 4
// If you don't want to setup permissions the proper way, just uncomment the following PHP line
// read http://symfony.com/doc/current/book/installation.html#checking-symfony-application-configuration-and-setup
// for more information
umask(0000);
... lines 9 - 26
```

```
33 lines | web/app_dev.php
<?php
... lines 2 - 5
// If you don't want to setup permissions the proper way, just uncomment the following PHP line
// read http://symfony.com/doc/current/book/installation.html#checking-symfony-application-configuration-and-setup
// for more information
umask(0000);
... lines 10 - 33
```

```
30 lines | bin/console
#!/usr/bin/env php

... lines 3 - 7
// if you don't want to setup permissions the proper way, just uncomment the following PHP line
// read http://symfony.com/doc/current/book/installation.html#configuration-and-setup for more information
umask(0000);
... lines 11 - 30
```

You can see this in the finished code download.

Ok, run the playbook!

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

By the way, needing to re-run the *entire* playbook after a tiny change is annoying! We'll learn a trick in a minute to help with this.

Done! Ok, switch back to your browser and try it. Woohoo! A working Symfony project... not *our* project yet, but still, winning! We'll use our *real* project next.

Fixing Permissions... in a more Secure Way?

Setting the directory permissions to 777 is easy... and perfectly fine for a development machine. But if this were a *production* machine, well, 777 isn't ideal... though honestly, a *lot* of people do this.

What's better? In a few minutes, we'll add a task to clear and warm up Symfony's cache. On a production machine, after you've done that, you can set the var/cache permissions *back* to be non-writeable, so 555. In theory, that should just work! But in practice, you'll probably need to tweak a few other settings to use non-filesystem cache - like making annotations cache in APC.

But, that's more about deployment - which we'll save for a different course!

Chapter 16: Symfony Console Commands

The project is working! Except... that it's not actually *our* project: this is the Symfony Standard Edition. Our cow customers are waiting: let's install MooTube!

Head over to https://github.com/knpuniversity/ansible to find the code behind this project. Copy the clone URL and open your editor. Find the spot where we clone the repo and use *our* new URL:

```
163 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 9
  tasks:
... lines 11 - 119
    - name: Checkout Git repository
      git:
        repo: https://github.com/knpuniversity/ansible.git
... lines 123 - 163
```

You know the drill: run the playbook!

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

Our repository is public... which makes life easy. If you have a *private* repository, you'll need to make sure that your server has access to it. They talk about that a bit in the git module docs. You can also use a deploy key.

Using the Console

Once we have the code, we need to setup a few other things, like the database. The README.md file talks about these: after you download the composer dependencies, you can set up the database by running these three commands. Each runs through Symfony's console: an executable file in the bin/ directory.

This is a perfect situation for the command module... because... well, we literally just need to run 3 commands. Head to your playbook. Right above the handlers, add a comment: "Symfony Console Commands". We'll start with a task called "Create DB if not exists":

```
174 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 10
  tasks:
... lines 12 - 151
    # Symfony console commands
    - name: Create DB if not exists
... lines 154 - 174
```

Use the command module. For the value... we need to know the path to that bin/console file.

This is another good spot for a variable! Create a new one called symfony_console_path set to {{ symfony_root_dir }}/bin/console:

```
174 lines | ansible/playbook.yml
---
- hosts: vb

  vars:
... line 5
    symfony_root_dir: /var/www/project
... lines 7 - 8
    symfony_console_path: "{{ symfony_root_dir }}/bin/console"
... lines 10 - 174
```

Use that in the command: {{ symfony_console_path }} doctrine:database:create --if-not-exists:

```
174 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 10
  tasks:
... lines 12 - 151
    # Symfony console commands
    - name: Create DB if not exists
      command: '{{ symfony_console_path }} doctrine:database:create --if-not-exists'
... lines 155 - 174
```

That last flag prevents an error if the database is already there.

Awesome! Copy that task to create the second one: "Execute migrations". Use doctrine:migrations:migrate --no-interaction:

```
174 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 10
  tasks:
... lines 12 - 151
    # Symfony console commands
    - name: Create DB if not exists
      command: '{{ symfony_console_path }} doctrine:database:create --if-not-exists'

    - name: Execute migrations
      command: '{{ symfony_console_path }} doctrine:migrations:migrate --no-interaction'
... lines 158 - 174
```

And add one more: "Load data fixtures". This is something that we only want to run if this is a development machine, because it resets the database. We'll talk about controlling that later.

For this command, use hautelook_alice:doctrine:fixtures:load --no-interaction:

```
174 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 10
  tasks:
... lines 12 - 151
    # Symfony console commands
    - name: Create DB if not exists
      command: '{{ symfony_console_path }} doctrine:database:create --if-not-exists'

    - name: Execute migrations
      command: '{{ symfony_console_path }} doctrine:migrations:migrate --no-interaction'

    - name: Load data fixtures
      command: '{{ symfony_console_path }} hautelook_alice:doctrine:fixtures:load --no-interaction'
... lines 161 - 174
```

Ok! The 3 commands are ready! Head back to the terminal. Woh! It exploded!

And actually... the reason is not that important: it says an error occurred during the cache:clear --no-warmup command. After we run composer install, Symfony runs several post install commands. One clears the cache. Changing from one project to an entirely *different* project temporarily put things in a weird state. This one time, in the virtual machine, just remove the cache manually:

```
$ rm -rf var/cache/*
```

Try the playbook now:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

This time composer install should work and *hopefully* our new commands will setup the database. By the way! A Symfony 3 app reads its configuration from a parameters.yml file... which is *not* committed to the repository. So... in theory, that file should *not* yet exist... and none of this should work. But that file *does* exist! Why? Thanks to a special line in composer.json, after composer install finishes, the parameters.yml.dist file is copied to parameters.yml. And thanks to that dist file, Symfony will try to connect to MySQL using the root user and no password. If that's *not* right, just modify the file on the VM directly for now. Later, we'll talk about how we could properly update this file.

Yes! It worked! Notice: the three new tasks all say *changed*. That's because the command module isn't smart enough to know whether or not these *actually* changed anything. But, more on that soon!

Find your browser and refresh! Welcome to MooTube! The fact that it's showing these videos means our database is working. Now, let's talk about *tags*: a cool way to help us run only *part* of our playbook.

Chapter 17: Tagging Tasks

Sometimes - *especially* when debugging - you just want to run only *some* of your playbook. Because... our playbook is getting so awesome... that, honestly, it takes some serious time to run!

For example, in my VM, I'm going to change the permissions on the var/ directory:

```
$ sudo chmod -R 555 var/
```

Definitely use sudo. Then, I'll remove the cache files:

```
$ sudo rm -rf var/cache/*
```

If you try the page now, it explodes! Ok, I don't expect my permissions to suddenly change like this under normal conditions. But, suppose that we had *just* hit this permission error for the first time and then added the "Fix var permissions" task. In that case, we would know that re-running the *entire* playbook should fix things.

But... couldn't we run *just* this *one* task? Yep! And a *great* way to do that is via *tags*.

Below the task, add tags, and then permissions:

176 lines | ansible/playbook.yml

```
---
- hosts: vb
... lines 3 - 10
  tasks:
... lines 12 - 144
    - name: Fix var directory permissions
... lines 146 - 150
      tags:
        - permissions
... lines 153 - 176
```

Now, from the command line, tell Ansible to *only* execute tasks with this tag: -t permissions:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini -t permissions
```

It still goes through its setup but then... yep! Only one task! Refresh the page. Permissions fixed!

Tagging for Deployment

Here's another example. Right now, our playbook has tasks for two separate jobs. Some tasks setup the server - making sure PHP, Nginx and other stuff is installed and configured. But others are really more about code deployment: making sure the project directory exists, cloning the code, installing composer dependencies, and setting up the database.

In the future - when we make changes to the code - we might want to *just* deploy that code... without going through all the server setup tasks. Let's add a new tag - deploy - to every step involved in deployment. See the task that creates the project directory? Yep, give it the deploy tag. Add it to "Checkout Git Repository" and also to the three tasks that install Composer:

195 lines | ansible/playbook.yml

```
---
- hosts: vb
... lines 3 - 10
  tasks:
... lines 12 - 111
    - name: Create project directory and set its permissions
... lines 113 - 119
      tags:
        - deploy

    - name: Checkout Git repository
... lines 124 - 127
      tags:
        - deploy

    - name: Download Composer
... line 132
      tags:
        - deploy

    - name: Move Composer globally
... lines 137 - 138
      tags:
        - deploy

    - name: Set permissions on Composer
... lines 143 - 146
      tags:
        - deploy

    - name: Install Composer's dependencies
... lines 151 - 153
      tags:
        - deploy
... lines 156 - 195
```

Actually, this is debatable: you might consider Composer as a "Server setup" task, not deployment. It's up to you.

Keep going! I'll add the task to everything that I want to run for *each* code update. It's not an exact science:

195 lines | ansible/playbook.yml

```
---
- hosts: vb
... lines 3 - 10
  tasks:
... lines 12 - 111
    - name: Create project directory and set its permissions
... lines 113 - 119
      tags:
        - deploy

    - name: Checkout Git repository
... lines 124 - 127
      tags:
        - deploy

    - name: Download Composer
... line 132
      tags:
        - deploy

    - name: Move Composer globally
... lines 137 - 138
      tags:
        - deploy

    - name: Set permissions on Composer
... lines 143 - 146
      tags:
        - deploy

    - name: Install Composer's dependencies
... lines 151 - 153
      tags:
        - deploy

    - name: Fix var directory permissions
... lines 158 - 162
      tags:
        - permissions
        - deploy

    # Symfony console commands
    - name: Create DB if not exists
... line 169
      tags:
        - deploy

    - name: Execute migrations
... line 174
      tags:
        - deploy

    - name: Load data fixtures
... line 179
      tags:
        - deploy
... lines 182 - 195
```

Let's see if it works! In the virtual machine, I'm going to manually edit a file:

```
$ vim app/Resources/views/default/index.html.twig
```

Let's add a few exclamation points to be *really* excited. Then hit escape, :wq to save. In the browser, that won't show up immediately - because we're in Symfony's prod environment. But if you add app_dev.php to the URL... yep! "Filter by Tag!".

By the way, going to app_dev.php only works because I've already modified some security logic in that file to allow me to access it:

33 lines | web/app_dev.php

```
... lines 1 - 10
// This check prevents access to debug front controllers that are deployed by accident to production servers.
// Feel free to remove this, extend it, or make something more sophisticated.
if (isset($_SERVER['HTTP_CLIENT_IP'])
    || isset($_SERVER['HTTP_X_FORWARDED_FOR'])
    || !(in_array(@$_SERVER['REMOTE_ADDR'], ['127.0.0.1', 'fe80::1', '::1', '192.168.33.1']) || php_sapi_name() === 'cli-server' || strpo
) {
    header('HTTP/1.0 403 Forbidden');
    exit('You are not allowed to access this file. Check '.basename(__FILE__).' for more information.');
}
... lines 20 - 33
```

Ok, back in our local machine, run the playbook... this time with -t deploy:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini -t deploy
```

Oh, much, much faster! Try the browser! Code deployed! You can also use --skip-tags if you want to get crazy and do the opposite:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini --skip-tags deploy
```

Next, let's talk about how we can "fix" the fact that some tasks say "Changed" *every* time we run them. Eventually, this will help us speed up our playbook.

Chapter 18: Idempotency, changed_when & Facts

I just ran the playbook with the deploy tag:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini -t deploy
```

Notice that several tasks say "Changed"... but that's a lie! The first two are related to Composer - we'll talk about those later. Right now, I want to focus on the last 4: fixing the directory permissions and the 3 bin/console commands:

```
195 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 10
  tasks:
... lines 12 - 156
    - name: Fix var directory permissions
      file:
        path: "{{ symfony_var_dir }}"
        state: directory
        mode: 0777
        recurse: yes
      tags:
        - permissions
        - deploy

    # Symfony console commands
    - name: Create DB if not exists
      command: '{{ symfony_console_path }} doctrine:database:create --if-not-exists'
      tags:
        - deploy

    - name: Execute migrations
      command: '{{ symfony_console_path }} doctrine:migrations:migrate --no-interaction'
      tags:
        - deploy

    - name: Load data fixtures
      command: '{{ symfony_console_path }} hautelook_alice:doctrine:fixtures:load --no-interaction'
      tags:
        - deploy
... lines 182 - 195
```

Changed and Idempotency

But first... why do we care? I mean, sure, it says "Changed" when nothing *really* changed... but who cares? First, let me give you a fuzzy, philosophical reason. Tasks are meant to be *idempotent*... which is a hipster tech word to mean that it should be safe to run a task over and over again without any side effects.

And in reality, our tasks *are* idempotent. If we run this "Fix var directory permissions" task over and over and over again... that's fine! Nothing weird will happen. It's simply that the tasks are *reporting* that something is changing each time... when really... it's not!

I know, I know... this seems like *such* a silly detail. But soon, we're going to start making decision in our playbook *based* on whether or not a task reports as "changed".

Actually, this is already happening:

```
195 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 10
  tasks:
... lines 12 - 182
  handlers:
    - name: Restart Nginx
      become: true
      service:
        name: nginx
        state: restarted
... lines 189 - 195
```

The "Restart Nginx" handler is *only* called when this task changes:

```
195 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 10
  tasks:
... lines 12 - 56
    - name: Enable Symfony config template from Nginx available sites
      become: true
      file:
        src: "/etc/nginx/sites-available/{{ server_name }}.conf"
        dest: "/etc/nginx/sites-enabled/{{ server_name }}.conf"
        state: link
      notify: Restart Nginx
... lines 64 - 195
```

So, as a best practice - as much as we can - we want our tasks to correctly report whether or not they changed.

Using changed_when: false

How do we fix this? Well, the first task - fixing var directory permissions - is a little surprising. This is a core module... so, shouldn't it be correctly reporting whether or not the permissions *actually* changed? Well yes... but when you set recurse to yes, it *always* says changed:

```
195 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 10
  tasks:
... lines 12 - 156
    - name: Fix var directory permissions
      file:
... lines 159 - 161
        recurse: yes
... lines 163 - 195
```

The easiest way to fix this is to add changed_when set to false:

```
196 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 10
  tasks:
... lines 12 - 156
    - name: Fix var directory permissions
... lines 158 - 162
      changed_when: false
... lines 164 - 196
```

That's not *perfect*, but it's fine here:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini -t deploy
```

Dynamic changed_when

But what about the other tasks... like creating the database?

```
196 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 10
  tasks:
... lines 12 - 167
    # Symfony console commands
    - name: Create DB if not exists
      command: '{{ symfony_console_path }} doctrine:database:create --if-not-exists'
      tags:
        - deploy
... lines 173 - 196
```

Technically, the first time we run this, it *will* create the database. But each time after, it does nothing! If we want this task to be smart, we need to *detect* whether it did or did *not* create the database.

And there's a really cool way to do that. First, add a register key under the task set to db_create_result:

```
202 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 10
  tasks:
... lines 12 - 167
    # Symfony console commands
    - name: Create DB if not exists
... line 170
      register: db_create_result
... lines 172 - 202
```

This will create a new variable containing info about the task, including its output. This is called a *fact*, because we're collecting facts about the system.

To see what it looks like, below this, temporarily add a debug task:

```
202 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 10
  tasks:
... lines 12 - 167
    # Symfony console commands
    - name: Create DB if not exists
... line 170
      register: db_create_result
... lines 172 - 174
    - debug:
... lines 176 - 202
```

This is a shorthand way of using the debug module. Add var: db_create_result to print that. Oh, and below, give it the deploy tag:

```
202 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 10
  tasks:
... lines 12 - 174
    - debug:
        var: db_create_result
      tags:
        - deploy
... lines 179 - 202
```

Ok, try it!

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini -t deploy
```

Whoa, awesome! Check this out. That variable shows when the task started, when it ended *and* most importantly, its output! It says:

> Database symfony for connection named default already exists. Skipped.

Ah, ha! Copy the "already exists. Skipped" part. We can use this in our playbook to know whether or not the task did anything.

How? Instead of saying changed_when: false, use an expression: not db_create_result.stdout - stdout is the key in the variable - not db_create_result.stdout|search('already exists. Skipped'):

```
201 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 10
  tasks:
... lines 12 - 167
    # Symfony console commands
    - name: Create DB if not exists
... line 170
      register: db_create_result
      changed_when: "not db_create_result.stdout|search('already exists. Skipped')"
... lines 173 - 201
```

If you use Twig, this will look familiar: we're reading a variable and piping it through some search filter, which comes from Jinja.

Tip

Using tests as filters is deprecated since Ansible v2.5 and will be removed in v2.9, use is %test_name% or is not %test_name% instead where %test_name% might be any test like success, failed, search, etc, for example:

```
# ansible/playbook.yml
---
- hosts: vb
  # ...
  tasks:
    # ...
    # Symfony console commands
    - name: Create DB if not exists
      command: '{{ symfony_console_path }} doctrine:database:create --if-not-exists'
      register: db_create_result
      changed_when: db_create_result.stdout is not search('already exists. Skipped')
```

For the migration task, we can do the same. Register the variable first: db_migrations_result. Copy the changed_when and paste that below:

```
201 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 10
  tasks:
... line 12

... lines 14 - 175
    - name: Execute migrations
... line 177
      register: db_migrations_result
      changed_when: "not db_migrations_result.stdout|search('No migrations to execute')"
... lines 180 - 201
```

So what language happens when there are *no* migrations to execute? Go to your virtual machine and run the migrations to find out:

```
$ ./bin/console doctrine:migrations:migrate --no-interaction
```

Yes! It says:

> No migrations to execute

That's the key! Copy that language. Now, the same as before: paste that into the expression and update the variable name to db_migration_results:

201 lines | ansible/playbook.yml

```
---
- hosts: vb
... lines 3 - 10
  tasks:
... line 12

... lines 14 - 175
    - name: Execute migrations
... line 177
      register: db_migrations_result
      changed_when: "not db_migrations_result.stdout|search('No migrations to execute')"
... lines 180 - 201
```

Awesome! Finally, the last task loads the fixtures. This is tricky because... technically, this task fully empties the database and re-adds the fixture each time. Because of that, you could say this is *always* changing something on the server.

So, you can let this say "changed" or set changed_when: false if you want all your tasks to show up as not changed. Unless we start relying on the changed state of this task to trigger other actions, it doesn't really matter.

Moment of truth: let's head to our terminal and try the playbook:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini -t deploy
```

Yes! The last 4 tasks are *all* green: not changed. Now, let's do something *totally* crazy - like run doctrine:database:drop --force on the virtual machine:

```
$ ./bin/console doctrine:database:drop --force
```

Try the playbook now: we *should* see some changes. Yes! Both the database create task and migrations show up as changed.

Ok, let's do more with facts by introducing environment variables.

Chapter 19: vars_prompt & Environment Variables

We missed a deploy step! We're not clearing Symfony's cache. That sounds like a job for a new task! Call it "Clear Cache". It's easy: we just need to run a command: {{ symfony_console_path }} cache:clear --env=prod:

206 lines | ansible/playbook.yml

```
---
- hosts: vb
... lines 3 - 10
  tasks:
... lines 12 - 188
    - name: Clear cache
      command: '{{ symfony_console_path }} cache:clear --env=prod'
... lines 191 - 206
```

Easy! Except... that --env=prod part is interesting. *Sometimes* I might want to deploy our app in the prod environment - like if we're deploying to production. But other times, like if we're deploying to some test machine, I might want to deploy our app in the dev environment.

What I'm saying is: I want that environment to be configurable. And actually, this is important in *two* places: here, and when installing the Composer dependencies.

Symfony Composer Deps in the prod Environment

Google for "Symfony deploy" to find a page on Symfony.com. Scroll down to a section about installing and updating your vendors. Ah, so it recommends that when you deploy, you use:

```
$ composer install --no-dev
```

That's actually what the composer module tried to do by default. But then we - via a no_dev option - told it to stop that nonsense!

206 lines | ansible/playbook.yml

```
---
- hosts: vb
... lines 3 - 10
  tasks:
... lines 12 - 149
    - name: Install Composer's dependencies
      composer:
... line 152
        no_dev: no
... lines 154 - 206
```

We *had* to do that because some of the Composer post-install commands in a Symfony app *require* the packages in the require-dev section.

In reality, this --no-dev flag is not a big deal. But, we *can* use it... as long as we set an environment variable: SYMFONY_ENV=prod. Yep, those problematic post-install commands are setup to look for this environment variable, and *not* to do certain things rely on the require-dev dependencies.

So this is our mission: make the environment configurable, use it in the "Clear Cache" task *and* set a new environment variable.

Prompting for Input?

How? Start at the top: add a new vars_prompt key with name set to symfony_env:

```
221 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 10
  vars_prompt:
    - name: symfony_env
... lines 13 - 221
```

Then, prompt: Enter the environment for your Symfony app (prod|dev|test):

```
221 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 10
  vars_prompt:
    - name: symfony_env
      prompt: "Enter the environment for your Symfony app (prod|dev|test)"
... lines 14 - 221
```

As you're probably guessing, Ansible will now *ask* us what environment we want to use. Set a default value to prod and private: no - you can set that to yes to obscure passwords as you type them:

```
221 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 10
  vars_prompt:
    - name: symfony_env
      prompt: "Enter the environment for your Symfony app (prod|dev|test)"
      default: prod
      private: no
... lines 16 - 221
```

Cool!

Setting an Environment Variable

Next question: how can we use this variable to set an environment variable? How about... an environment key! Yep, setting environment variables is a native task for Ansible. Set SYMFONY_ENV to {{ symfony_env|lower }}:

```
221 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 10
  vars_prompt:
    - name: symfony_env
... lines 13 - 16
  environment:
    SYMFONY_ENV: "{{ symfony_env|lower }}"
... lines 19 - 221
```

This uses the variable we just set... but pipes it through a lower filter... just in case we get crazy and use upper-case letters.

To see what this *all* looks like, at the top, let's debug some variables. First, debug one called ansible_env:

221 lines | ansible/playbook.yml

```
---
- hosts: vb
... lines 3 - 19
  tasks:
    - debug:
        var: ansible_env
... lines 23 - 221
```

This is a built-in variable that has a lot of info about the "host" environment - meaning, the machine (or machines) that you're running Ansible against. It *should* also contain our environment variable.

Let's also debug the symfony_env variable that we set above:

221 lines | ansible/playbook.yml

```
---
- hosts: vb
... lines 3 - 19
  tasks:
    - debug:
        var: ansible_env

    - debug:
        var: symfony_env
... lines 26 - 221
```

Oh, and down on the "Clear Cache" task, I forgot to add the tag for deploy:

221 lines | ansible/playbook.yml

```
---
- hosts: vb
... lines 3 - 19
  tasks:
... lines 21 - 203
    - name: Clear cache
      command: '{{ symfony_console_path }} cache:clear --env=prod'
      tags:
        - deploy
... lines 208 - 221
```

Change over to your terminal and run the playbook - but take off the -t option so that *everything* runs:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

Yes! *Right* at the start, it asks us for the environment. I'll leave it blank to use prod. Then, hit ctrl+c to quit: we can already see the variables!

First, it printed ansible_env... which has some pretty cool stuff! It has a HOME key for the home directory, PWD for the current directory and other goodies. AND, it has SYMFONY_ENV set to prod. Not surprisingly, the symfony_env variable also prints prod.

Try this again.. but be tricky... with an uppercase PROD:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

Yep! The environment variable was lowercased, but not the symfony_env variable. That's no surprise... but if we want to guard against this, it *will* be a problem in a minute when we try to use this in more places... like down on my "Clear Cache" task:

```
221 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 19
  tasks:
... lines 21 - 203
    - name: Clear cache
      command: '{{ symfony_console_path }} cache:clear --env=prod'
... lines 206 - 221
```

We *could* keep using the lower filter. But, there's a cooler way: a "pre task".

Chapter 20: pre_tasks and set_fact

I want to *guarantee* that the symfony_env variable is *always* lowercased so that I can safely use. A cool way to do this is with a "pre task".

Add a new key called... well... pre_tasks and a new task called "Convert entered Symfony environment to lowercase":

228 lines | ansible/playbook.yml

```
---
- hosts: vb
... lines 3 - 19
  pre_tasks:
    - name: Convert entered Symfony environment to lowercase
... lines 22 - 26
  tasks:
... lines 28 - 228
```

Pre-tasks are exactly like tasks... they just run *first*.

Wait... then what the heck is the difference between a "pre task" and just putting the task at the top of the task section?

Nothing! Well, nothing yet. But later, when we talk about *roles* - ooh roles are fancy - there *will* be a difference: pre tasks run before roles.

Anywho, in this pre task, we basically want to re-set the symfony_env variable to a lowercased version of itself. To do that, we'll use a new module: set_fact. We already know that we can set variables in 3 different ways: with vars, vars_prompt or by using the register key below a task. The set_fact module is yet *another* way.

Facts versus Variables

But wait... why set_fact and not set_variable? So... here's the deal: you'll hear the words "facts" and "variables" in Ansible... basically interchangeably. Both facts and variables can be set, and are referenced in exactly the same way. And while there *do* seem to be some subtle differences between the two, if you think of a fact and a variable as the same thing, it'll make your life easier. When you run the playbook, the first task is called "setup", and it prepares some *facts* about the host machine. That's usually where you hear the word facts: it's info about each host.

So, we're using set_fact... to set a fact... or a variable... Set symfony_env to {{ symfony_env|lower }}:

228 lines | ansible/playbook.yml

```
---
- hosts: vb
... lines 3 - 19
  pre_tasks:
    - name: Convert entered Symfony environment to lowercase
      set_fact:
        symfony_env: "{{ symfony_env|lower }}"
... lines 24 - 228
```

Love it! Oh, and just like with tasks, these can be tagged. We'll probably want this to run *all* the time... so let's use a special tag called always:

```
228 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 19
  pre_tasks:
    - name: Convert entered Symfony environment to lowercase
      set_fact:
        symfony_env: "{{ symfony_env|lower }}"
      tags:
        - always
... lines 26 - 228
```

Try the playbook!

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

Be difficult and use "PROD" in all capital letters again.

Ha, ha! This time, our playbook outsmarts us and lowercases the value.

Environment Variables and Pre Tasks

Remove the two debug tasks:

```
222 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 16
  environment:
    SYMFONY_ENV: "{{ symfony_env|lower }}"
... lines 19 - 26
  tasks:
    - ping: ~
... lines 29 - 222
```

Can we also remove the |lower from the environment variable? Actually... no! The environment section runs *before* tasks, even pre_tasks. And this has nothing to do with the order we have things in this file - environment always runs first. So, keep the |lower.

Using the symfony_env Variable

Ok, time to use our fancy symfony_env variable! First, when we install the composer dependencies, we currently have no_dev: no. But now, if the environment is prod, this can be yes. Let's use a fancy expression!
{{ 'yes' if (prod == symfony_env) else 'no' }}:

```
224 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 26
  tasks:
... lines 28 - 165
    - name: Install Composer's dependencies
      composer:
... line 168
        no_dev: "{{ 'yes' if ('prod' == symfony_env) else 'no' }}"
... lines 170 - 224
```

Don't forget your curly braces. Weird, but cool! This is a special Jinja syntax.

Conditionally Running Tasks

Next, find the fixtures task. Hmm. If we're deploying in the prod environment... we might not want to load the data fixtures at *all*. But so far, we don't have any way to *conditionally* run a task: tasks *always* run.

Well guess what? We *can* tell a task to *not* run with the when key. In this case, say when: 'symfony_env != "prod":

224 lines | ansible/playbook.yml

```
---
- hosts: vb
... lines 3 - 26
  tasks:
... lines 28 - 198
    - name: Load data fixtures
... line 200
      when: symfony_env != "prod"
... lines 202 - 224
```

Finally, down in the Clear cache task, instead of prod, use {{ symfony_env }}:

224 lines | ansible/playbook.yml

```
---
- hosts: vb
... lines 3 - 26
  tasks:
... lines 28 - 205
    - name: Clear cache
      command: '{{ symfony_console_path }} cache:clear --env={{ symfony_env }}'
... lines 208 - 224
```

Let's try this thing! Re-run the playbook, but use -t deploy:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini -t deploy
```

Use the prod environment. As we just saw, the vars_prompt runs *always*: it doesn't need a tag:

224 lines | ansible/playbook.yml

```
---
- hosts: vb
... lines 3 - 10
  vars_prompt:
    - name: symfony_env
... lines 13 - 224
```

Then, our "pre task" should run thanks to the always tag:

```
224 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 19
  pre_tasks:
    - name: Convert entered Symfony environment to lowercase
... lines 22 - 23
      tags:
        - always
... lines 26 - 224
```

By the time the "Composer Install" task executes, it should run with no_dev: yes, and then hopefully it'll skip data fixtures and change "Clear cache":

```
224 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 26
  tasks:
... lines 28 - 165
    - name: Install Composer's dependencies
      composer:
... line 168
        no_dev: "{{ 'yes' if ('prod' == symfony_env) else 'no' }}"
... lines 170 - 198
    - name: Load data fixtures
... line 200
      when: symfony_env != "prod"
... lines 202 - 205
    - name: Clear cache
      command: '{{ symfony_console_path }} cache:clear --env={{ symfony_env }}'
... lines 208 - 224
```

The "Install Composer's dependencies" *does* show as *changed*: that's a good sign: it should have installed less packages than before. And yea! It's skipping the fixtures!

In the VM, try to ls vendor/sensio. Ok cool! One of the require-dev dependencies is sensio/generator-bundle. That is *not* here, proving that the dev dependencies did NOT install this time. We are in business!

And before we continue, under the "Clear Cache" task, add changed_when: false:

```
224 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 26
  tasks:
... lines 28 - 205
    - name: Clear cache
... line 207
      changed_when: false
... lines 209 - 224
```

That's not critical, it'll just prevent it from showing up as changed on every run.

Now, let's create a faster, smarter playbook by skipping some redundant tasks!

Chapter 21: Faster Smarter Playbook

If you executed the playbook twice in a row, you would see "Changed 2". On *every* playbook run, we're doing unnecessary work: we're *always* downloading Composer and then moving it globally... even if it's already there!

That's not a big deal... but it *is* wasteful. Let's make our playbook smarter by running tasks *conditionally*, based on *facts* that we collect about the host. In other words: let's *not* download Composer if it already exists!

Checking if a File Exists: stat

So that's the first mission: figure out if the /usr/local/bin/composer file exists. To do that, we can use a really handy module called stat. It's similar to the Unix stat command... which gives you a ton of info about a file.

Before we download composer, add a new task called "Check for Composer". Use the stat module and set the path to /usr/local/bin/composer. Then, register a new variable called composer_stat. And don't forget the deploy tag!

```
236 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 26
  tasks:
... lines 28 - 146
    - name: Check for Composer
      stat:
        path: /usr/local/bin/composer
      register: composer_stat
      tags:
        - deploy
... lines 153 - 158
    - name: Download Composer
... lines 160 - 236
```

To see what goodies that variable has in it, add a debug task to print composer_stat. Give that the deploy tag too:

```
236 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 26
  tasks:
... lines 28 - 146
    - name: Check for Composer
      stat:
        path: /usr/local/bin/composer
      register: composer_stat
      tags:
        - deploy

    - debug:
        var: composer_stat
      tags:
        - deploy
... lines 158 - 236
```

Ok! Change over to your terminal and run the playbook with -t deploy:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini -t deploy
```

There it is! Hit CTRL+C to quit.

The new variable has a stat key with a lot of info. The key we want is exists. Remove the debug task.

Skipping Tasks

Our goal is to *skip* the next three tasks if that file exists. Like before, use the when key set to not composer_stat.stat.exists:

```
233 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 26
  tasks:
... lines 28 - 153
    - name: Download Composer
... line 155
      when: not composer_stat.stat.exists
... lines 157 - 233
```

In other words, check the stat.exists key, and if that is true, we want to *not* run this.

Copy that and put it below "Move Composer globally", and also "Set Permissions on Composer":

```
233 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 26
  tasks:
... lines 28 - 159
    - name: Move Composer globally
... lines 161 - 162
      when: not composer_stat.stat.exists
... lines 164 - 233
```

I think we're ready! Try it:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini -t deploy
```

Wait... yes! Skipped! Stop the playbook.

Upgrading Composer

This is a nice performance improvement... but like with a lot of things in programming, new features mean new complexity. Thanks to this... the Composer executable will eventually become *really* old and out-of-date. We need to make sure it's upgraded to the latest version.

Below the set permissions task, add a new one called "Make sure Composer is at its latest version". This time we can use the composer module. Set working_dir to {{ symfony_root_dir }}... though that doesn't matter in this case... because we'll use self-update to upgrade Composer:

```yaml
240 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 26
  tasks:
... lines 28 - 166
    - name: Set permissions on Composer
... lines 168 - 174
    - name: Make sure composer is at its latest version
      composer:
        working_dir: "{{ symfony_root_dir }}"
        command: self-update
... lines 179 - 240
```

We can even add when: composer_stat.stat.exists to avoid running this if Composer was just downloaded. Give it our favorite deploy tag:

```yaml
240 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 26
  tasks:
... lines 28 - 174
    - name: Make sure composer is at its latest version
      composer:
        working_dir: "{{ symfony_root_dir }}"
        command: self-update
      tags:
        - deploy
... lines 181 - 240
```

Ok, run it! But this time, add a --verbose flag:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini -t deploy --verbose
```

This is a *sweet* trick... because... as you can see, it prints the output from each task... which can save us from needing to add so many debug tasks to print variables.

Boom! The new Composer upgrade task ran! I'll stop with "CTRL+C". And, we can see the stdout:
You are already using composer version 1.4.1.

So it worked! But like with other tasks... it's reporting as "Changed" even though it did *not* actually change anything! We already know how to fix that. Scroll down to the "Create DB" task and copy its changed_when. Scroll back up and paste it under this task. Register a new variable - composer_self_update - and use that in changed_when. Then, all we need to do is search for You are already using composer version:

242 lines | ansible/playbook.yml

```
---
- hosts: vb
... lines 3 - 26
  tasks:
... lines 28 - 174
    - name: Make sure composer is at its latest version
... lines 176 - 178
      register: composer_self_update
      changed_when: "not composer_self_update.stdout|search('You are already using composer version')"
... lines 181 - 242
```

If that shows up in the command, then we know nothing changed. Paste that into the search filter.

Test it out!

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini -t deploy --verbose
```

There it is! This time it's Okay - not changed. And on that note, we can even start skipping tasks based on whether or not other tasks are "changed" or "ok". Let's try that next!

Chapter 22: Skipping Tasks based on Changed

Thanks to the when key, you can make your playbooks really really smart and really really fast. What else could we do? Well, sometimes, our code doesn't change. When that happens, the "Checkout Git repository" will report as "ok", so *not* Changed.

If we know that the code didn't change... then it might not make sense to install our composer dependencies. After all, if the code didn't change, how would the composer dependencies need to change? And we could skip other things, like running the migration or even clearing the cache.

Under the "Git" task, register a new variable: repo_code:

```
248 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 26
  tasks:
... lines 28 - 138
    - name: Checkout Git repository
... lines 140 - 143
      register: repo_code
... lines 145 - 248
```

We already know from the output that we're looking for the changed key on this variable. That means, we could use repo_code.changed in the when option of some tasks to skip them.

Using set_fact to Clean Variables

But, we can get fancier! Below this task, add a new one called "Register code_changed variable". We'll use the set_fact module from earlier. This time, create a new variable called code_changed set to, very simply, repo_code.changed:

```
250 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 26
  tasks:
... lines 28 - 138
    - name: Checkout Git repository
... lines 140 - 143
      register: repo_code
... lines 145 - 147
    - name: Register code_changed variable
      set_fact:
        code_changed: repo_code.changed
... lines 151 - 250
```

The *only* reason we're doing this is to make our when statements a little cleaner. Add our tag onto that.

Down below, under "Install Composer's Dependencies", add when: code_changed:

```
250 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 26
  tasks:
... lines 28 - 188
    - name: Install Composer's dependencies
... lines 190 - 194
      when: code_changed
... lines 196 - 250
```

Tip

Ansible does not allow evaluating bare variables anymore. The code_changed variable is not obviously a boolean value because it just references to another variable called repo_code.changed, so Ansible requires the |bool filter to be added to the code_changed in when clauses:

```
# ansible/playbook.yml
---
- hosts: vb
  # ...
  tasks:
    # ...
    - name: Install Composer's dependencies
      # ...
      when: code_changed | bool
```

Ah, so nice. Copy that and put it anywhere else it makes sense like "Execute migrations" and "Clear Cache":

```
250 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 26
  tasks:
... lines 28 - 215
    - name: Execute migrations
... lines 217 - 221
      when: code_changed
... lines 223 - 230
    - name: Clear cache
... lines 232 - 235
      when: code_changed
... lines 237 - 250
```

Phew! Ok, run the playbook - but take off the verbose flag:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini -t deploy
```

As cool as this is, you need to be careful. As you make your playbook smarter, it's also more and more possible that you introduce a bug: you skip a task when it should actually run. I've just made a small mistake... which we'll discover soon.

But for now, it's so cool: the 3 Composer tasks were skipped, as well as installing Composer's dependencies, migrations and the "Clear Cache". And thanks to that, the playbook ran *way* faster than before.

It's time to talk about getting our playbook a bit more organized. As you can see... it's getting big... and I'm getting a bit lost in it. Fortunately, we have a few good ways to fix this.

Chapter 23: Organizing with include

Our playbook is a mess: there's just a *lot* of tasks in one file. Let's get organized!

There are a few "categories" of things in the playbook. One category deals with "Bootstrapping Symfony", like installing the composer dependencies, fixing the var directory permissions and all of our Symfony console commands:

250 lines | ansible/playbook.yml

```yaml
---
- hosts: vb
... lines 3 - 26
  tasks:
... lines 28 - 188
    - name: Install Composer's dependencies
      composer:
        working_dir: "{{ symfony_root_dir }}"
        no_dev: "{{ 'yes' if ('prod' == symfony_env) else 'no' }}"
      tags:
        - deploy
      when: code_changed

    - name: Fix var directory permissions
      file:
        path: "{{ symfony_var_dir }}"
        state: directory
        mode: 0777
        recurse: yes
      changed_when: false
      tags:
        - permissions
        - deploy

    # Symfony console commands
    - name: Create DB if not exists
      command: '{{ symfony_console_path }} doctrine:database:create --if-not-exists'
      register: db_create_result
      changed_when: "not db_create_result.stdout|search('already exists. Skipped')"
      tags:
        - deploy

    - name: Execute migrations
      command: '{{ symfony_console_path }} doctrine:migrations:migrate --no-interaction'
      register: db_migrations_result
      changed_when: "not db_migrations_result.stdout|search('No migrations to execute')"
      tags:
        - deploy
      when: code_changed

    - name: Load data fixtures
      command: '{{ symfony_console_path }} hautelook_alice:doctrine:fixtures:load --no-interaction'
      when: symfony_env != "prod"
      changed_when: false
      tags:
        - deploy

    - name: Clear cache
      command: '{{ symfony_console_path }} cache:clear --env={{ symfony_env }}'
      changed_when: false
      tags:
        - deploy
      when: code_changed
... lines 237 - 250
```

Using include()

I'd like to isolate these tasks into their own file. And... there are 2 good ways to do that. The first is.... just to include it. Create a new directory inside ansible called includes/, and inside there, a new file called symfony-bootstrap.yml. Start with the --- on top:

50 lines | ansible/includes/symfony-bootstrap.yml

```
---
... lines 2 - 50
```

Now, let's move some tasks here! Grab "Install Composer Dependencies" and move it. We also want "Fix var directory permissions" all the way down to the end: "Clear Cache". Delete all of it and paste it into symfony-bootstrap.yml:

50 lines | ansible/includes/symfony-bootstrap.yml

```yaml
---
- name: Install Composer's dependencies
  composer:
    working_dir: "{{ symfony_root_dir }}"
    no_dev: "{{ 'yes' if ('prod' == symfony_env) else 'no' }}"
  tags:
    - deploy
  when: code_changed

- name: Fix var directory permissions
  file:
    path: "{{ symfony_var_dir }}"
    state: directory
    mode: 0777
    recurse: yes
  changed_when: false
  tags:
    - permissions
    - deploy

# Symfony console commands
- name: Create DB if not exists
  command: '{{ symfony_console_path }} doctrine:database:create --if-not-exists'
  register: db_create_result
  changed_when: "not db_create_result.stdout|search('already exists. Skipped')"
  tags:
    - deploy

- name: Execute migrations
  command: '{{ symfony_console_path }} doctrine:migrations:migrate --no-interaction'
  register: db_migrations_result
  changed_when: "not db_migrations_result.stdout|search('No migrations to execute')"
  tags:
    - deploy
  when: code_changed

- name: Load data fixtures
  command: '{{ symfony_console_path }} hautelook_alice:doctrine:fixtures:load --no-interaction'
  when: symfony_env != "prod"
  changed_when: false
  tags:
    - deploy

- name: Clear cache
  command: '{{ symfony_console_path }} cache:clear --env={{ symfony_env }}'
  changed_when: false
  tags:
    - deploy
  when: code_changed
```

Now that these are in their own file, they should *not* be indented: so un-indent them twice.

In the playbook, to bring those in - it's really cool: where you would normally put a module, say include: ./includes/symfony-bootstrap.yml:

```
204 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 26
  tasks:
... lines 28 - 188
    # Bootstrap Symfony app
    - include: ./includes/symfony-bootstrap.yml

  handlers:
... lines 193 - 204
```

Tip

include module is deprecated since 2.4 and will be removed in version 2.8, use import_tasks module instead.

The tasks *will* run in a *slightly* different order than they did before, but it won't make any difference for us. But, to be sure try the playbook with -t deploy:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini -t deploy
```

All good! Including a file is the easiest way to organize and re-use tasks. But, there's a better, cooler, more robust, more hipster way. It's called roles.

Chapter 24: Organizing into Roles

Using include isn't the only way to organize your playbook. In fact, the *best* way is with roles... which are a *really* important concept in Ansible.

If you think of a package of functionality - like bootstrapping Symfony, or getting Nginx set up - it involves a number of things. In the case of Nginx, sure, we definitely need to run some tasks. But we also need to set a variable that's used by those tasks *and* register the "Restart Nginx" handler!

204 lines | ansible/playbook.yml

```yaml
---
- hosts: vb
... lines 3 - 26
  tasks:
... lines 28 - 58
    - name: Install Nginx web server
      become: true
      apt:
        name: nginx
        state: latest
      notify: Restart Nginx

    - name: Add Symfony config template to the Nginx available sites
      become: true
      template:
        src: templates/symfony.conf
        dest: "/etc/nginx/sites-available/{{ server_name }}.conf"
      notify: Restart Nginx

    - name: Enable Symfony config template from Nginx available sites
      become: true
      file:
        src: "/etc/nginx/sites-available/{{ server_name }}.conf"
        dest: "/etc/nginx/sites-enabled/{{ server_name }}.conf"
        state: link
      notify: Restart Nginx

    - name: Add enabled Nginx site to /etc/hosts
      become: true
      lineinfile:
        dest: /etc/hosts
        regexp: "{{ server_name }}"
        line: "127.0.0.1 {{ server_name }}"
... lines 87 - 191
  handlers:
    - name: Restart Nginx
      become: true
      service:
        name: nginx
        state: restarted
... lines 198 - 204
```

So a collection of functionality is more than just tasks: it's tasks, variables, and sometimes other things like handlers. A role is a way to organize all of that into a pre-defined directory structure so that Ansible can automatically discover everything.

Creating the Role

Let's turn *all* of our Nginx setup into an Nginx role. In the ansible/ directory, create a new directory called roles... and inside that, another directory called nginx.

In a moment, we're going to point our playbook at this directory. When we do that, Ansible will automatically discover the tasks, variables, handlers and other things that live inside of it. This will work because roles *must* have a very specific structure.

Tasks in the Role

First, we know that a few tasks need to live in the role. So, create a directory called tasks and inside, a new file called main.yml. Start with the ---:

```
30 lines | ansible/roles/nginx/tasks/main.yml
---
... lines 2 - 30
```

Below - *just* like with symfony-bootstrap.yml - we add tasks. In the playbook, search for "Install Nginx web server". We need that! Move it into main.yml:

```
30 lines | ansible/roles/nginx/tasks/main.yml
---
- name: Install Nginx web server
  become: true
  apt:
    name: nginx
    state: latest
  notify: Restart Nginx
... lines 8 - 30
```

Let's copy a few others: like "Add Symfony config template to the Nginx available sites" and the two tasks below it. Move those to the role. Then, select everything and un-indent them:

30 lines | ansible/roles/nginx/tasks/main.yml

```yaml
---
- name: Install Nginx web server
  become: true
  apt:
    name: nginx
    state: latest
  notify: Restart Nginx

- name: Add Symfony config template to the Nginx available sites
  become: true
  template:
    src: templates/symfony.conf
    dest: "/etc/nginx/sites-available/{{ server_name }}.conf"
  notify: Restart Nginx

- name: Enable Symfony config template from Nginx available sites
  become: true
  file:
    src: "/etc/nginx/sites-available/{{ server_name }}.conf"
    dest: "/etc/nginx/sites-enabled/{{ server_name }}.conf"
    state: link
  notify: Restart Nginx

- name: Add enabled Nginx site to /etc/hosts
  become: true
  lineinfile:
    dest: /etc/hosts
    regexp: "{{ server_name }}"
    line: "127.0.0.1 {{ server_name }}"
```

Beautiful!

Role templates

Okay, what else does this role need? Well, this task refers to templates/symfony.conf... which lives at the root of the ansible/ directory:

30 lines | ansible/roles/nginx/tasks/main.yml

```yaml
---
... lines 2 - 8
- name: Add Symfony config template to the Nginx available sites
... line 10
  template:
    src: templates/symfony.conf
... lines 13 - 30
```

Drag the templates/ directory into the role.

Role Variables

The tasks also use a variable - server_name:

```
30 lines | ansible/roles/nginx/tasks/main.yml
---
... lines 2 - 8
- name: Add Symfony config template to the Nginx available sites
... line 10
  template:
... line 12
    dest: "/etc/nginx/sites-available/{{ server_name }}.conf"
... lines 14 - 15
- name: Enable Symfony config template from Nginx available sites
... line 17
  file:
    src: "/etc/nginx/sites-available/{{ server_name }}.conf"
    dest: "/etc/nginx/sites-enabled/{{ server_name }}.conf"
... lines 21 - 23
- name: Add enabled Nginx site to /etc/hosts
... line 25
  lineinfile:
... line 27
    regexp: "{{ server_name }}"
    line: "127.0.0.1 {{ server_name }}"
```

This is set at the top of our playbook, but it's only used by the Nginx tasks:

```
204 lines | ansible/playbook.yml
---
- hosts: vb

  vars:
    server_name: mootube.l
... lines 6 - 204
```

Let's move it into the role.

This time, create a vars/ directory and - once again - a main.yml file.

Remove the variable from playbook.yml and paste it here:

```
3 lines | ansible/roles/nginx/vars/main.yml
---
server_name: mootube.l
```

Notice that it's *not* under the vars keyword anymore: Ansible knows its a variable because it's in the vars/ directory.

Role Handlers

Finally, if you look back at the tasks, the last thing they reference is the "Restart Nginx" handler. Go find that at the bottom of our playbook:

```
204 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 191
  handlers:
    - name: Restart Nginx
      become: true
      service:
        name: nginx
        state: restarted
... lines 198 - 204
```

Copy it, remove it, then create - surprise! - a handlers directory with a main.yml file inside. Put the 3 dashes, paste it and un-indent!

```
6 lines | ansible/roles/nginx/handlers/main.yml
---
- name: Restart Nginx
  become: true
  service:
    name: nginx
    state: restarted
```

Using the Role

Phew! This is the file structure for a role, and as long as you follow it, Ansible will take care of including and processing everything. All *we* need to do is activate the role in our playbook. At the top, add roles, then - nginx:

```
171 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 18
  pre_tasks:
... lines 20 - 25
  roles:
    - nginx

  tasks:
... lines 30 - 171
```

That's it! Time to try it out. Run the *entire* playbook:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

Deploy to the dev environment this time - I'll show you why in a minute. Hey, it looks good! The Nginx stuff happens *immediately*.

Looking good... looking good... woh! The Nginx stuff was happy, but we have a huge error at the end. Go back to your browser and load the site with http://mootube.l/app_dev.php. This is that same error!

Don't be too Smart

What's going on? Well, I made our playbook too smart and it became self-aware. Ok, not quite - but we do have a mistake... and it's unrelated to roles. Inside symfony-bootstrap.yml, we only install composer dependencies when code_changed:

```
50 lines | ansible/includes/symfony-bootstrap.yml
---
- name: Install Composer's dependencies
... lines 3 - 7
  when: code_changed
... lines 9 - 50
```

Well, remember that the composer dependencies are a little different for the dev environment versus the prod environment. The last time I ran Ansible I used the prod environment. This time I used dev... but the task that should have installed the dependencies for the dev environment was skipped!

Shame on me! Find your balance between speed and being too clever. I'll comment out the when. Run the playbook again in the dev environment:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

This time "Install Composer dependencies" is marked as "changed" because it *did* download the dev dependencies. And the page works!

Role and Task Ordering

Go back to your terminal and scroll up a little. Ah, there's one slight problem: Nginx was installed *before* updating the apt repositories cache. That means that if there's a new version of Nginx, it might install the old one first. We didn't intend for them to run in this order!

In fact, only *one* thing ran before Nginx - other than the setup task - our pre-task! In the playbook, I've put pre_tasks first, then roles and then tasks:

```
171 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 18
  pre_tasks:
... lines 20 - 25
  roles:
... lines 27 - 28
  tasks:
... lines 30 - 171
```

And that's the order they run it. But it's not because of how I ordered them in my YAML file: Ansible always executes pre_tasks, then roles then tasks.

So how can we update the apt repository cache first? Just move those two tasks into pre_tasks:

```
171 lines | ansible/playbook.yml
---
- hosts: vb
... lines 3 - 18
  pre_tasks:
... lines 20 - 25
    - name: Update APT package manager repositories cache
      become: true
      apt:
        update_cache: yes

    - name: Upgrade installed packages
      become: true
      apt:
        upgrade: safe
... lines 35 - 171
```

Done! Next, let's download a third-party role for free playbook functionality!

Chapter 25: Using a 3rd Party to Install Redis

The *best* part about roles is they're *shareable*. There are a *ton* of third-party roles that you can download to give your playbook free stuff! I love free stuff!

Refresh the page in the "dev" environment. The page took over *2* seconds to load! Why? In DefaultController, our app is trying to use Redis:

111 lines | src/AppBundle/Controller/DefaultController.php

```
... lines 1 - 9
class DefaultController extends Controller
{
... lines 12 - 14
    public function indexAction()
    {
... lines 17 - 20
        // Redis cache
        try {
            if ($this->getRedisClient()->exists('total_video_uploads_count')) {
                $totalVideoUploadsCount = $this->getRedisClient()->get('total_video_uploads_count');
            } else {
                $totalVideoUploadsCount = $this->countTotalVideoUploads();
                $this->getRedisClient()->set('total_video_uploads_count', $totalVideoUploadsCount, 'ex', 60); // 60s
            }

            if ($this->getRedisClient()->exists('total_video_views_count')) {
                $totalVideoViewsCount = $this->getRedisClient()->get('total_video_views_count');
            } else {
                $totalVideoViewsCount = $this->countTotalVideoViews();
                $this->getRedisClient()->set('total_video_views_count', $totalVideoViewsCount, 'ex', 60); // 60s
            }
        } catch (ConnectionException $e) {
            $totalVideoUploadsCount = $this->countTotalVideoUploads();
            $totalVideoViewsCount = $this->countTotalVideoViews();
        }
... lines 40 - 46
    }
... lines 48 - 109
}
```

But, it's not installed! So, this fails... and just for a good example, our code rescues thing but sleeps for 2 seconds to "fake" the page being really slow:

```
111 lines | src/AppBundle/Controller/DefaultController.php
... lines 1 - 9
class DefaultController extends Controller
{
... lines 12 - 70
    /**
     * @return int
     */
    private function countTotalVideoUploads()
    {
        sleep(1); // simulating a long computation: waiting for 1s

        $fakedCount = intval(date('Hms') . rand(1, 9));

        return $fakedCount;
    }

    /**
     * @return int
     */
    private function countTotalVideoViews()
    {
        sleep(1); // simulating a long computation: waiting for 1s

        $fakedCount = intval(date('Hms') . rand(1, 9)) * 111;

        return $fakedCount;
    }
... lines 94 - 109
}
```

In other words, without Redis, our site is slow. But after we install it, the page should be super quick!

Installing the Redis Role

We already have all the skills needed to install Redis. But... maybe someone already did the work for us? Google for "Redis Ansible role". Hello DavidWittman/ansible-redis! This role helps install Redis... and it looks fairly active. And check it out: it looks like *our* role, with templates, vars, handlers and a few other things.

So... if we could download this into our project, we could activate the role and get free stuff! The way to do that is by using a command called ansible-galaxy. Copy it! Then, find your terminal and paste!

```
$ ansible-galaxy install DavidWittman.redis
```

In my case, it's already installed.

By default, ansible-galaxy downloads *roles* to a global directory. When you tell Ansible to load a role, it looks in your local directory but also looks in that global spot to find possible roles.

You could also download the role *locally* in your project. Add --help to the command:

```
$ ansible-galaxy install DavidWittman.redis --help
```

The -p option is the key! Downloading the role locally *might* be even better than downloading it globally. When it's in your project, you can commit it to your repository and manage its version.

Activate & Configure the Role

With the role downloaded, all we need to do is activate it! Easy! Copy the role name. Under our roles, I'll use a longer syntax: role: DavidWittman.redis then become: true:

173 lines | ansible/playbook.yml

```
---
- hosts: vb
... lines 3 - 35
  roles:
... line 37
    - role: DavidWittman.redis
      become: true
... lines 40 - 173
```

If you tried the role, you'd find out you need that. We didn't need it for the nginx role because we had the become: true lines internally.

Ok team, run the *entire* playbook in the dev environment:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini
```

While we're waiting for someone else to install Redis for us - thanks David - head back to the documentation. The main way to control how a role works is via variables, like redis_bind in this example:

```
---
- hosts: redis01.example.com
  vars:
    - redis_bind: 127.0.0.1
  roles:
    - DavidWittman.redis
```

It's cool how it works: internally, the role sets some variables and then uses them. We, of course, can override them. Simple, but powerful.

There *is* one downside to third-party roles: they can add some serious bloat to your playbook. Yea, it's running a *lot* of tasks. Often, a role is designed to work on multiple operating systems and versions. Some of these tasks are determining facts about the environment, like what Ubuntu version we have or what utilities are available. The role is really flexible, but takes extra time to run.

And, this first execution will take a *long* time: it's installing Redis!

Finally... it gets into our stuff.

Ding! Let's try it! Refresh MooTube. Then, refresh again. Yes! 29 milliseconds! Amazing! So much faster!

This works because our application is already configured to look for Redis on localhost. So as soon as it was installed, our app picked it up.

Next, let's talk more about configuration and how you might control things like the Redis host or database password.

Chapter 26: Variables and parameters.yml

How could we handle *sensitive* variables - like a database password? Well, committing them to our playbook is probably *not* a good idea. Nope, we need something better!

Organizing Vars into a File

First, let's reorganize a little bit! Create a new vars/ directory with a vars.yml file inside. Now, copy *all* of the variables, add the ---, paste them here, and - you know the drill - un-indent them:

6 lines | ansible/vars/vars.yml

```yaml
---
symfony_root_dir: /var/www/project
symfony_web_dir: "{{ symfony_root_dir }}/web"
symfony_var_dir: "{{ symfony_root_dir }}/var"
symfony_console_path: "{{ symfony_root_dir }}/bin/console"
```

Ansible gives us a way to import variables from a file... called vars_files. Point it to ./vars/vars.yml:

170 lines | ansible/playbook.yml

```yaml
---
- hosts: vb

  vars_files:
    - ./vars/vars.yml
... lines 6 - 170
```

Cool! Believe it or not, we're one step closer to being able to handle *sensitive* configuration.

Adding the secret Variable

In your VM move to /var/www/project:

```
$ cd /var/www/project
```

I want to look at the app/config/parameters.yml file:

```
$ cat app/config/parameters.yml
```

This file holds config for the Symfony project, like the database password. Notice one is called secret. This is *supposed* to be a unique string that's used for creating some random strings. Right now ours is... not so secret: that's the default value from Symfony.

Let's set this for real! In the vars.yml file, create a new variable: symfony_secret set to udderly secret $tring:

7 lines | ansible/vars/vars.yml

```yaml
---
... lines 2 - 5
symfony_secret: "udderly secret $tring"
```

Now, in symfony-bootstrap.yml, we can use that variable to modify parameters.yml. Create a new task: "Set Symfony secret in parameters.yml". Use our favorite lineinfile module with dest set to {{ symfony_root_dir }} - that's a variable from our vars file - {{ symfony_root_dir }}/app/config/parameters.yml:

```
58 lines | ansible/includes/symfony-bootstrap.yml
---
... lines 2 - 20
- name: Set Symfony secret in parameters.yml
  lineinfile:
    dest: "{{ symfony_root_dir }}/app/config/parameters.yml"
... lines 24 - 58
```

For regexp, use ^ secret:. Yep, we're looking for 4 spaces then secret:. For line, 4 spaces again then secret: {{ symfony_secret }}:

```
58 lines | ansible/includes/symfony-bootstrap.yml
---
... lines 2 - 20
- name: Set Symfony secret in parameters.yml
  lineinfile:
    dest: "{{ symfony_root_dir }}/app/config/parameters.yml"
    regexp: "^    secret:"
    line: "    secret: {{ symfony_secret }}"
  tags:
    - deploy
... lines 28 - 58
```

Don't forget to give this the deploy tag!

This *will* work... but don't even try it! Nope, we need to go further: having sensitive keys committed to my vars.yml file is *not* a good solution. We need the vault.

Chapter 27: The Variable Vault

The symfony_secret variable needs to be secret! I don't want to commit things like this to my repository in plain text!

Creating the Vault

One really cool solution to this is the *vault*: an *encrypted* variables file.

To create a vault file, go back to your main machine's terminal and run:

```
$ ansible-vault create ansible/vars/vault.yml
```

It'll ask you to create a vault password. We'll use - of course - beefpass. Keep this handy: it's needed to *decrypt* the vault.

Nice! It opens up an editor. Once you're inside, treat this like a normal variable file: add ---, then a new variable: vault_symfony_secret:

```
# ansible/vars/vault.yml
---
vault_symfony_secret:
```

I am purposely starting the variable name with vault_ - we'll talk about why in a minute.

Then, set the value to udderly secret $tring:

```
# ansible/vars/vault.yml
---
vault_symfony_secret: "udderly secret $tring"
```

Save with :wq Enter.

As *soon* as we do that, we have a vars/vault.yml file... and it is *not* human-readable: it's encrypted:

9 lines | ansible/vars/vault.yml

```
$ANSIBLE_VAULT;1.1;AES256
62313836626639653563363463316530346563316361316633613938626563373934363834336639
3735343165343739653866643763376431666239643435330a343234396138373965323537633333
61616261336138356436663266383533373166393837373561393365363130633864323262333531
6563633238636465640a643337316361393132633963303361653434613462353363373234336333
38636163653630343665363537663334643137626336633435336566316565623464346239343961
61366666636131373164316436396664646437353635343464653030626432373939353439353433
6335636634653066663036386639316666637
```

You can use the vault again to view it:

```
$ ansible-vault view ansible/vars/vault.yml
```

It, of course, needs your password: beefpass. Or, you can edit it:

```
$ ansible-vault edit ansible/vars/vault.yml
```

Importing the vault File

In vars.yml, we can now remove the secret string and use the variable instead: {{ vault_symfony_secret }}:

```
7 lines | ansible/vars/vars.yml
---
... lines 2 - 5
symfony_secret: "{{ vault_symfony_secret }}"
```

To make that variable available, import it like a normal variable file: vars/vault.yml:

```
171 lines | ansible/playbook.yml
---
- hosts: vb

  vars_files:
    - ./vars/vault.yml
    - ./vars/vars.yml
... lines 7 - 171
```

Make sure the vault is loaded first so we can use its variables inside vars.yml.

Let's walk through this: import vault.yml, then in vars.yml use the vault_symfony_secret variable to set symfony_secret. Then, *that* variable is used elsewhere.

Why the big dance? Why not just call the variable symfony_secret in the vault? Well, the vault_ prefix is nice because it's really easy to know that it came from the *vault*. That makes it easier to track things down.

Running a Vaulted Playbook

Let's try it! Run the playbook like before, with -t deploy and a new flag: --ask-vault-pass:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini -t deploy --ask-vault-pass
```

Nice! Enter beefpass for the password, use the prod environment, then let it run! Nice! The "Set Symfony secret in parameters.yml" task reported Changed!

In the VM, let's check out that file:

```
$ cat app/config/parameters.yml
```

```
# app/config/parameters.yml
# This file is auto-generated during the composer install
parameters:
    # ...
    secret: udderly secret $tring
    # ...
```

Woohoo! There's our ridiculous secret string.

Using a Secret Loggly Token

Let's do one more fun example. In your browser, remove the app_dev.php from the URL. It still works!

Now, open app/config/parameters.yml. The last key is called loggly_token:

```
# app/config/parameters.yml
# This file is auto-generated during the composer install
parameters:
    # ...
    loggly_token: 12345
    # ...
```

This is used in config_prod.yml:

```
26 lines | app/config/config_prod.yml
... lines 1 - 9
monolog:
    handlers:
... lines 12 - 21
        loggly:
... line 23
            token: '%loggly_token%'
... lines 25 - 26
```

Basically, in the prod environment, the system is already setup to send *all* logs to Loggly: a cloud log collector. Right now... there's not too much in my account.

The only thing *we* need to do to get this to work is replace this line in parameters.yml with a working Loggly token. That's perfect for the vault!

Adding more to the Vault

Edit the vault!

```
$ ansible-vault edit ansible/vars/vault.yml
```

Add a new variable - vault_loggly_token:

```
---
# ...
vault_loggly_token:
```

I'll paste a real token for my account and save:

```
---
# ...
vault_loggly_token: fb4aa5b2-30a3-4bd8-8902-1aba5a683d62
```

And as much fun as it would be for me to to see all of *your* logs, you'll need to create your own token - I'll revoke this one after recording.

We know the next step: open vars.yml and set loggly_token to vault_loggly_token:

```
8 lines | ansible/vars/vars.yml
---
... lines 2 - 6
loggly_token: "{{ vault_loggly_token }}"
```

And finally, we can use loggly_token in symfony-bootstrap.yml. Copy the previous lineinfile task, paste it, and rename it to "Set Loggly token in parameters.yml":

```
66 lines | ansible/includes/symfony-bootstrap.yml
    ... lines 1 - 20
- name: Set Symfony secret in parameters.yml
    ... lines 22 - 28
- name: Set Loggly token in parameters.yml
    ... lines 30 - 66
```

Update the keys to loggly_token, and the variable to loggly_token:

```
66 lines | ansible/includes/symfony-bootstrap.yml
    ... lines 1 - 28
- name: Set Loggly token in parameters.yml
  lineinfile:
    dest: "{{ symfony_root_dir }}/app/config/parameters.yml"
    regexp: "^    loggly_token:"
    line: "    loggly_token: {{ loggly_token }}"
  tags:
    - deploy
    ... lines 36 - 66
```

Ok, try it!

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini -t deploy --ask-vault-pass
```

Fill in beefpass and use the prod environment again.

Ding! Head to your VM and look at the parameters.yml file:

```
$ cat app/config/parameters.yml
```

Nice! So... it should work, right? Refresh Mootube to fill in some logs. Now, reload the Loggly dashboard. Hmm... nothing! What's going on?

Actually, we have another mistake in our playbook. Once again, I got too smart! We *only* clear the cache when the code has changed:

```
66 lines | ansible/includes/symfony-bootstrap.yml
---
    ... lines 2 - 59
- name: Clear cache
    ... lines 61 - 64
  when: code_changed
```

But in this situation, parameters.yml changed... which is not technically part of our code. In other words, this isn't working because we have not *yet* cleared our prod cache.

I'll comment out the when under "Cache Clear" to make it *always* clear. Then, just for right now, in the VM, clear the cache manually:

```
$ ./bin/console cache:clear --env=prod
```

If you get an error about permissions, that's fine: it's just having problems clearing out the old cache directory. You can delete that:

```
$ sudo rm -rf var/cache/pro~
```

Let's see if that was the problem! Refresh MooTube. Then, try the Loggly dashboard. Got it! So cool! If you're coding with me, it might take a few minutes to show up, so be patient!

Our playbook is really powerful. Could we use it to deploy to a cloud server like AWS? Why not!? Let's do it!

Chapter 28: Deploy to AWS!

So far, we've been deploying to a virtual machine. But... there's nothing stopping us from deploying to... the CLOUD! Let's try it - let's deploy to Amazon EC2. This is not an exhaustive tutorial about using EC2... but let's at least get our feet wet and see if we can get into some trouble!

Tip

Want to properly deploy with Ansible? Check out Ansistrano.

Manually Launching an EC2 Instance

I'm already on my EC2 dashboard. In a few minutes, we're going to use Ansible to actually *launch* a new instance. But for now, just hit "Launch Instance" to do it by hand. I'm looking for an image that's similar to what we're using with Vagrant: Ubuntu 14.04. Select that image, use the micro instance size, and just use the default settings on the next screens.

Tip

The instance id we used is ami-41d48e24 if you need to find it manually.

For the security group, I'm going to select a group I already created: "Web Access Testing." The important thing is to allow port 22 for SSH and ports 80 and 443 for web stuff. Hit "Review and Launch", then "Launch" that instance!

Bah! What a tease! No instance yet: we need to choose a key pair for SSH. I already created a pair for this tutorial called Ansible_AWS_tmp. When we launch the instance, instead of logging in with a username and password, we will SSH with a username and a private key. You'll need to create your own key pair. When you do that, you'll download its private key. In this case, the file is called Ansible_AWS_tmp.pem and I already downloaded it.

Ok, *now* launch the instance! Cool! Click to view its progress.

Configuring the new Host

While it's loading, let's get to work!

This new server represents a new *host*. In hosts.ini we have a local group with one server and a vb group with one server. Create a new group called aws:

13 lines | ansible/hosts.ini

```
... lines 1 - 6
[aws]
... lines 8 - 13
```

Below, we need the IP to the server. Wait for it to boot.

When it's ready, copy its public IP address, go back to the hosts file, and paste! This time, set ansible_user to ubuntu: that's the user that's setup for this image. And instead of a password, use ansible_ssh_private_key_file= and put the path to your downloaded private key: ~/.ssh/Ansible_AWS_tmp.pem for me:

13 lines | ansible/hosts.ini

```
... lines 1 - 6
[aws]
54.205.128.194 ansible_user=ubuntu ansible_ssh_private_key_file=~/.ssh/KnpU-Tutorial.pem
... lines 9 - 13
```

Tip

Depends on your AWS instance, you may need to specify a new path to Python interpreter. By default, Ansible uses

/usr/bin/python but new AWS instances have Python 3 pre-installed and the path to it is /usr/bin/python3. You can specify the correct Python interpreter path explicitly with ansible_python_interpreter key in case you got an error from Ansible about not found Python:

```
# ansible/hosts.ini

# ...

[aws]
54.205.128.194 ansible_python_interpreter=/usr/bin/python3 ansible_user=ubuntu ansible_ssh_private_key_file=~/.ssh/KnpU-Tutorial.pem
```

Host Group Children

Here's where things get cool! I want to run our playbook against the virtual machine *and* my EC2 instance. Because... it's totally valid to build two servers at once! That's where Ansible shines!

Right now, each lives under its own host group - vb and aws:

13 lines | ansible/hosts.ini

```
... lines 1 - 3
[vb]
192.168.33.10 ansible_user=vagrant ansible_ssh_pass=vagrant

[aws]
54.205.128.194 ansible_user=ubuntu ansible_ssh_private_key_file=~/.ssh/KnpU-Tutorial.pem
... lines 9 - 13
```

Inside of our playbook, we've configured the play to only run against the vb group:

171 lines | ansible/playbook.yml

```
---
- hosts: vb
... lines 3 - 171
```

How could we run that against the hosts in the vb group *and* in the aws group?

With a host group... group! Check it out: create a new group called webserver, but add a :children after. That special children syntax allows us to list other host *groups* below this: vb and aws:

13 lines | ansible/hosts.ini

```
... lines 1 - 3
[vb]
192.168.33.10 ansible_user=vagrant ansible_ssh_pass=vagrant

[aws]
54.205.128.194 ansible_user=ubuntu ansible_ssh_private_key_file=~/.ssh/KnpU-Tutorial.pem

[webserver:children]
vb
aws
```

Yep, we now have a new group - webserver - that's a combination of these two.

Back in the playbook, change vb to webserver:

171 lines | ansible/playbook.yml

```
---
- hosts: webserver
... lines 3 - 171
```

Running the Playbook

Deep breath. Run the playbook:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini --ask-vault-pass
```

Enter beefpass and deploy to the prod environment. You'll need to verify the authenticity of the new host this first time. And, by the way, you *can* disable this check in Ansible.

Now, watch the magic! You'll start to see it execute each task against *both* servers. The first time we do this, it'll take awhile: the new EC2 server is being setup from scratch. And, I was cheap - it's only a *micro* instance.

While we're waiting, let's go copy the IP address to the new server again. Temporarily open a new terminal tab and edit the /etc/hosts file:

```
$ sudo vim /etc/hosts
```

To test thing, update mootube.l to point to the IP address of the EC2 instance:

```
# /etc/hosts
# ...
#192.168.33.10 mootube.l
54.205.128.194 mootube.l
```

Then, save, quit and close the tab.

Even though Ansible is still working, if I go to http://mootube.l right now, I see the "Welcome to Nginx" page. Ha, cool! Ansible is already *part* way through the process!

Let's *try* to be patient... but also fast forward!

Done! And beautiful - it finished with no errors. That's kind of amazing: we launched a new cloud server from scratch... with no changes. Refresh the page. Got it! Welcome to MooTube, hosted on our fancy new EC2 instance. Notice that there's no data because we loaded in the prod environment: so the fixtures didn't run.

The only weird thing is that after changing my hosts file, I can't access MooTube on my VM anymore. But, we can solve that with host group vars.

Chapter 29: Host Group Vars

Right now, both my EC2 machine *and* my virtual machine are configured to respond to mootube.l. And that means I can only access one at a time: I can setup my /etc/hosts to make mootube.l point to my EC2 instance *or* my VM... but not both.

What if instead, we setup the VM to be mootube.l and the EC2 instance to be mootube.ec2? That would fix it! How can we do that?

The problem is that in our roles/nginx/vars/main.yml file, we have a server_name variable... but it's just hardcoded to mootube.l:

3 lines | ansible/roles/nginx/vars/main.yml

```
---
server_name: mootube.l
```

I want to override that variable to a different value for each host group. And that is totally possible.

But first, re-open /etc/hosts and point mootube.l back to the virtual machine IP. Then, add a new mootube.ec2 entry that points to the EC2 instance:

```
$ sudo vim /etc/hosts
```

```
# /etc/hosts
# ...
192.168.33.10 mootube.l
54.205.128.194 mootube.ec2
```

Nice!

Setting Variables for a Host Group

Now, how can we override the server_name variable *only* for the aws host group? Create a new directory called group_vars with a file inside: aws.yml. *Just* by having this exact directory and filename, whenever the aws group is executed, it will automatically load this file and use the variables inside. But those variables will *only* apply to the aws group.

Inside, create a new host_server_name variable set to mootube.ec2:

3 lines | ansible/group_vars/aws.yml

```
---
host_server_name: ec2-54-205-128-194.compute-1.amazonaws.com
```

Copy that variable name. Next, open roles/nginx/vars/main.yml, replace the hardcoded mootube.l with something fancier: {{ host_server_name|default('mootube.l') }}:

3 lines | ansible/roles/nginx/vars/main.yml

```
---
server_name: "{{ host_server_name|default('mootube.l') }}"
```

This says: use host_server_name if it exists. But if it doesn't, default to mootube.l. This should give us a unique host_name variable for each group.

We're ready: try the playbook:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini --ask-vault-pass
```

Nice - we can see a few changes, but only to the EC2 server, as it changes the host name. Ding!

Go back to your browser and refresh mootube.l. This is coming from the VM: I know because it has data! Now try http://mootube.ec2. Boom! This comes from EC2. Super fun.

We just used Ansible to provision an entirely new server on EC2. Could we even use it to *launch* the server programmatically? Nope! I'm kidding - totally - let's do it!

Chapter 30: Launch a Cloud Instance!

Alas, this is our *final* chapter. So, I want to do something fun, and also talk about how Ansible can be pushed further.

First, our Ansible setup could be a lot more powerful. We already learned that instead of having one big play in our playbook, we could have *multiple* plays. One play might setup the web servers, another play might provision the database servers, and one final play could configure all of the Redis instances. We're using one play because - in our smaller setup - all of that lives on the same host.

Also, each host group can have *multiple* servers below it. We could launch 10 EC2 instances and provision all of them at once.

And finally, Ansible can even be used to *launch* the instances themselves! A few minutes ago, we manually launched the EC2 instance through the web interface. Lame! Let's teach Ansible to do that.

Launching EC2 Instances?

How? A module of course! The ec2 module. This module is really good at interacting with EC2 instances. Actually, if you click the Cloud Modules section on the left, you'll find a *ton* of modules for dealing with EC2 and many other services, like IAM, RDS and S3. And of course, modules exist for all of the major cloud providers. Ansible rocks!

So far, our playbook has been executing commands on the remote hosts - like our virtual machine. But, in this case... we don't need to do that. Yea, we can run the ec2 module *locally*... because the purpose of this module is to talk to the AWS API. In other words, it doesn't matter what host we execute it from!

Wherever you decide to execute these tasks, you need to make sure that something called Boto is installed. It's an extension for Python... which you might also need to install locally. So far, Python has already come pre-installed on our VM and EC2 instances.

If you're not sure if you have this Boto thing, just try it. If you get an error about Boto, check into installing it.

Creating the Playbook

Since these new tasks will run against a new host - localhost - we can organize them as a new *play* in our playbook... or create a new playbook file entirely. To keep things simple, I'll create a new playbook file - aws.yml.

Inside, you know the drill: start with the host, set to local. Below that, set gather_facts to false:

31 lines | ansible/aws.yml

```
---
- hosts: local
  gather_facts: False
... lines 4 - 31
```

What's that? Each time we run the playbook, the first task is called "Setup". That task gathers information about the host and creates some "facts"... which is cool, because we can use those facts in our tasks.

But since we're simply running against our local machine, we're not going to need these facts. This saves time.

EC2 Auth: AWS Secret Key

For the EC2 module to work, we need an AWS access key and secret key. You can find these inside of the IAM section of AWS under "Users". I already have mine prepared. Let's use them!

But wait! We *probably* don't want to hardcode the secret key directly in our playbook. Nope, let's use the vault!

```
$ ansible-vault edit ansible/vars/vault.yml
```

Type in beefpass. Then, I'll paste in 2 new variables: vault_aws_access_key and vault_aws_secret_key:

```
# ansible/vars/vault.yml
---
# ...
vault_aws_access_key: "AKIAJAWKEZQ6S7LM3EKQ"
vault_aws_secret_key: "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"
```

Save and quit!

Just like before, open vars.yml and create two new variables: aws_access_key set to vault_aws_access_key and aws_secret_key set to vault_aws_secret_key:

10 lines | ansible/vars/vars.yml

```
---
... lines 2 - 7
aws_access_key: "{{ vault_aws_access_key }}"
aws_secret_key: "{{ vault_aws_secret_key }}"
```

Finally, open up playbook.yml so we can steal the vars_files section. Paste that into the new playbook:

31 lines | ansible/aws.yml

```
---
- hosts: local
  gather_facts: False

  vars_files:
    - ./vars/vault.yml
    - ./vars/vars.yml
... lines 8 - 31
```

To use the keys, you have two options: pass them directly as options to the ec2 module, or set them as environment variables: AWS_ACCESS_KEY and AWS_SECRET_KEY. In fact, if those environment variables are already setup on your system, you don't need to do anything! The module will just pick them up!

Let's *set* the environment variables... because it's a bit more interesting. Just like before, use the environment key. Then set AWS_ACCESS_KEY to {{ aws_access_key }}. Repeat for AWS_SECRET_KEY set to {{ aws_secret_key }}:

31 lines | ansible/aws.yml

```
---
- hosts: local
... lines 3 - 8
  environment:
    AWS_ACCESS_KEY: "{{ aws_access_key }}"
    AWS_SECRET_ACCESS_KEY: "{{ aws_secret_key }}"
    # or use aws_access_key/aws_secret_key parameters on ec2 module instead
... lines 13 - 31
```

Boom! We are ready to start *crushing* it with this module... or any of those AWS modules.

Using the ec2 Module

And actually, using the module is pretty simple! We're just going to give it a lot of info about the image we want, the security group to use, the region and so on.

Add a new task called "Create an Instance". Use the ec2 module and start filling in those details:

31 lines | ansible/aws.yml

```yaml
---
- hosts: local
... lines 3 - 13
  tasks:
    - name: Create an instance
      ec2:
... lines 17 - 31
```

For instance_type, use t2.micro and set image to ami-41d48e24:

31 lines | ansible/aws.yml

```yaml
---
- hosts: local
... lines 3 - 13
  tasks:
    - name: Create an instance
      ec2:
        instance_type: t1.micro
        image: ami-a15e0db6
... lines 19 - 31
```

That's the exact image we used when we launched the instance manually.

Next, set wait to yes - that's not important for us, but it tells Ansible to wait until the machine gets into a "booted" state. If you're going to do more setup afterwards, you'll need this:

31 lines | ansible/aws.yml

```yaml
---
- hosts: local
... lines 3 - 13
  tasks:
    - name: Create an instance
      ec2:
        instance_type: t1.micro
        image: ami-a15e0db6
        wait: yes
... lines 20 - 31
```

Then, group: web_access_testing, count: 1, key_name: Ansible_AWS_tmp, region: us-east-2 and instance_tags with Name: MooTube instance:

31 lines | ansible/aws.yml

```
---
- hosts: local
... lines 3 - 13
  tasks:
    - name: Create an instance
      ec2:
        instance_type: t1.micro
        image: ami-a15e0db6
        wait: yes
        group: web_access
        count: 1
        key_name: KnpU-Tutorial
        region: us-east-1
        instance_tags:
          Name: MooTube instance
... lines 26 - 31
```

Obviously, tweak whatever you need!

Just like any other module, we can register the output to a variable. I wonder what that looks like in this case? Add register: ec2 to find out:

31 lines | ansible/aws.yml

```
---
- hosts: local
... lines 3 - 13
  tasks:
    - name: Create an instance
      ec2:
... lines 17 - 25
      # Could be useful further to get Public IP, DNS, etc.
      register: ec2
... lines 28 - 31
```

Then, debug it: debug: var=ec2:

31 lines | ansible/aws.yml

```
---
- hosts: local
... lines 3 - 13
  tasks:
    - name: Create an instance
      ec2:
... lines 17 - 25
      # Could be useful further to get Public IP, DNS, etc.
      register: ec2

    # debug the output to see what AWS returns back
    - debug: var=ec2
```

Give it a try!

```
$ ansible-playbook ansible/aws.yml -i ansible/hosts.ini --ask-vault-pass
```

Cool, it skipped the setup task and went straight to work! If you get an error about Boto - either it doesn't exist, or it can't find the region - you may need to install or upgrade it. I *did* have to upgrade mine - I could use the us-east-1 region, but not us-east-2. Weird, right? Upgrading for me meant running:

```
$ easy_install -U boto
```

And, done! Yes! It's green! And the variable is *awesome*: it gives us an instance id and a lot of other great info, like the public IP address. If I refresh my EC2 console, and remove the search filter... yes! Two instances running.

I can feel the power!

Boot and then Provision?

We now have 2 playbooks: one for booting the instances, and another for provisioning them. If you wanted Ansible to boot the instances and then provision them, that's totally possible! Ultimately, we could take this public IP address and add it as a new host under the aws group:

```
13 lines | ansible/hosts.ini
... lines 1 - 6
[aws]
54.205.128.194 ansible_user=ubuntu ansible_ssh_private_key_file=~/.ssh/KnpU-Tutorial.pem
... lines 9 - 13
```

Of course... with our current setup, the hosts.ini inventory file is static: each time we launch a new instance, we would need to manually put its IP address here:

```
13 lines | ansible/hosts.ini
... lines 1 - 6
[aws]
54.205.128.194 ansible_user=ubuntu ansible_ssh_private_key_file=~/.ssh/KnpU-Tutorial.pem
... lines 9 - 13
```

But, there *are* ways to have a *dynamic* hosts file. Imagine a setup where Ansible automatically looks at the servers booted in the cloud and uses *them* for your inventory. That's beyond the scope of this tutorial, but if you need that, go for it!

Woh, we're done! Thanks for sticking with me to cover this huge, but *super* powerful tool! When you finally figure out how to get Ansible to do your laundry for you, send me your playbook. Or better, create a re-usable role and share it with the world.

All right guys, seeya next time.