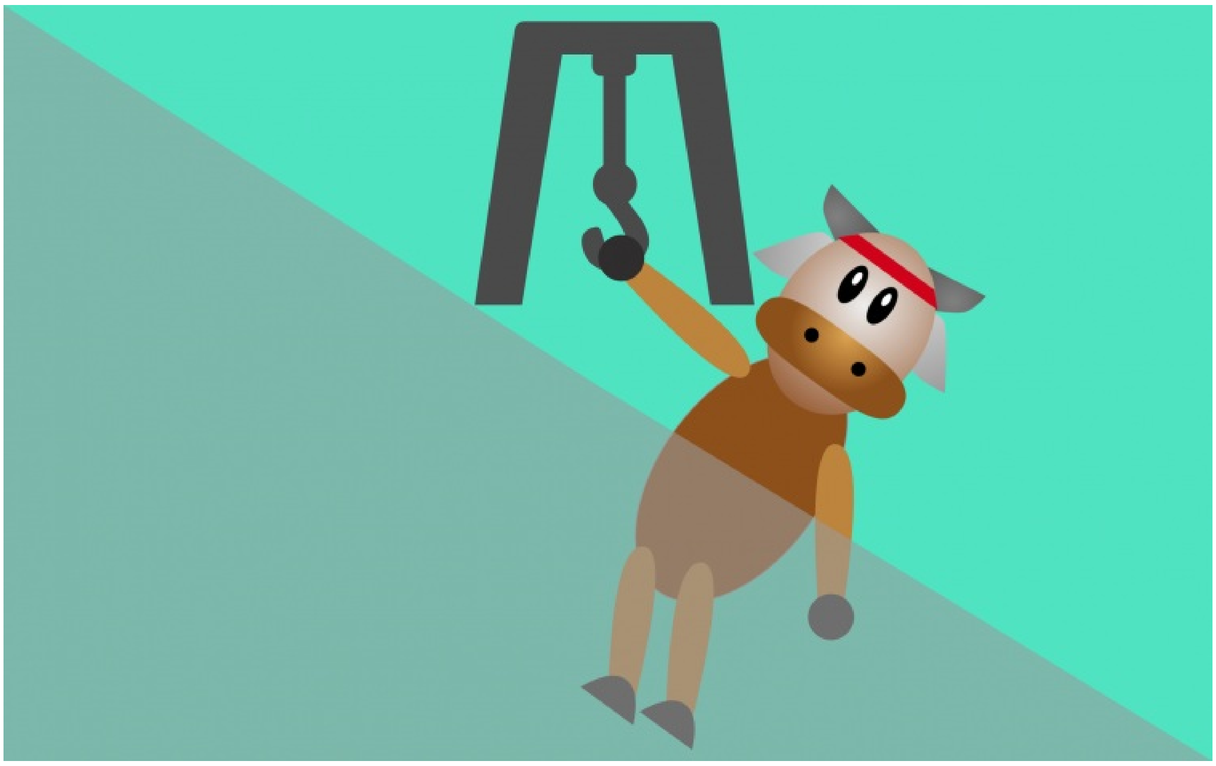
Animated Deployment with Ansistrano



With <3 from SymfonyCasts

Chapter 1: Setup: Server Provisioning

Hey guys! Ok, here's my situation: I've built this *amazing* new app: MooTube: the latest *fad* in cow fitness. There's just one more problem to solve, before cattle start signing up in herds: the site *only* lives on my local computer! It's time to release it to the cow masses. Yep, it's time to deploy!

But... how? There are probably 50 good ways to deploy! Bah! And these days, you can even deploy with a Platform as a Service: something like Platform.sh or Heroku. These take care of almost *everything* for you. I love these, and we use Platform.sh for part of KnpUniversity. They *do* have some limitations, but they are the *fastest* way to get your app to production.

In this tutorial, we're going to talk about one, *really* nice deployment tool: Ansistrano. It's built on top of Ansible... so if you watched our Ansible tutorial, you're going to *love* it! And if you haven't, what are you waiting for!? Well actually, I'll give you all the details you need, regardless.

Download the Project

As always, learning, like grazing, is best done in a group: so you should *definitely* code along with me. Download the course code from this page and unzip it. Inside, you'll find a start/ directory, which will have the same code I have here. See that README file? It holds all the secrets for getting the project setup. But actually... this is a tutorial about deployment! So... you don't *really* need to get the project running *locally*... because we're going to get the project running... in the **cloud**!

But, if you *do* want to get the project running, the last step will be to find your terminal, sip some coffee, and run:

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

to start the built-in PHP web server. Open the app in your browser at http://localhost:8000. Ah yes, MooTube: our bovine fitness app that is about to stampede through the cow world! This is the *same* app we used in our Ansible tutorial, with just a few small changes.

Booting a new Server

Let's get to work! So first... well... we need a server! You can use any service to get a server, but I already booted a new EC2 instance from AWS. I actually did this via Ansible. In our Ansible tutorial, we created a small playbook - aws.yml - whose *only* job is to boot an EC2 instance using an Ubuntu 14.04 image.

You're free to get a server from *anywhere*... but if you *do* want to use this script to boot a new instance, you'll just need to do 2 things. First, edit the ansible vault at ansible/vars/aws_vault.yml.

```
$ ansible-vault edit ansible/vars/aws_vault.yml
```

The password is beefpass.

These access keys are mine... and as much fun as it would be for me to pay for your servers... these keys won't work anymore. Sorry! Replace them with your own. Second, in aws.yml, see that key_name?

31 lines | ansible/aws.yml

```yaml
---
- hosts: local
... lines 3 - 13
  tasks:
    - name: Create an instance
      ec2:
... lines 17 - 21
        key_name: KnpU-Tutorial
... lines 23 - 31
```

You'll need to create your own "Key Pair" in the EC2 management console, and put its name here. The key pair will give you the private key needed to SSH onto the new server.

Server Provisioning

Once you have a server... deploying is really *two* steps. Step 1: provisioning: the fancy word that basically means installing everything you need, like Nginx, PHP, PHP extensions and whatever else. And *then* step 2: actually deploying.

I don't care *how* you setup - or *provision* - your server. In the Ansible tutorial, we - of course! - used Ansible to do this, but that is *not* a requirement for using Ansistrano.

But since we already have a working provision playbook, let's use it! First, I'll find the public IP address to my new server. Open ansible/hosts.ini and put this under the aws group:

13 lines | ansible/hosts.ini

```ini
... lines 1 - 6
[aws]
54.205.128.194
... lines 9 - 13
```

If you're still new to Ansible, we'll talk more about this file once we start to deploy.

Now, run Ansible:

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

Go Ansible go! See that -l aws at the end? Well, the provision playbook - playbook.yml - is setup to provision both my aws hosts and also a local VirtualBox machine. The -l tells Ansible to only provisioning the AWS server right now.

Ansible Authentication

Behind the scenes, Ansible is SSH'ing onto the server and running commands. But... how does authentication to the server work? In our case, it's with the private key from the "Key Pair" that we used to boot the server. Open ansible/group_vars/aws.yml:

6 lines | ansible/group_vars/aws.yml

```yaml
---
ansible_user: ubuntu
ansible_ssh_private_key_file: ~/.ssh/KnpU-Tutorial.pem
ansible_python_interpreter: /usr/bin/python3
host_server_name: mootube.example.com
```

Because we're using the aws group, this file is automatically loaded. It sets two important variables: ansible_user and ansible_ssh_private_key_file.

When you use Ansistrano to deploy, you'll need to make sure these two variables are set... or ansible_ssh_pass if you're using a password. You don't need to set them in a fancy group variable file like this. If you're still new to Ansible, I'll show you how to create variables right in your deploy playbook later.

For now, just know that we *are* telling Ansible how to authenticate: there's no magic.

Checking out the Server

Since Ansible is still working, open a third tab. Let's SSH onto the server: ssh -i, the path to the private key, then ubuntu@ followed up the IP address:

```
$ ssh -i ~/.ssh/KnpU-Tutorial.pem ubuntu@54.XXX.XX.XXX
```

Perfect! Now... pour a fresh cup of coffee and learn a foreign language while we wait for provisioning to finish. Fast forward!!!!!

Ding! Our server is ready! Check this out! We have PHP 7.1, Nginx and even an Nginx Virtual host.

```
$ cd /etc/nginx/sites-available
$ sudo vim mootube.example.com.conf
```

Our site will be mootube.example.com, and this is setup with a document root at /var/www/project/web. Right now, there *is* a /var/www/project directory... but it's empty. Putting code there? Yea, that's the job of this tutorial.

Let's go!

Chapter 2: Ansistrano Role Installation

We already have an ansible/ directory, which has a bunch of files to support two playbooks: aws.yml that boots new EC2 servers and playbook.yml that can *provision* those servers, installing things like Nginx, PHP, and anything else we need. Now, we're going to create a *third* playbook: deploy.yml that will deploy our code.

But! There's one *really* important thing I want you to understand: this new playbook will not use *any* of the files inside of the ansible/ directory. So, don't worry or think about them: pretend that the ansible/ directory is *completely* empty, except for deploy.yml. If you *do* need any other files, we will talk about them!

To help us deploy with Ansible, we're going to - of course - use Ansistrano! Open up ansistrano.com in your browser. It has some cool deployment stats... but the most important thing is the ansistrano.deploy link that goes to the GitHub page and their docs.

Ansistrano is an Ansible role... which basically means it gives us free Ansible tasks! That's like getting a free puppy... but without all that responsibility and carpet peeing!

Installing the Role

The docs show an ansible-galaxy command that will install the role. Don't do it! There's a better way!

Open ansible/requirements.yml:

3 lines | ansible/requirements.yml

```yaml
- src: DavidWittman.redis
  version: 1.2.4
```

You *can* use ansible-galaxy to install whatever random Ansible role you want. *Or*, you can describe the roles you need in a YAML file and tell galaxy to install everything you need at once. This is just a nicer way to keep track of what roles we need.

Add another src: line. Then, go copy the role name - just the deploy role:

6 lines | ansible/requirements.yml

```yaml
... lines 1 - 3
- src: ansistrano.deploy
... lines 5 - 6
```

Tip

Due to the changes in Ansible Galaxy, Ansistrano is installed now via ansistrano.deploy instead of the old carlosbuenosvinos.ansistrano-deploy.

We'll talk about rollback later. Paste that and add version. So... what's the latest version of this role? Let's find out! On the GitHub page, scroll up and click "Releases". But be careful! There are actually newer tags. Ok, so right now, the latest version is 2.7.0. Add that to requirements.yml:

6 lines | ansible/requirements.yml

```yaml
... lines 1 - 3
- src: ansistrano.deploy
  version: 2.7.0
```

Great! To make sure all of the roles are installed, run:

```
$ ansible-galaxy install -r ansible/requirements.yml
```

We *already* have the Redis role installed that's used in the provision playbook. And now it downloads ansistrano-deploy to some /usr/local/etc directory. Perfect!

Configuring the Hosts

In our deploy.yml, start with the meaningless, but ceremonial three dashes. Then, below that, add hosts set to aws:

6 lines | ansible/deploy.yml

```
---
- hosts: aws
... lines 3 - 6
```

This is important: if you're *only* using Ansible for deployment, then you don't need *any* of these other files in the ansible/ directory... except for hosts.ini. You *do* need this file. It doesn't need to be as complex as mine. You just need to have one host group with at least one IP address below it:

13 lines | ansible/hosts.ini

```
... lines 1 - 6
[aws]
54.205.128.194
... lines 9 - 13
```

In our case, we have a host group called aws with the IP address to one server below it.

Using the Role

Back in deploy.yml, let's import the role! Add roles:, copy the name of the role, and then paste it here: ansistrano.deploy:

6 lines | ansible/deploy.yml

```
---
- hosts: aws

  roles:
    - ansistrano.deploy
```

If you went through our Ansible tutorial, then you know that a role magically gives our playbook new tasks! Plus, a few other things, like variables and handlers.

So... what new tasks did this add? Let's find out! Run:

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

Thanks to the --list-tasks flag, this won't *execute* our playbook, it will just tell us what tasks it *would* run. Try it!

Not all of this will make sense yet... but you can see things like "Ensure deployment base path exists". And later, it creates something called a "current folder" and performs some cleanup.

What does this all mean? It's time to learn *exactly* how Ansistrano works and run our first deploy. That's next!

Chapter 3: Anatomy of an Ansistrano Deploy

Go back to the Ansistrano GitHub page. Find the table of contents near the top, and click Deploying. Excellent! This tells us how to use this role *and* how it works! Ansistrano is based off of Capistrano... which means it creates a really cool directory structure on your server. In this example, we're deploying to a /var/www/my-app.com directory on the server. Each time we deploy, it creates a new, timestamped, directory inside releases/ with our code. Then, when the deploy finishes, it creates a symlink from a current/ directory to this release directory.

Next time you deploy? Yep, the same thing happens: it will create a *new* release directory and update the symbolic link to point to it instead. This means that our web server should use the current/ directory as its document root.

This is *amazing*. Why? With this setup, we can patiently do *all* the steps needed to prepare the new release directory. *No* traffic hits this directory until the very end of deployment, when the symbolic link is changed.

There's also a shared/ directory, which allows you to share some files between releases. We'll talk more about that later.

Set the Deploy Directory then Deploy!

To start with Ansistrano... well... the *only* thing we need to do is tell it *where* to deploy to on the server! How? The same way you control any role: by overriding *variables* that it exposes.

Scroll up a little on their docs to find a *giant* box of variables. Yes! This tells you *every* single possible variable that you can override to control how Ansistrano works. This is documentation gold!

The first variable we need ansistrano_deploy_to. Copy that. Inside deploy.yml, add a vars key and paste. Set this to the directory that's already waiting on our server: /var/www/project:

10 lines | ansible/deploy.yml

```
---
- hosts: aws

  vars:
    # Ansistrano vars
    ansistrano_deploy_to: "/var/www/project" # Base path to deploy to.
... lines 7 - 10
```

Ok... well... we haven't done much... but let's see if it works! In your local terminal, run the same command as before, but without the --list-tasks flag:

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

Ok... it looks like it's working. A few of the tasks mention rsync. That's because, by default, Ansistrano uses rsync to get the files from your local machine up to your server. We'll change to a different strategy in a few minutes.

Ding! It finished! Let's go see what it did! Change to the terminal where you're SSH'ed onto your server. Inside /var/www/project, run ls.

```
$ ls
```

Awesome! We have the Ansistrano directory structure: current/, releases/ and shared/. So far, we only have one directory in releases/ and current/ is a symlink to it.

Now, move into the current/ directory and look inside:

```
$ cd /var/www/project/current
$ ls
```

Woh! There's almost nothing here: just a REVISION file that Ansistrano created and an ansible/ directory... which is a copy of our local ansible/ directory.

This looks weird... but it makes sense! Right now, Ansistrano is using rsync to deploy *only* the directory where the playbook lives... so, ansible/. This is not what we want. So next, let's change our deployment strategy to git so that Ansistrano pulls down our *entire* repository.

Chapter 4: Deploy with git

Our *first* deployment task is simple: we need to get our code to the server! By default, Ansistrano does that via rsync... but it has a *bunch* of options! Check out the ansistrano_deploy_via variable. Beyond rsync, you can use copy, git, svn, s3 - if you want to fetch your code from an S3 bucket - and download. We're going to use the git strategy! And most of the rest of the variables in the docs are specific to your deploy strategy.

Setting up the git Repo

Ok, so we're going to deploy via git... which means... well, we should probably create a git repository! First, I'll commit everything locally. You may even need to initialize your git repository with git init. I already have a repo, so I'll just commit:

```
$ git add .
$ git commit -m "I'm king of the world!"
```

Perfect! Next, we need to host our Git repository somewhere. It doesn't matter where, but I'll use GitHub. Create a brand new repository! Woo! I'm making this *public*, but we *will* talk soon about how to deploy *private* repositories.

Copy the 2 lines to add the remote and push our code. Then, in your terminal, paste them:

```
$ git remote add origin git@github.com:weaverryan/ansistrano-deploy.git
$ git push -u origin master
```

Progress!

Back on GitHub, refresh! There is our beautiful code!

Configuring the Deploy

Back in Ansistrano land, the first thing we need to do is configure that ansistrano_deploy_via variable. Set it to git:

15 lines | ansible/deploy.yml

```yaml
---
- hosts: aws

  vars:
... lines 5 - 6
    ansistrano_deploy_via: git # Method used to deliver the code to the server. Options are copy, rsync, git, svn, s3 or download
... lines 8 - 15
```

For the Git-specific variables, we need to configure two: the URL to the repo and what *branch* to deploy. Copy ansistrano_git_repo first and paste it. For the URL, go back to GitHub and click on "Clone or download". For now, use the https version of the URL. We're going to change this in a few minutes - but this makes life simpler to start:

15 lines | ansible/deploy.yml

```yaml
---
- hosts: aws

  vars:
... lines 5 - 6
    ansistrano_deploy_via: git # Method used to deliver the code to the server. Options are copy, rsync, git, svn, s3 or download

    # Variables used in the Git deployment strategy
    ansistrano_git_repo: "https://github.com/knpuniversity/ansible.git" # Location of the git repository
... lines 11 - 15
```

Now copy the last variable: ansistrano_git_branch. We don't *really* need to set this... because it defaults to master. But let's set it anyways:

15 lines | ansible/deploy.yml

```yaml
---
- hosts: aws

  vars:
... lines 5 - 6
    ansistrano_deploy_via: git # Method used to deliver the code to the server. Options are copy, rsync, git, svn, s3 or download

    # Variables used in the Git deployment strategy
    ansistrano_git_repo: "https://github.com/knpuniversity/ansible.git" # Location of the git repository
    ansistrano_git_branch: master # What version of the repository to check out. This can be the full 40-character SHA-1 hash, the liter
... lines 12 - 15
```

Moment of truth! Go back to your terminal and run the playbook again:

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

This time, we see some Git-related tasks. So that's probably good! And it finishes without any errors.

Let's go see what it did! I'll move back to my terminal that's SSH'ed onto the server. Move *out* of the current/ directory. That's important: the current symlink *did* change, but until you move out of it, you're still looking at the *old* release directory.

Ok cool! There are two things in releases/, and the symlink points to the new one. Move back into current/. And... there's our project! Our code is deployed! Yea, we *are* missing some things, like parameters.yml, but we'll get there. For now, celebrate!

Chapter 5: Virtual Host Setup

There is one more immediate problem: the document root of the project is /var/www/project/current/web. But... you might remember that our Nginx virtual host points to /var/www/project/web. This needs to change to /var/www/project/current/web.

That's easy to do: we could just edit this file right now! But since we provisioned our server with Ansible, let's make this change to our provision playbook... so that we feel super cool and responsible.

Sharing Provision Variables

First, in deploy.yml, add a new vars_files key. Load a file called vars/vars.yml:

```
18 lines | ansible/deploy.yml
---
- hosts: aws

  vars_files:
    - ./vars/vars.yml
... lines 6 - 18
```

This *very* small file holds two variables that point to where the project lives:

```
4 lines | ansible/vars/vars.yml
---
project_deploy_dir: /var/www/project
server_document_root: /var/www/project/web
```

These are used by the provision playbook: playbook.yml. The first tells it where to create the directory. And the second - server_document_root - is used to set the document root in the Nginx virtual host!

Before we change that variable, go back to deploy.yml. Now that we're including vars.yml here, we can use the project_deploy_dir variable:

```
18 lines | ansible/deploy.yml
---
- hosts: aws
... lines 3 - 6
  vars:
... line 8
    ansistrano_deploy_to: "{{ project_deploy_dir }}" # Base path to deploy to.
... lines 10 - 18
```

This doesn't change anything: it just kills some duplication.

Updating the Document Root

Back in vars.yml, we need to change server_document_root. But hold on! Let's get fancy! Ansistrano has a variable called ansistrano_current_dir. This is the *name* of the symlinked directory and - as we know - it defaults to current. Put this inside vars.yml and set it to current:

```
6 lines | ansible/vars/vars.yml
---
project_deploy_dir: /var/www/project
ansistrano_current_dir: current # Softlink name. You should rarely changed it.
... lines 4 - 6
```

This won't change how Ansistrano works. But now, we can safely *use* that variable here. Set server_document_root to "{{ project_deploy_dir }}/{{ ansistrano_current_dir }}/web":

```
6 lines | ansible/vars/vars.yml
---
project_deploy_dir: /var/www/project
ansistrano_current_dir: current # Softlink name. You should rarely changed it.
server_document_root: "{{ project_deploy_dir }}/{{ ansistrano_current_dir }}/web"
... lines 5 - 6
```

I love it! After all these changes, well, we didn't *actually* change our deploy playbook at all. But we *did* change the provision playbook.

Find your local terminal and re-provision the server:

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

This will take a few minutes to finish... but all it's *really* doing is changing the virtual host to point to /var/www/project/current/web. If you're not using Ansible to provision, change this however you want!

Done! Move back to your server and open the Nginx config!

```
$ sudo vim /etc/nginx/sites-available/mootube.example.com.conf
```

Got it! The root is set to the correct spot:

```
# /etc/nginx/sites-available/mootube.example.com.conf
server {
    // ...
    root /var/www/project/current/web;
    // ...
}
```

Try out the Site

The ultimate goal is to get our site working at mootube.example.com. In the *real* world, you would configure a DNS record and point it to the server. With AWS, you can do that with their Route 53 service.

But since this is a *fake* domain, we need to cheat. Open up /etc/hosts:

```
$ sudo vim /etc/hosts
```

Near the bottom, put the IP address to our server - I'll copy it from hosts.ini - and point this to mootube.example.com:

```
# /etc/hosts
# ...
54.162.54.206 mootube.example.com
```

At this point, our code is on the server and the Nginx virtual host is pointing to it. We have the absolute basics finished! Find your browser, and try the site! http://mootube.example.com.

It works! I'm just kidding - it's totally a 500 error: we're still missing a few steps.

To see what the exact error is, go to the server and check the logs. In the virtual host, you can see that the error_log config is set to /var/log/nginx/mootube.example.com_error.log:

```
# /etc/nginx/sites-available/mootube.example.com.conf
server {
    # ...
    error_log /var/log/nginx/mootube.example.com_error.log;
    access_log /var/log/nginx/mootube.example.com_access.log;
}
```

Tail that file: sudo tail and then the path:

```
$ sudo tail /var/log/nginx/mootube.example.com_error.log
```

Ah! Look closely:

> Failed opening required vendor/autoload.php

Of course! We have not run composer install yet. In fact, we also haven't configured our database credentials or any file permissions. All we've done is put our code on the server. But... that *is* pretty awesome: we already have a system that deploys in a very cool way: creating a new releases/ directory and symlinking that to current/. Our deploy is missing some steps, but it's already pretty awesome.

But before we finish it, let's talk about deploy keys so that we can deploy *private* repositories.

Chapter 6: Deploying Keys & Private Repos

I want to show you a quick trick. Right now, we're always deploying the master branch:

```yaml
18 lines | ansible/deploy.yml
---
- hosts: aws
... lines 3 - 6
  vars:
... lines 8 - 13
    ansistrano_git_branch: master # What version of the repository to check out. This can be the full 40-character SHA-1 hash, the liter
... lines 15 - 18
```

That probably make sense. But, sometimes, you might want to deploy a different branch, like maybe a feature branch that you're deploying to a beta server. There are a few ways to handle this, but one option is to leverage a native Ansible feature: vars_prompt:

```yaml
24 lines | ansible/deploy.yml
---
- hosts: aws
... lines 3 - 6
  vars_prompt:
... lines 8 - 24
```

With this, we can just ask the user, well, *us*, which branch we want to deploy. Whatever we type will become a new variable called git_branch. For the prompt, say: Enter a branch to deploy. Default the value to master and set private to no... so we can see what we type: this is not a sensitive password:

```yaml
24 lines | ansible/deploy.yml
---
- hosts: aws
... lines 3 - 6
  vars_prompt:
    - name: git_branch
      prompt: 'Enter a branch to deploy'
      default: master
      private: no
... lines 12 - 24
```

Down below, use the variable: "{{ git_branch }}":

```yaml
24 lines | ansible/deploy.yml
---
- hosts: aws
... lines 3 - 12
  vars:
... lines 14 - 19
    ansistrano_git_branch: "{{ git_branch }}" # What version of the repository to check out. This can be the full 40-character SHA-1 has
... lines 21 - 24
```

This is nothing *super* amazing, but if it's useful, awesome! The downside is that it will ask you a question at the beginning of *every* deploy. Try it:

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

There's the prompt! I'll stop the deploy.

How SSH Authentication Works

Now, to the *real* thing I want to talk about: deploy keys. Right now, the *only* reason our deploy works is that... well, our repository is *public*! The server is able to access my repo because *anyone* can. Go copy the ssh version of the URL and use that for ansistrano_git_repo instead:

25 lines | ansible/deploy.yml

```
---
- hosts: aws
... lines 3 - 12
  vars:
... lines 14 - 18
    ansistrano_git_repo: "git@github.com:knpuniversity/ansible.git" # Location of the git repository
... lines 20 - 25
```

Now try the deploy:

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

It starts off good... but then... error! It says:

> Permission denied (public key). Could not read from remote repository

Woh! When you use the ssh protocol for Git, you authenticate with an ssh key. Basically, you generate a private and public key on your machine and then *upload* the public key to your GitHub account. Once you do that, each time you communicate with GitHub, you send your public key so that GitHub knows who you are and what repositories you have access to. And also, behind the scenes, the private key on your local machine is used to *prove* that you own that public key. Actually, none of this is special to git, this is how SSH key-based authentication works anywhere.

Even though our repository is still *public*, you need *some* valid SSH key pair in order to authenticate... and our server has nothing. That's why this is failing. To fix this, we'll use a *deploy* key... which will allow our server to clone the repository, whether it's public or private.

Creating a Deploy Key

Here's how it works. First, locally, generate a new public and private key: ssh-keygen -t rsa -b 4096 -C, your email address - ryan@knpuniversity.com then -f ansible/id_rsa:

```
$ ssh-keygen -t rsa -b 4096 -C "ryan@knpuniversity.com" -f ansible/id_rsa
```

You can use a pass phrase if you want, but I won't. When this is done, we have two new fancy files inside the ansible/ directory: id_rsa - the private key - and id_rsa.pub the key to your local pub. I mean, the public key.

Back on GitHub, on the repository, click "Settings" and then "Deploy Keys". Add a deploy key and give it a name that'll help you remember why you added it. Go find the public key - id_rsa.pub - copy it, and paste it here. Add that key!

Boom! The nice thing is that this will only give our server *read* access to the repository.

Configuring Ansistrano to use the private key

But this is only one half of the equation: we need to tell Ansistrano to *use* the private key - id_rsa - when it communicates with GitHub.

But that's why we use Ansistrano! They already thought about this, and exposed two variables to help: ansistrano_git_identity_key_path and ansistrano_git_identity_key_remote_path. Basically, we need to store the private key *somewhere*: it can live on our local machine where we *execute* Ansible - that's the first variable - or you can put it on the server and use the second variable.

Let's use the first option and store the key locally. Copy the first variable: ansistrano_git_identity_key_path. Set it to {{ playbook_dir }}/id_rsa:

```
25 lines | ansible/deploy.yml
---
- hosts: aws
... lines 3 - 12
  vars:
... lines 14 - 20
    ansistrano_git_identity_key_path: "{{ playbook_dir }}/id_rsa" # If specified this file is copied over and used as the identity key for the
... lines 22 - 25
```

playbook_dir is an Ansible variable, and it points to the ansible/ directory: the directory that holds the playbook file. As *soon* as we do this, Ansistrano will use this private key when it talks to GitHub. And because we've added its partner public key as a deploy key to the repo, it *will* have access!

Storing the Private Key

Of course, this means that you *need* to make sure that the id_rsa file exists. You can either do this manually somehow... or you can do something a bit more controversial: commit it to your repository. I'll do that: add the file, then commit: "adding private deploy identity key to repo".

```
$ git add ansible/id_rsa
$ git commit -m "adding private deploy identity key to repo"
```

This is controversial because I *just* committed a private key to my repository! That's like committing a password! Why did I do this? Mostly, simplicity! Thanks to this, the private key will *always* exist.

How bad of a security issue is this? Well, this key only gives you read-only access to the repository. And, if you were already able to download the code... then you were *already* able to access it. This key doesn't give you any *new* access. But, if you *remove* someone from your GitHub repository... they could still use this key to continue accessing it in the future. *That's* the security risk.

An alternative would be to store the private key on S3, then use the S3 Ansible module to download that onto the server during deployment. Make the decision that's best for you.

Whatever you choose, the point is: the variable is set to a local path on our filesystem where the private key lives. This means... we can deploy again! Try it:

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

It's working... working... and ... it's done! Scroll up a little. Cool! It ran a few new tasks: "Ensure Git deployment key is up to date" and then later "shred Git deployment key". It uses the key, but then removes it from the server after. Nice!

The server can now pull down our code... even if the repository is private.

Next! Deployment is *not* working yet: we still need to setup parameters.yml and do a few other things.

Chapter 7: Ansistrano Stages & Shared Files

When we deploy, our code *is* delivered to the server. But there are a few other things we still need to do, like setting up database credentials in parameters.yml.

The Ansistrano Deploy Stages

Scroll up to the top of the Ansistrano documentation and find the Main Workflow link. When we deploy, Ansistrano goes through five stages: Setup, Update Code, Symlink Shared, Symlink, and Clean Up. The reason this is *really* interesting is that we can add our own custom tasks *before* or *after* any of these stages. For example, we could add a hook to run composer install or create parameters.yml.

The most important stages are "Update Code" - that's when our code is pulled down from git and put into the new releases directory - and "Symlink", which is when the current symlink is changed to that new directory. It's at *that* moment that the site becomes live and traffic starts using the new code.

The Shared Symlink

But look at the third stage: "Symlink Shared". Right now, each release is *completely* separate from the others. We have 3 releases in 3 entirely isolated directories: nothing is shared. But sometimes... you *do* want a file or directory to be shared between deployments. For example, a log file: I want to have just *one* log file that's used across deployments. I don't want each deployment to create a new, empty log file.

In Ansistrano, this is done via the shared/ directory. It's empty right now, but we can configure it to hold certain *shared* paths. For example, eventually, we will want the var/logs directory to be shared. We'll actually *do* this later, but I want you to understand how it works now. When you configure var/logs to be shared in Ansistrano, on the next deploy, this directory will be created inside shared/. Then, every release will have a *symlink* to this shared directory.

That's what the "Symlink Shared" stage does: it creates all the shared symlinks in the new release directory. That's important, because - after this stage - your code should be fully functional.

Creating parameters.yml

Google for "Symfony deployment basics": you should find Symfony's deployment article. It lists the basic things you need to do when deploying a Symfony application, like upload the code, install vendor dependencies and create your app/config/parameters.yml file. Let's handle that next... via an Ansistrano hook!

Chapter 8: Deploy Hooks & parameters.yml

Go look inside the current/ directory on your server. Guess what? There is *no* parameters.yml file yet! That's no surprise! This is *not* committed to Git, so it's not downloaded from Git.

Adding a Hook

Let's add this by adding a *hook* into Ansistrano. How? To add a hook before or after any of these stages, you can override a *variable*... and you can see those variables back down in the variable reference. Ah, yes! Choose the correct variable for the hook you want, set it to a new file, and start adding tasks!

Copy the ansistrano_after_symlink_shared_tasks_file variable: we're going to add a hook *after* the "Symlink Shared" stage. Why there? Well, this is *after* any shared symlinks have been created... but *just* before the site becomes live. Said differently, at this stage, our site is functional... but it's not live yet. It's a great hook spot.

Inside deploy.yml, paste that variable and set it to a new file: {{ playbook_dir }}/deploy/after-symlink-shared.yml:

28 lines | ansible/deploy.yml

```
---
- hosts: aws
... lines 3 - 12
  vars:
... lines 14 - 22
    # Hooks: custom tasks if you need them
    ansistrano_after_symlink_shared_tasks_file: "{{ playbook_dir }}/deploy/after-symlink-shared.yml"
... lines 25 - 28
```

Copy that filename and, inside ansible/, create that deploy/ directory and a new file: after-symlink-shared.yml.

Creating parameters.yml

Ok, next question: how should we create parameters.yml? There are two options. The easier, but less automatic option is to configure app/config/parameters.yml as a *shared* file. If we did that, on the next deploy, Ansistrano would create an app/config/parameters.yml file inside shared/. We could then SSH onto the server manually and configure that file. As *soon* as we did that, all future deploys would use this shared file. We'll cover shared files more later.

But... this requires manual work... and each time the file needs to change... you need to *remember* to update it... manually. I remember nothing!

The second option is to create parameters.yml via Ansible. Inside the ansible/ directory, create a new templates/ directory. Next, copy app/config/parameters.yml.dist from your project into here. Big picture, here's the plan: we will use the Ansible template module, to render variables inside this file, and deploy it to the server. But... to start, we're going to just use these hardcoded values.

Back in after-symlink-shared.yml, add a new task: "Setup infrastructure-related parameters". Use the template module to, for now, copy {{ playbook_dir }}/templates/parameters.yml.dist into the new release directory:

6 lines | ansible/deploy/after-symlink-shared.yml

```
---
- name: Set up infrastructure-related parameters
  template:
    src: '{{ playbook_dir }}/templates/parameters.yml.dist'
... lines 5 - 6
```

But... um... how do we know what the name of the new release directory is? I mean, it's always changing!? And this hook is *before* the current symlink is created, so we can't use that.

Go back to the Ansistrano docs and search for ansistrano_release_path. Yes! Near the bottom, there's a section called "Variables in custom tasks". Ansistrano gives us a few *really* helpful variables... and this explains them.

And yes! The first variable is *exactly* what we need. But don't forget about the others: you may need them someday.

Back in after-symlink-shared.yml, set the destination to {{ ansistrano_release_path.stdout }}/app/config/parameters.yml:

6 lines | ansible/deploy/after-symlink-shared.yml

```
---
- name: Set up infrastructure-related parameters
  template:
    src: '{{ playbook_dir }}/templates/parameters.yml.dist'
    dest: '{{ ansistrano_release_path.stdout }}/app/config/parameters.yml'
```

We're not *customizing* anything in this file yet... but this should be enough to get it onto the server. Let's try it: deploy, deploy!

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

It takes a few moments... but it worked! On your server, move back into the current directory. Yes! *Now* we have a parameters.yml file.

Cool! But... of course... it's still full of hardcoded info. Next, we need to fill this file with our *real*, production config. And we need to do that securely.

Chapter 9: parameters.yml: Handling Secret Config

No matter how you deploy, eventually, you hit the same problem: handling sensitive configuration, like your production database password or Loggly token. Depending on your app, this info will need to be stored in different places, like parameters.yml for a Symfony 3 app or as environment variables for Symfony 4.

But no matter *where* the config needs to ultimately live, the problem is more or less the same: how can we put secret things onto the server in an automated way?

Options for parameters.yml

Like with *everything*, there are a few good answers. And this is where things can get complicated. For parameters.yml, one option is to store the production parameters.yml in a private S3 bucket. Then, during deployment, use the s3 Ansible module to download that into your project.

Another option - the way that *we* will do it - is to store a parameters.yml.dist file in our project, and make it dynamic by printing Ansible variables inside it. To keep things secure, those variables will be stored in the Ansible vault.

Setting up the Deploy Vault

Let's create a new vault to store the secret values:

```
$ ansible-vault create ansible/vars/deploy_vault.yml
```

Choose a safe password... something safer than what I'll choose: beefpass. Here's the plan: we will define some new variables here, then use them inside parameters.yml.dist. So, what needs to be dynamic? For now the secret, loggly_token, database_host, database_user and database_pass.

Back in the vault, create some variables: vault_symfony_secret set to udderly secret $tring and vault_loggly_token set to our production loggly token... this long string:

```
# ansible/vars/deploy_vault.yml
---
vault_symfony_secret: 'udderly secret $string'
vault_loggly_token: 'fb4aa5b2-30a3-4bd8-8902-1aba5a683d62'
```

Oh, and, get your own token... because this is fake.

Then, vault_database_host: 127.0.0.1, vault_database_user: root, and vault_database_password set to null:

```
# ansible/vars/deploy_vault.yml
---
vault_symfony_secret: 'udderly secret $string'
vault_loggly_token: 'fb4aa5b2-30a3-4bd8-8902-1aba5a683d62'
vault_database_host: 127.0.0.1
vault_database_user: root
vault_database_password: null
```

In the provision playbook, we actually install a MySQL server locally. That's why I'm using the local database server... and no, I haven't *bothered* to create a proper user with a decent password. But you *should*.

But also, if I were using AWS for a real application, I would use Amazon's RDS - basically, a hosted MySQL or PostgreSQL database - so that I don't need to manage it on my own. In that case, the database host would be something specific to my RDS instance. But, it's the same idea.

Tip

If you decide to use Amazon's RDS - basically, you need to perform the next steps:

1. Create an RDS instance. The easiest way to do this is by hand via the AWS web interface;
2. Put its credentials in the Ansible Vault.

You can automate the creation of the RDS instance, though it's not as important because these instances are not destroyed and recreated in the same way as EC2 instances.

Save this file and quit. We now have a new, but encrypted, file with those variables:

16 lines | ansible/vars/deploy_vault.yml

```
$ANSIBLE_VAULT;1.1;AES256
37633866356562303665643432386636313937613063373666613739313732313130633364333138
3635376437663463326538633961663931393136643962610a636635363630656264343630613731
30323432303666373366366237393637333932373962373730626465663330326462396532363330
3561346639616234610a353530366337616164633864363038336437623165623534663166633966
31656536326665613063343537333564656334616638663866653934373566303232323036653566
63366436346139333966316364656265633462303233623162353630626661353930343939396661
36326338383538363138633836636235663734303464396236396132306430636362393232366264
34653762303761356262316234333631386262396666313037643661653336613133396439313362
61326464323131646132393538333236643664356539636464373937666530366334313335663637
62383230653762396662653265613133343137336264353233303932313330663831316565636134
35646135303966333639616361306236393134656263386666643532353534313662376232653566
64393665343230353362643363663031636334626466656130616562363038396335323530613135
36313232323766663564666638646562366437616233376632323164303635356631346361376633
6231386263646661383539313439316663333231643261653834
```

Inside deploy.yml, under vars_files, add ./vars/deploy_vault.yml:

30 lines | ansible/deploy.yml

```
---
- hosts: aws

  vars_files:
    - ./vars/deploy_vault.yml
    - ./vars/vars.yml
... lines 7 - 30
```

Creating a Simpler Variables File

At this point, we *could* go directly into parameters.yml.dist and start using those vault_ variables. But, as a best practice, I like to create a separate vars file - deploy_vars.yml - where I assign each of those vault_ variables to a normal variable. Just, stay with me.

Re-open the vault file - type beefpass:

```
$ ansible-vault edit ansible/vars/deploy_vault.yml
```

And copy everything. Then, in deploy_vars.yml, paste that. Now, for each variable, create a *new* variable that's set to it, but *without* the vault_ prefix:

6 lines | ansible/vars/deploy_vars.yml

```
---
symfony_secret: "{{ vault_symfony_secret }}"
loggly_token: "{{ vault_loggly_token }}"
database_host: "{{ vault_database_host }}"
database_user: "{{ vault_database_user }}"
database_password: "{{ vault_database_password }}"
```

This is totally optional. The advantage is that you can quickly see a list of *all* available variables, without needing to open the vault. I can just look in here and say:

> Oh! Apparently there is a variable called symfony_secret!

Back in deploy.yml, import this new file: ./vars/deploy_vars.yml:

30 lines | ansible/deploy.yml

```
---
- hosts: aws

  vars_files:
    - ./vars/deploy_vault.yml
    - ./vars/vars.yml
    - ./vars/deploy_vars.yml
... lines 8 - 30
```

Variables inside parameters.yml.dist

Finally, in parameters.yml.dist, let's print some variables! For database_host, print {{ database_host }}. Repeat that for database_user, database_password, and then down below for symfony_secret, and loggly_token:

23 lines | ansible/templates/parameters.yml.dist

```
# This file is a "template" of what your parameters.yml file should look like
# Set parameters here that may be different on each deployment target of the app, e.g. development, staging, production.
# http://symfony.com/doc/current/best_practices/configuration.html#infrastructure-related-configuration
parameters:
    database_host:     "{{ database_host }}"
    database_port:     ~
    database_name:     mootube
    database_user:     "{{ database_user }}"
    database_password: "{{ database_password }}"
    # You should uncomment this if you want use pdo_sqlite
    #database_path: '%kernel.project_dir%/var/data/data.sqlite'

    mailer_transport:  smtp
    mailer_host:       127.0.0.1
    mailer_user:       ~
    mailer_password:   ~

    # A secret key that's used to generate certain security-related tokens
    secret:            "{{ symfony_secret }}"

    redis_host:        localhost
    loggly_token:      "{{ loggly_token }}"
```

That's it! We put secret things in the vault and then print them inside the parameters file.

Let's try it. Run the playbook with the same command:

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

Yep! This fails because it can't decrypt the vault file. From now on, we need to add a --ask-vault-pass flag. And then type, beefpass:

```
$ ansible-playbook -i ansible/hosts.ini ansible/deploy.yml --ask-vault-pass
```

If this gets really annoying, you can store the password in a file and use --vault-password-file to point to it. Just don't commit that file to your repository!

And... done! Let's go check it out! Move out of the current/ directory and then back in:

```
$ cd ..
$ cd current
```

Deep breath: open parameters.yml. Yes! Everything has its dynamic vault value!

Ok, we're getting *really* close! Next, let's run Composer and fix some permissions!

Chapter 10: Composer & Cache Permissions

Look back at the Symfony Deployment article: we *now* have a parameters file! Woo! Next, we need to run composer install - which was the *original* reason the site didn't work - and then warm up the Symfony cache. We're *really* close to a functional site. We won't need to dump the Assetic assets - we're not using Assetic. But we *will* need to do some asset processing later.

Running composer install

Let's add a new task to run composer install. In the hook file, add "Install Composer deps". Use the composer module and tell it to run the install command. We also need to set the working_dir: use {{ ansistrano_release_path.stdout }}:

11 lines | ansible/deploy/after-symlink-shared.yml

```yaml
---
- name: Set up infrastructure-related parameters
... lines 3 - 6
- name: Install Composer dependencies
  composer:
    command: install
    working_dir: '{{ ansistrano_release_path.stdout }}'
```

Perfect! One gotcha with the composer module is that, by default, it runs composer install --no-dev. That means that your require-dev dependencies in composer.json will *not* be downloaded:

71 lines | composer.json

```json
{
... lines 2 - 30
    "require-dev": {
        "sensio/generator-bundle": "^3.0",
        "symfony/phpunit-bridge": "^3.0",

        "doctrine/data-fixtures": "^1.1",
        "hautelook/alice-bundle": "^1.3"
    },
... lines 38 - 69
}
```

For production, that's a good thing: it will give you a small performance boost. Just make sure that you're not relying on anything in those packages!

Also, in Symfony 3, if you use --no-dev, then some of the post-install Composer tasks will fail, because they *need* those dependencies. To fix that, we need to set an environment variable: SYMFONY_ENV=prod.

No problem! In deploy.yml, add a new key called environment. And below, SYMFONY_ENV set to prod:

33 lines | ansible/deploy.yml

```yaml
---
- hosts: aws
... lines 3 - 27
  environment:
    SYMFONY_ENV: prod
... lines 30 - 33
```

Thanks to this, the Composer post-install tasks will *not* explode. And that's good... it's not great when your deployment explodes.

Oh, and important note: for this all to work, Composer must be already installed on your server. We did that in our provision playbook.

Clearing & Warming Cache

Before we try this, let's tackle one last thing: clearing the Symfony cache... which basically means running two console commands.

To make this easier, in deploy.yml, add a new variable: release_console_path. Copy the ansistrano_release_path.stdout variable and paste it: {{ ansistrano_release_path.stdout }}/bin/console:

35 lines | ansible/deploy.yml

```
---
- hosts: aws
... lines 3 - 14
  vars:
    release_console_path: "{{ ansistrano_release_path.stdout }}/bin/console"

    # Ansistrano vars
... lines 19 - 35
```

Cool! Back in the hook file, add a new task to clear the cache. Use the command module to simply say {{ release_console_path }} cache:clear --no-warmup --env=prod:

17 lines | ansible/deploy/after-symlink-shared.yml

```
... lines 1 - 6
- name: Install Composer dependencies
... lines 8 - 11
- name: Clear the cache
  command: '{{ release_console_path }} cache:clear --no-warmup --env=prod'
... lines 14 - 17
```

That's basically the command that you see in the docs.

If you're not familiar with the --no-warmup flag, it's important. In Symfony 4, instead of running cache:clear and expecting it to clear your cache *and* warm up your cache, cache:clear will *only* clear your cache. Then, you should use cache:warmup separately to warm it up. By passing --no-warmup, we're imitating the Symfony 4 behavior so that we're ready.

Add the second task: "Warm up the Cache". Copy the command, but change it to just cache:warmup --env=prod:

17 lines | ansible/deploy/after-symlink-shared.yml

```
... lines 1 - 11
- name: Clear the cache
  command: '{{ release_console_path }} cache:clear --no-warmup --env=prod'

- name: Warm up the cache
  command: '{{ release_console_path }} cache:warmup --env=prod'
```

Now, technically, since the cache/ directory is not shared between deploys, we don't *really* need to run cache:clear: it will always be empty at this point! But, I'll keep it.

Ok! Phew! I think we've done everything. Let's deploy! Find your local terminal and run the playbook:

```
$ ansible-playbook -i ansible/hosts.ini ansible/deploy.yml --ask-vault-pass
```

Use beefpass as the vault password and deploy to master. Then... wait impatiently! Someone fast forward, please!

Yes! No errors! On the server, move out of current/ and then back in. Check it out! Our vendor/ directory is filled with goodies!

Fixing the File Permissions

Moment of truth: try the site again: mootube.example.com. Bah! It *still* doesn't work. Let's find out why. On the server, tail the log file:

```
$ sudo tail /var/log/nginx/mootube.example.com_error.log
```

Ooooh:

> PHP Fatal error: The stream or file "var/logs/prod.log" could not be opened

Of course! We have permissions problems on the var/ directory! Fixing this is actually a *very* interesting topic. There is an easy way to fix this... and a more complex, but more secure way.

For now, let's use the simple way: I *really* want our app to work! Add a new task: "Setup directory permissions for var". Use the file module. But, quickly, go back to deploy.yml and make another variable: release_var_path set to the same path {{ ansistrano_release_path.stdout }}/var:

36 lines | ansible/deploy.yml

```
---
- hosts: aws
... lines 3 - 14
  vars:
    release_console_path: "{{ ansistrano_release_path.stdout }}/bin/console"
    release_var_path: "{{ ansistrano_release_path.stdout }}/var"
... lines 18 - 36
```

Now, back in after-symlink-shared.yml, set the path to {{ release_var_path }}, state to directory, mode to 0777 and recurse: true:

24 lines | ansible/deploy/after-symlink-shared.yml

```
... lines 1 - 17
- name: Setup directory permissions for var/
  file:
    path: "{{ release_var_path }}"
    state: directory
    mode: 0777
    recurse: true
```

On deploy, this will make sure that the directory exists and is set to 777. That's not the *best* option for security... but it should get things working!

Deploy one more time:

```
$ ansible-playbook -i ansible/hosts.ini ansible/deploy.yml --ask-vault-pass
```

Type beefpass, deploy to master... and watch the magic. I can see the new directory permissions task... and it finishes.

Refresh the site! Eureka! Yea, it's *still* a 500 error, but this comes from Symfony! Symfony *is* running! Change the URL to http://mootube.example.com/about. It works! Yea, it's *super* ugly - we need to do some work with our assets - but it *does* work. The homepage is broken because our database isn't setup. But this static page proves our deploy is functional! Victory!

Now, let's smooth out the missing details... like the insecure permissions, the database and our assets... because this site is *horrible* to look at!

Chapter 11: Building Webpack Encore Assets

Our site is at *least* functional. Well any page that doesn't use the *database* is functional... like the about page. But it *is* super ugly. Oof. Why? It's simple! Our CSS file - build/styles.css is missing! Oooh, a mystery!

Our MooTube asset setup is pretty awesome: instead of having simple CSS files, we're getting sassy with Sass! And to build that into CSS, we're using an awesome library called Webpack Encore. Actually, we have a tutorial on Webpack... so go watch that if you're curious!

Basically, Webpack Encore is a Node executable: you run it from the command line, and it - in our simple app - transforms that Sass file into build/styles.css.

The reason styles.css wasn't deployed is that its directory - /web/build - is *ignored* in .gitignore!

22 lines | .gitignore

```
... lines 1 - 20
/web/build
```

We need to do more work to deploy it.

Running assets:install

But before we get too far into that, there's one other *little* command that you often run during deployment:

```
$ bin/console assets:install --symlink
```

Actually, when you run composer install, this command is run automatically... so you may not even realize it's happening. And for deploy... well... you may or may *not* even need it! Here's the deal: sometimes, a bundle - usually a third-party bundle - will come with some CSS, JS or other public assets. Those files, of course, live in the vendor/ directory... which is a *problem*... because it means they're not publicly accessible. To *make* them public, we run this command. For each bundle that has public assets, it creates a symlink in the web/bundles/ directory.

For our app... yea... we don't have *any* bundles that do this! But, let's run that command on deploy to be safe. Open up after-symlink-shared.yml. Let's add a new task called "Install bundle assets":

45 lines | ansible/deploy/after-symlink-shared.yml

```
... lines 1 - 14
- name: Warm up the cache
  command: '{{ release_console_path }} cache:warmup --env=prod'

- name: Install bundle assets
... lines 19 - 45
```

Set this to run a command, and use the release_console_path variable that we setup in deploy.yml:

39 lines | ansible/deploy.yml

```
---
- hosts: aws
... lines 3 - 14
  vars:
    release_console_path: "{{ ansistrano_release_path.stdout }}/bin/console"
... lines 17 - 39
```

Add assets:install --symlink and then --env=prod:

45 lines | ansible/deploy/after-symlink-shared.yml

```
... lines 1 - 17
- name: Install bundle assets
  command: '{{ release_console_path }} assets:install {{ release_web_path }} --symlink --no-debug --env=prod'
... lines 20 - 45
```

Tip

The --symlink is optional, and depending on your setup, you may need to *not* pass this flag

That's important: we need to run all of our console commands with --env=prod. Running commands in dev mode won't even work, because our require-dev Composer dependencies were never installed.

Perfect!

Installing Node Dependencies

Let's move on to the *real* task: building our assets on production. I'm not going to talk about Encore too much, but building the assets is a two-step process.

First, run:

```
$ yarn install
```

to download all of your Node dependencies. Basically, this reads package.json

8 lines | package.json

```
{
  "devDependencies": {
    "@symfony/webpack-encore": "^0.12.0",
    "node-sass": "^4.5.3",
    "sass-loader": "^6.0.6"
  }
}
```

And downloads that stuff into a node_modules/ directory. It's basically like Composer for Node.

Step 2 is to run Encore and build your assets. First, I'll clear out the build directory:

```
$ rm -rf web/build/*
```

Then, run:

```
$ ./node_modules/.bin/encore production
```

Cool! Check it out! Yes! We still have styles.css, and it's beautifully minified.

Where to Build the Assets?

So how can we do this during deploy? Well... we have a few options. First, you *could* decide to run these commands locally and commit those files to your repository. That makes deploying easy... but you need to remember to run this command before each deploy. And committing built files to your repo is a bummer. This is an easy, but hacky way to handle things.

But... we *do* need to run these commands *somewhere*. The most obvious solution is actually to run these commands on your production server *during* deployment. This *is* what we're going to do... but it's *also* a bummer. It means that we will need to install Node on our production server... *just* to build these assets.

So, if you really don't like the idea of running these commands on production, you have a few other options. If you already have a build system of some sort, you could build your assets on that machine, upload them to S3 or something similar, then download them during deployment. Or, skip the downloading part, and update your script and link tags to point to S3 or some CDN.

A second option is to add a *play* to your deploy playbook that would first build those assets *locally*, before using the copy module to move them up to production.

Installing Node & Yarn

We're going to build the assets on production. This is not the best solution, but it's easy, and works great in most situations. The only big requirement is that we need to install Node and Yarn there. I'm going to open up my *provision* playbook: playbook.yml. Then, near the bottom, paste some tasks that do this:

171 lines | ansible/playbook.yml

```
---
- hosts: webserver
... lines 3 - 33
  tasks:
... lines 35 - 124
    # Node
    - name: Register NodeJS distribution
      shell: 'curl -sL https://deb.nodesource.com/setup_8.x | sudo -E bash -'
      changed_when: false

    - name: Install NodeJS
      become: true
      apt:
        name: nodejs
        state: latest
        update_cache: yes

    # Yarn
    - name: Add Yarn APT key
      become: true
      apt_key:
        url: 'https://dl.yarnpkg.com/debian/pubkey.gpg'
        state: present

    - name: Add Yarn to the source lists
      become: true
      lineinfile:
        path: '/etc/apt/sources.list.d/yarn.list'
        regexp: 'deb https://dl.yarnpkg.com/debian/ stable main'
        line: 'deb https://dl.yarnpkg.com/debian/ stable main'
        create: yes

    - name: Install Yarn package manager
      become: true
      apt:
        name: yarn
        state: latest
        update_cache: yes
... lines 158 - 171
```

If you're not using Ansible to provision your server, just install Node and yarn however you want. Let's get this running!

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini --ask-vault-pass -l aws
```

Use beefpass for the password. Go go go!

Build the Assets on Deploy

While that's doing its magic, open after-symlink-shared.yml. We need to run 2 commands on deploy. Add a new task called "Install Node dependencies". Use command set to yarn install. Oh, and make sure this runs *in* our project directory: add args then chdir set to our handy ansistrano_release_path.stdout:

32 lines | ansible/deploy/after-symlink-shared.yml

```yaml
... lines 1 - 17
- name: Install bundle assets
  command: '{{ release_console_path }} assets:install --symlink --env=prod'

- name: Install Node dependencies
  command: yarn install
  args:
    chdir: '{{ ansistrano_release_path.stdout }}'
... lines 25 - 32
```

Copy that to make the second task: Install Webpack Encore assets. For the command, use ./node_modules/.bin/encore production:

37 lines | ansible/deploy/after-symlink-shared.yml

```yaml
... lines 1 - 20
- name: Install Node dependencies
  command: yarn install
  args:
    chdir: '{{ ansistrano_release_path.stdout }}'

- name: Install Webpack Encore assets
  command: './node_modules/.bin/encore production'
  args:
    chdir: '{{ ansistrano_release_path.stdout }}'
... lines 30 - 37
```

Ok, that's pretty easy! The annoying part is just *needing* to setup Node on your production server. Let's go back and check on the provision!

Oh boy... let's fast forward... ding!

Thanks to that, we *should* have Node and yarn up on the server. Let's deploy! Same command, but with deploy.yml, and we don't need the -l aws... this already *only* runs against the aws host:

```
$ ansible-playbook ansible/deploy.yml -i ansible/hosts.ini --ask-vault-pass
```

Use beefpass, deploy master and... hold your breath! Most of this will be the same... but watch those two new tasks. The Node dependencies will take some time to install at first.

Ok done! No errors. So... let's try it! Refresh! We have CSS! Yes!

We are finally ready to fix the database... and our poor homepage.

Chapter 12: Database Setup

Our homepage is *busted*. Find your terminal and SSH onto the server:

```
$ ssh -i ~/.ssh/KnpU-Tutorial.pem ubuntu@54.XXX.XX.XXX
```

Let's find out what the problem is. I recognize the error page as one that comes from *Symfony*... which means Symfony *is* running, and the error will live in *its* logs.

Check out the var/logs/prod.log file:

```
$ cd /var/www/project/current
$ tail vars/logs/prod.log
```

Ah, no surprise!

> Unknown database "mootube"

So how do we setup the database? Well, it's up to you! This is just a one-time thing... so it doesn't need to be part of your deploy. You could go directly to MySQL and setup it up manually. That's super fine.

Creating the Database

I'm going to do it in Ansible! Add a new task... but move it above the cache tasks, in case the database is needed during cache warm up. Name it: "Create DB if not exists":

42 lines | ansible/deploy/after-symlink-shared.yml

```
... lines 1 - 14
- name: Warm up the cache
  command: '{{ release_console_path }} cache:warmup --env=prod'

- name: Create DB if not exists
... lines 19 - 42
```

We can use the doctrine:database:create command. So, use the command module... and I'll copy one of our other commands. Change it to doctrine:database:create then --if-not-exists so it won't explode if the database already exists:

42 lines | ansible/deploy/after-symlink-shared.yml

```
... lines 1 - 17
- name: Create DB if not exists
  command: '{{ release_console_path }} doctrine:database:create --if-not-exists --env=prod'
... lines 20 - 42
```

If you run this command locally... well... we *do* have a database already. So it says:

> ... already exists. Skipped.

Copy that text. This last part is optional: we're going to configure this task to know when it was, or was not *changed*. Register a variable called create_db_output. Then, add changed_when set to not create_db_output.stdout|search() and paste that text:

```
42 lines | ansible/deploy/after-symlink-shared.yml
... lines 1 - 17
- name: Create DB if not exists
  command: '{{ release_console_path }} doctrine:database:create --if-not-exists --env=prod'
  register: create_db_output
  changed_when: not create_db_output.stdout|search('already exists. Skipped')
... lines 22 - 42
```

Migrating / Creating the Schema

That'll give us a database. But... we need some schema! Some tables! How do we get those!? Well... you *should* use migrations in your app. We *do* have migrations: in app/DoctrineMigrations... and these contain *everything*. I mean, these have all the queries needed to add all the tables... starting from an empty database. I *highly* recommend creating migrations that can build from scratch like this.

So, to build the schema - or migrate any *new* schema changes on future deploys - we just need to run our migrations.

Create a new task: "Run migrations". Then cheat and copy the previous task. This is simple enough: run doctrine:migrations:migrate with --no-interaction, so that it won't interactively ask us to confirm before running the migrations. Interactive prompts are *no* fun for an automated deploy:

```
47 lines | ansible/deploy/after-symlink-shared.yml
... lines 1 - 17
- name: Create DB if not exists
  command: '{{ release_console_path }} doctrine:database:create --if-not-exists --env=prod'
  register: create_db_output
  changed_when: not create_db_output.stdout|search('already exists. Skipped')

- name: Run migrations
  command: '{{ release_console_path }} doctrine:migrations:migrate --no-interaction --env=prod'
... lines 25 - 47
```

Register another variable - run_migrations_output - and use that below:

```
47 lines | ansible/deploy/after-symlink-shared.yml
... lines 1 - 22
- name: Run migrations
  command: '{{ release_console_path }} doctrine:migrations:migrate --no-interaction --env=prod'
  register: run_migrations_output
... lines 26 - 47
```

If you try to migrate and you are already fully migrated, it says:

> No migrations to execute.

Let's search for that text: "No migrations to execute":

```
47 lines | ansible/deploy/after-symlink-shared.yml
... lines 1 - 22
- name: Run migrations
  command: '{{ release_console_path }} doctrine:migrations:migrate --no-interaction --env=prod'
  register: run_migrations_output
  changed_when: not run_migrations_output.stdout|search('No migrations to execute')
... lines 27 - 47
```

Oh, before we try this, make sure you don't have any typos: the variable is create_db_output:

```
47 lines | ansible/deploy/after-symlink-shared.yml
... lines 1 - 17
- name: Create DB if not exists
... line 19
  register: create_db_output
  changed_when: not create_db_output.stdout|search('already exists. Skipped')
... lines 22 - 47
```

Ok, try it!

```
$ ansible-playbook ansible/deploy.yml -i ansible/hosts.ini --ask-vault-pass
```

After a bunch of setup tasks... if you watch closely... yea! The migrations ran successfully! We *should* have a database full of tables.

Go back to the site and refresh! It works! Of course... there's no *data*, but it works!

Why dev Commands Don't Work

To help bootstrap my data, *just* this once, I'm going to load my fixtures on production. I'm obviously *not* going to make this part of my deploy: you won't make any friends if you constantly empty the production database. Believe me.

Find the terminal that is SSH'ed to the server. Move out of the current/ directory and then back in:

```
$ cd ..
$ cd current/
```

First, try running bin/console *without* --env=prod:

```
$ bin/console
```

Error! It can't find a bundle! Why? In the dev environment, we use a few bundles - like HautelookAliceBundle - that are in the require-dev section of our composer.json. So, these do *not* exist inside vendor/ right now!

That is why you *must* run all commands with --env=prod. But, of course, the fixtures bundle is *only* available in the dev environment. So, *just* this one time... manually... let's install the dev dependencies with:

```
$ composer install
```

Now we can load our fixtures:

```
$ ./bin/console hautelook_alice:doctrine:fixtures:load
```

Beautiful! And *now*, we've got some great data to get us started. Next, let's talk more about migrations... because if you're not careful, you may temporarily take your site down! That's not as bad as emptying the production database, but it still ain't great.

Chapter 13: Safe Migrations

When we deploy, our migrations run! Woohoo! Yep, we can just generate migrations and everything happens automatically on deploy.

Making a Schema Change

Oooooooh, but there's a catch! Open src/AppBundle/Entity/Video.php. This entity has a field called image:

```
99 lines | src/AppBundle/Entity/Video.php
... lines 1 - 10
class Video
{
... lines 13 - 28
    /**
     * @var string
     *
     * @ORM\Column(type="string", length=255)
     */
    private $image;
... lines 35 - 66
    /**
     * @return string
     */
    public function getImage()
    {
        return $this->image;
    }

    /**
     * @param string $image
     */
    public function setImage($image)
    {
        $this->image = $image;
    }
... lines 82 - 97
}
```

Ya know what? I'd rather call that *poster*, because it's the *poster* image for this video.

Because the annotation doesn't have a name option, renaming the property means that the column will be renamed in the database. And that means... drum roll... we need a migration!

But first, we also need to update a few parts of our code, like our fixtures. I'll search for image: and replace it with poster::

32 lines | src/AppBundle/DataFixtures/ORM/video.yml

```
AppBundle\Entity\Video:
    video_1:
... line 3
        poster: images/cowbell.png
... lines 5 - 7
    video_2:
... line 9
        poster: images/dandelion.png
... lines 11 - 13
    video_3:
... line 15
        poster: images/bovine1.png
... lines 17 - 19
    video_4:
... line 21
        poster: images/bovine2.png
... lines 23 - 25
    video_5:
... line 27
        poster: images/milkjug.png
... lines 29 - 32
```

Then, open a template: app/Resources/views/default/index.html.twig. Search for .image:

53 lines | app/Resources/views/default/index.html.twig

```
... lines 1 - 2
{% block content %}
    <div class="video-content-container">
... lines 5 - 22
        <div class="video-box">
... lines 24 - 27
            {% if tags %}
                {% for video in videos %}
                    <a href="#video-{{ video.id }}">
                        <div class="video-container">
                            <img class="video-image" src="{{ asset(video.image) }}">
... lines 33 - 36
                        </div>
                    </a>
                {% endfor %}
... lines 40 - 48
            {% endif %}
        </div>
    </div>
{% endblock %}
```

Ah, yes! Change that to .poster:

53 lines | app/Resources/views/default/index.html.twig

```
... lines 1 - 2
{% block content %}
    <div class="video-content-container">
... lines 5 - 22
        <div class="video-box">
... lines 24 - 27
            {% if tags %}
                {% for video in videos %}
                    <a href="#video-{{ video.id }}">
                        <div class="video-container">
                            <img class="video-image" src="{{ asset(video.poster) }}">
... lines 33 - 36
                        </div>
                    </a>
                {% endfor %}
... lines 40 - 48
            {% endif %}
        </div>
    </div>
{% endblock %}
```

Brilliant! All we need to do now is write a migration to rename that column. Easy! Switch to your local terminal and run:

```
$ ./bin/console doctrine:migrations:diff
```

Go check it out in app/DoctrineMigrations. Wow... it's actually perfect:

> ALTER TABLE video CHANGE image (to) poster...

35 lines | app/DoctrineMigrations/Version20170927100553.php

```
... lines 1 - 7
/**
 * Auto-generated Migration: Please modify to your needs!
 */
class Version20170927100553 extends AbstractMigration
{
    /**
     * @param Schema $schema
     */
    public function up(Schema $schema)
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'mysql', 'Migration can only be executed safely on \'mys

        $this->addSql('ALTER TABLE video CHANGE image poster VARCHAR(255) NOT NULL');
    }
... lines 23 - 33
}
```

Doctrine is smart enough to know that we should *rename* the column instead of dropping the old column and adding a new one.

Dangerous Deploy Ahead!

Great! Let's deploy! Right!? Sure... if you want to take your site down for a minute or two! Can you see the problem? If we deploy now, this migration will run... and about 1 minute later, the deploy will finish and the new code will be used. The problem is *during* that period. As *soon* as this migration executes, the image column will be *gone*... but the live site will still try to use it! That's a huge problem.

Nope, we need to be smarter: we need to write *safe* migrations. Here's the idea: only write migrations that *add* new things & never write migrations that *remove* things... unless that thing is not being used at *all* by the live site.

Writing Safe Migrations

This creates a *slightly* different workflow... with *two* deploys. For the first deploy, change the migration:
ALTER TABLE video ADD poster:

39 lines | app/DoctrineMigrations/Version20170927100553.php

```
... lines 1 - 10
class Version20170927100553 extends AbstractMigration
{
    /**
     * @param Schema $schema
     */
    public function up(Schema $schema)
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'mysql', 'Migration can only be executed safely on \'mys

        //$this->addSql('ALTER TABLE video CHANGE image poster VARCHAR(255) NOT NULL');
        $this->addSql('ALTER TABLE video ADD poster VARCHAR(255) NOT NULL');
... line 23
    }
... lines 25 - 37
}
```

We're not going to remove the image column yet. But now, we *do* need to migrate the data:
UPDATE video SET poster = image:

39 lines | app/DoctrineMigrations/Version20170927100553.php

```
... lines 1 - 10
class Version20170927100553 extends AbstractMigration
{
    /**
     * @param Schema $schema
     */
    public function up(Schema $schema)
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'mysql', 'Migration can only be executed safely on \'mys

        //$this->addSql('ALTER TABLE video CHANGE image poster VARCHAR(255) NOT NULL');
        $this->addSql('ALTER TABLE video ADD poster VARCHAR(255) NOT NULL');
        $this->addSql('UPDATE video SET poster = image');
    }
... lines 25 - 37
}
```

Honestly, I usually don't worry about the down()... I've actually never rolled back a deploy before. But, let's update it to be safe: SET image = poster, and then ALTER TABLE to drop poster:

```
39 lines | app/DoctrineMigrations/Version20170927100553.php
... lines 1 - 10
class Version20170927100553 extends AbstractMigration
{
... lines 13 - 25
    /**
     * @param Schema $schema
     */
    public function down(Schema $schema)
    {
        // this down() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'mysql', 'Migration can only be executed safely on \'mys

        //$this->addSql('ALTER TABLE video CHANGE poster image VARCHAR(255) NOT NULL COLLATE utf8_unicode_ci');
        $this->addSql('UPDATE video SET image = poster');
        $this->addSql('ALTER TABLE video DROP poster');
    }
}
```

This is a safe migration. First, try it locally:

```
$ ./bin/console doctrine:migrations:migrate
```

Perfect! And now... deploy! Right? No! Stop that deploy! If you deploy now... well... you're not going to deploy *anything*. We have not committed or pushed our changes yet!

This is actually the first time that we've made changes to our *code*, and that's why this is the first time we've needed to worry about this. Commit the changes and run:

```
$ git push origin master
```

Now deploy:

```
$ ansible-playbook ansible/deploy.yml -i ansible/hosts.ini --ask-vault-pass
```

Type in beefpass and deploy to master. If you watch closely, the migration task should show as *changed*... because it *is* running one migration.

The site still works with *no* downtime.

Removing Columns/Tables

What about the extra image column that's still in the database? Now that it's not being used at *all* on production, it's safe to remove on a *second* deploy. Run:

```
$ ./bin/console doctrine:migrations:diff
```

This time it perfectly sees the DROP:

```
35 lines | app/DoctrineMigrations/Version20170927102503.php
... lines 1 - 10
class Version20170927102503 extends AbstractMigration
{
    /**
     * @param Schema $schema
     */
    public function up(Schema $schema)
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'mysql', 'Migration can only be executed safely on \'mys

        $this->addSql('ALTER TABLE video DROP image');
    }

    /**
     * @param Schema $schema
     */
    public function down(Schema $schema)
    {
        // this down() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'mysql', 'Migration can only be executed safely on \'mys

        $this->addSql('ALTER TABLE video ADD image VARCHAR(255) NOT NULL');
    }
}
```

Commit this new file and push:

```
$ git add .
$ git commit -m "Removing unused column"
$ git push origin master
```

Deploy!

```
$ ansible-playbook ansible/deploy.yml -i ansible/hosts.ini --ask-vault-pass
```

This time, when the image column is removed, the production code is *already* not using it.

The Edge Case: Updated Data

There *is* still one edge-case problem. On the first deploy, we used an UPDATE statement to set poster = image. That makes those columns identical:

```
39 lines | app/DoctrineMigrations/Version20170927100553.php
... lines 1 - 10
class Version20170927100553 extends AbstractMigration
{
... lines 13 - 15
    public function up(Schema $schema)
    {
... lines 18 - 22
        $this->addSql('UPDATE video SET poster = image');
    }
... lines 25 - 37
}
```

But, for then next few seconds, the production code is *still* using the old image column. That's fine... unless people are making *changes* to its data! Any changes made to image during this period will be *lost* when the *new* production code stops reading that column.

If you have this problem, you're going to need to be a little bit more intelligent, and potentially run another UPDATE statement immediately after the new code becomes live.

Ok! Our final migration ran, the deploy finished and the site still works... with no downtime.

Next! Let's *share* files... and make our deploy faster!

Chapter 14: Faster Deploy with Shared Files

So far, each release is independent. And sometimes, that sucks! Each release has its *own* log files. There's nothing in logs/ right now, but usually you'll find a prod.log file. The problem is that if you need to go look inside to debug an issue, you might have to look through 10 separate prod.log files across 10 separate deploys!

Sharing a Path between Deploys

This is a perfect example of a file that we want to share *between* deployments. Fortunately - as I mentioned earlier - Ansistrano has us covered. Check out the Ansistrano documentation. Ah yes, we need this ansistrano_shared_paths variable! Copy it! Then, in deploy.yml, add it near the top:

53 lines | ansible/deploy.yml

```
---
- hosts: aws
... lines 3 - 14
  vars:
... lines 16 - 18
    # Ansistrano vars
... lines 20 - 21
    # Arrays of directories and files to be shared.
    # The following arrays of directories and files will be symlinked to the current release directory after the 'update-code' step and its c
    # Notes:
    # * Paths are relative to the /shared directory (no starting /)
    # * If your items are in a subdirectory, write the entire path to each shared directory
    #
    # Example:
    # ansistrano_shared_paths:
    #   - path/to/first-dir
    #   - path/next-dir
    # ansistrano_shared_files:
    #   - my-file.txt
    #   - path/to/file.txt
    ansistrano_shared_paths:
... lines 36 - 53
```

It's simple, we want to share var/logs: that entire directory:

53 lines | ansible/deploy.yml

```yaml
---
- hosts: aws
... lines 3 - 14
  vars:
... lines 16 - 18
    # Ansistrano vars
... lines 20 - 21
    # Arrays of directories and files to be shared.
    # The following arrays of directories and files will be symlinked to the current release directory after the 'update-code' step and its c
    # Notes:
    # * Paths are relative to the /shared directory (no starting /)
    # * If your items are in a subdirectory, write the entire path to each shared directory
    #
    # Example:
    # ansistrano_shared_paths:
    #   - path/to/first-dir
    #   - path/next-dir
    # ansistrano_shared_files:
    #   - my-file.txt
    #   - path/to/file.txt
    ansistrano_shared_paths:
      - var/logs
... lines 37 - 53
```

Oh, and now that var/logs will be a symlink, in after-symlink-shared.yml, under the permissions task, we need to add follow: true so that the permissions change *follows* the symlink:

49 lines | ansible/deploy/after-symlink-shared.yml

```yaml
... lines 1 - 40
- name: Setup directory permissions for var/
  file:
... lines 43 - 46
    # We need it because logs are symlinks now
    follow: yes
```

And back in deploy.yml... yea...my variable didn't paste well. Make sure your indentation is correct!

53 lines | ansible/deploy.yml

```yaml
---
- hosts: aws
... lines 3 - 14
  vars:
... lines 16 - 18
    # Ansistrano vars
... lines 20 - 21
    # Arrays of directories and files to be shared.
    # The following arrays of directories and files will be symlinked to the current release directory after the 'update-code' step and its c
    # Notes:
    # * Paths are relative to the /shared directory (no starting /)
    # * If your items are in a subdirectory, write the entire path to each shared directory
    #
    # Example:
    # ansistrano_shared_paths:
    #   - path/to/first-dir
    #   - path/next-dir
    # ansistrano_shared_files:
    #   - my-file.txt
    #   - path/to/file.txt
    ansistrano_shared_paths:
      - var/logs
... lines 37 - 53
```

Now! Find your local terminal and deploy!

```
$ ansible-playbook ansible/deploy.yml -i ansible/hosts.ini --ask-vault-pass
```

Shared Paths & Avoiding Server Storage

Use beefpass and deploy to master. Make sure you think about what other directories or files you might need to share between deploys, like web/uploads if you store uploaded files on your server. Or, web/imagine if you use LiipImagineBundle. Otherwise, all the thumbnails will need to be re-created after each deploy. That's lame!

But also keep in mind that there are some big advantages to *not* storing files like these on your server. Instead of putting uploaded files in web/uploads, you could store them in a cloud store, like S3. If you put *nothing* extra on your server, it's *really* easy to destroy your server and launch a new one... without needing to worry about copying over a bunch of random, uploaded files. It also makes using *multiple* servers possible.

Ding! Move over to the terminal that's SSH'ed onto the server. Go out of current/ and then back in:

```
$ cd ..
$ cd current/
```

Check out var/logs:

```
$ ls -la var/
```

it's now a symlink to shared/var/logs. That directory is empty, but as soon as we log something, a prod.log file will show up there.

Sharing Files for Performance

There's *one* other reason you might want to share some paths: speed! Right now, our deploy is slow! You may not have noticed because of the magic of television: we've been fast-forwarding through the deploys! But, in real life, the Install Node dependencies task takes almost 2 minutes to run! Woh!

Why is it so slow? Because it needs to download *all* of the dependencies on *every* single deploy. But if we *shared* the node_modules/ directory, then yarn install would start with a *full* directory. It would only need to download any *changes* since the last deploy! This is an easy win!

Add node_modules to the shared paths:

54 lines | ansible/deploy.yml

```
---
- hosts: aws
... lines 3 - 14
  vars:
... lines 16 - 34
    ansistrano_shared_paths:
      - var/logs
      - node_modules
... lines 38 - 54
```

Cleaning up old Releases

Oh, and before we deploy, it's time to fix one other thing. Back on the server, go into the releases/ directory:

```
$ cd releases/
$ ls -la
```

Ok! It's getting crowded in here! Each deploy creates a new directory... and this will gone on forever and ever until we run out of disk space. Fun! Go back to the Ansistrano docs and find the ansistrano_keep_releases variable. This is the key. In deploy.yml, paste that and set it to 3:

55 lines | ansible/deploy.yml

```
---
- hosts: aws
... lines 3 - 14
  vars:
... lines 16 - 18
    # Ansistrano vars
... line 20
    ansistrano_keep_releases: 3 # Releases to keep after a new deployment. See "Pruning old releases".
... lines 22 - 55
```

Ok, let's try it! Find your local terminal and deploy:

```
$ ansible-playbook ansible/deploy.yml -i ansible/hosts.ini --ask-vault-pass
```

Use beefpass and deploy to master. I'll fast-forward... but I'll tell you how long the deploy *really* took. This *first* deploy will still be slow: the node_modules/ directory will *start* empty. By the way, the composer install command is also a little bit slow... but not *nearly* as slow as yarn install. Why? Because Composer caches the packages behind the scenes. So even though the vendor/ directory starts empty, composer install runs *pretty* quickly. We *could* make it faster by sharing vendor/... but that's a bad idea! If we did that, when a future deploy updated the vendors, this would affect the *live* site during the deploy! Scary!

Ok, done! I'm deploying to a *tiny*, slow server, so that took 3 and a half minutes. Almost half of that was for yarn install!

Let's deploy again:

```
$ ansible-playbook ansible/deploy.yml -i ansible/hosts.ini --ask-vault-pass
```

While we're waiting, go back into the server:

```
$ cd releases/
$ ls -la
```

Yes! There are only 3 releases. In shared/, node_modules/ is populated, thanks to the last deploy:

```
$ cd ..
$ cd shared/
$ ls -la node_modules/
```

When the deploy finishes... awesome! The yarn install task was almost *instant*, and the deploy was nearly two minutes faster! Zoom!

Next, it's time to demystify and fix our cache directory permissions.

Chapter 15: Logs, Sessions & File Permissions

Let's tackle one of the most *confusing* things in Symfony: how to handle file permissions for the cache directory.

To get our site working, we're setting the *entire* var/ directory to 777:

```
49 lines | ansible/deploy/after-symlink-shared.yml
... lines 1 - 40
- name: Setup directory permissions for var/
  file:
    path: "{{ release_var_path }}"
    state: directory
    mode: 0777
    recurse: true
    # We need it because logs are symlinks now
    follow: yes
```

This includes cache/, logs/ and sessions/.

This is a bummer for security. Here's my big question: after we deploy, which files *truly* need to be writable by the web server?

Let's solve this ancient Symfony mystery. To start, instead of setting the entire var/ directory to 777, let's *just* do this for var/logs. This is actually the reason we originally created this task: our site was failing because var/logs wasn't writable.

But first, back in deploy.yml, create a new variable: release_logs_path set to {{ ansistrano_shared_path }}/var/logs:

```
56 lines | ansible/deploy.yml
---
- hosts: aws
... lines 3 - 14
  vars:
... lines 16 - 17
    release_logs_path: "{{ ansistrano_shared_path }}/var/logs"
... lines 19 - 56
```

ansistrano_shared_path is a special variable that Ansistrano gives us. Thanks!

Copy that variable, and back in after-symlink-shared.yml, use it:

```
48 lines | ansible/deploy/after-symlink-shared.yml
... lines 1 - 40
- name: Setup directory permissions for var/logs
  become: true
  file:
    path: '{{ release_logs_path }}'
    state: directory
    mode: 0777
    recurse: true
```

Oh, and we don't need follow anymore. But *do* add become: true. Why? The files in this directory - like prod.log - will probably be created by the web server, so, www-data. The become: true will allow us to change those permissions.

Ok, let's try this! Find your local terminal, and deploy!

```
$ ansible-playbook ansible/deploy.yml -i ansible/hosts.ini --ask-vault-pass
```

When this finishes, *only* var/logs/ should be writable.

Deep breath. Refresh! Dang! It fails! That's ok! Let's play detective and uncover the problem.

Using Native PHP Sessions

Back on the server, find the var/logs directory and tail prod.log:

```
$ cd shared/var/logs
$ tail prod.log
```

Oh!

> Unable to create the directory var/sessions

Apparently the var/sessions directory needs to be writable so that the session data can be stored.

But wait! Before we make that writable, I have a better solution. Open up app/config/config.yml. Look under framework and session:

72 lines | app/config/config.yml

```
... lines 1 - 10
framework:
... lines 12 - 26
    session:
        # http://symfony.com/doc/current/reference/configuration/framework.html#handler-id
        handler_id:  session.handler.native_file
        save_path:   '%kernel.project_dir%/var/sessions/%kernel.environment%'
... lines 31 - 72
```

Ah! *This* is the reason why sessions are stored in var/sessions. Change that: set handler_id to ~. I'll add a comment: this means that the default PHP session handler will be used:

71 lines | app/config/config.yml

```
... lines 1 - 10
framework:
... lines 12 - 26
    session:
        # use the default PHP session handler
        handler_id: ~
... lines 30 - 71
```

Why are we doing this? Well, PHP *already* knows how to handle and store sessions. It will find a directory on the file system to store them and *it* will handle permissions... because making them 777 isn't a great idea. In fact, this will be the default setting for new Symfony 4 projects.

Go back to the local terminal. We just made a change to our *code*, so we need to commit and push:

```
$ git add -u
$ git commit -m "PHP native sessions"
$ git push origin master
```

Now, deploy!

```
$ ansible-playbook ansible/deploy.yml -i ansible/hosts.ini --ask-vault-pass
```

An even *better* session setup - especially if you want to move servers or use multiple servers - is to store them somewhere else, like the database or Memcache. You can find details about that in the Symfony docs. That's what we do for KnpU.

Ok! Let's try it again.. refresh! It works! OMG, it's alive! So... does this mean that the var/cache directory does *not* need to be writable? Well... not so fast. Go back to the server. Move up a few directories and into current/. Check out the var/cache/prod directory:

```
$ ls -l var/cache/prod
```

Woh! The cache files are writable by everyone! And so *of course* the site is working! But... we didn't set the cache directory to 777 in our playbook? So, what's going on?

We still have two unanswered questions. First, why the heck is var/cache/prod/ writable by everyone? And second, if we make it *not* writable, will our site still work?

Let's solve these mysteries next.

Chapter 16: Cache Permission Secrets

Why are the cache files writable by everyone? The answer is inside our code.

Open up bin/console. In my project, I uncommented a umask(0000) line:

27 lines | bin/console

```
... lines 1 - 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 - 27
```

I also added this in web/app.php:

27 lines | web/app.php

```
... lines 1 - 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 - 27
```

Thanks to this, whenever Symfony creates a file - like cache files - the permissions default to be writable by everyone.

No umask: Making Cache not Writable

I added these *precisely* to avoid permissions problems. But it's time to fix them properly. In app.php, comment that out:

27 lines | web/app.php

```
... lines 1 - 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 - 27
```

In console, comment it out... but also copy it and move it inside the debug if statement:

28 lines | bin/console

```
... lines 1 - 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 - 19
if ($debug) {
    umask(0000);
... line 22
}
... lines 24 - 28
```

During development, umask() makes our life really easy... cache files can be created and re-created by everyone. So I want to keep it. In fact, in web/app_dev.php, we also have a umask() call:

```
32 lines | web/app_dev.php
... lines 1 - 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 - 32
```

Again, this matches how Symfony 4 will work, out of the box.

Find your local terminal, commit those changes and push them:

```
$ git add -u
$ git commit -m "no writable in prod mode"
$ git push origin master
```

Deploy!

```
$ ansible-playbook ansible/deploy.yml -i ansible/hosts.ini --ask-vault-pass
```

Ok! Let's see what happens without umask on production. When it finishes, find your server terminal, move out of current/ and then back in. Check the permissions:

```
$ ls -la var/cache/prod
```

There it is! The files are writable by the user and group, but *not* by everyone. Our web server user - www-data - is *not* in the same group as our terminal user. Translation: the cache files are *not* writable by the web server.

So... will it blend? I mean, will the site work? Moment of Truth. Refresh! It *does* work! Woh! This is huge!

The Magical cache:warmup

How is this possible? How can the site work if our Symfony app can't write to the cache/ directory? The key is the cache:warmup task:

```
48 lines | ansible/deploy/after-symlink-shared.yml
... lines 1 - 14
- name: Warm up the cache
  command: '{{ release_console_path }} cache:warmup --env=prod'
... lines 17 - 48
```

I'm going to tell you a *small* lie first. The cache:warmup command creates *every* single cache file that your application will *ever* need. Thanks to this, the cache directory can *totally* be read-only after running this command.

Some Cache Cannot be Warmed Up

Great, right? Now, here is the *true* story. The cache:warmup task creates *almost* all of the cache files that you will ever need. But, there are a few types of things that simply *can't* be cached during warm up: they *must* be cached at the moment they're needed. These include the serializer and validation cache, for example.

Knowing this, our site works now, but it *should* break as soon as we try to use the serializer or validation system... because Symfony won't be able to cache their config. Well, let's try it!

I created an API endpoint: /api/videos that uses the serializer. Try it! Woh! It works! But... the serializer cache *shouldn't* be able to save. What the heck is going on?

The Dynamic cache.system Service

Here is the secret: whenever Symfony needs to cache something *after* cache:warmup, it uses a service called cache.system to do this:

```
$ ./bin/console debug:container cache.system
```

This is not a service you should use directly, but it's *critically* important.

Tip

Actually, you can use this service, but only to cache things that are needed to make your app work (e.g. config). It's cleared on each deploy

This service is special because it automatically tries *several* ways of caching. First, if APCu is available, it uses that. On the server, check for it:

```
$ php -i | grep apcu
```

Right now, we don't have that. No problem, the service then checks to see if OpCache is installed:

```
$ php -i | grep opcache
```

We *do* have this installed, and you should to. Thanks to it, instead of trying to write to the var/cache directory, Symfony uses temporary file storage and a super fast caching mechanism.

If neither APCu *nor* OpCache are installed, *then* it finally falls back to trying to write to the cache/ directory. So basically, in order for the cache directory to be read only... we don't need to do anything! Just, install OpCache - which you should *always* have - or APCu.

Great! But, I do have one more question: if we use APCu or OpCache, do we need to clear these caches when we deploy? For example, if some validation config was cached to APCu and that config is *changed*... don't we need to clear the old cache when we deploy? Actually, no! Each time you deploy, well, each time you run cache:warmup, Symfony chooses a new, random cache key to use for system.cache. This effectively clears the system cache on each deploy automatically!

This is a *long* way of saying that... well... the cache directory simply does *not* need to be writable. But, we *can* do a few things to improve performance!

Chapter 17: Optimizing with Cache

Yay! The var/cache directory does *not* need to be writable and our life is simple and all our cache dreams are fulfilled. Well actually, we can do more with caching to make our site *screaming* fast. Zoom!

APCu for Better Performance

First, let's install apcu - this should be a bit faster than OpCache. I'll do this during provision. Open up playbook.yml. Down a bit, yep! Add a new package: php-apcu:

172 lines | ansible/playbook.yml

```
---
- hosts: webserver
... lines 3 - 33
  tasks:
... lines 35 - 66
    - name: Install PHP packages
      become: true
      apt:
... lines 70 - 71
      with_items:
... lines 73 - 78
        - php-apcu
... lines 80 - 172
```

This package is named a bit different than the others.

Let's get the provision started - use playbook.yml and add -l aws to only provision the aws host:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini --ask-vault-pass -l aws
```

Use beefpass for the password.

Doctrine Metadata and Query Caching

There's one other way we can boost performance. Open app/config/config_prod.yml:

26 lines | app/config/config_prod.yml

```
... lines 1 - 3
#doctrine:
#    orm:
#        metadata_cache_driver: apc
#        result_cache_driver: apc
#        query_cache_driver: apc
... lines 9 - 26
```

See those Doctrine caches? Uncomment the first and third, and change them to apcu, which our server will *now* have! Woo!

```
26 lines | app/config/config_prod.yml
... lines 1 - 3
doctrine:
    orm:
        metadata_cache_driver: apcu
#        result_cache_driver: apc
        query_cache_driver: apcu
... lines 9 - 26
```

The metadata_cache_driver caches the Doctrine annotations or YAML mapping... this is not stuff we need to be parse on every request. The query_cache_driver is used when querying: it caches the translation from Doctrine's DQL into SQL. This is something that does not need to be done more than once.

So... now... I have the same question as before: do we need to clear this cache on each deploy? Nope! Internally, Symfony uses a cache *namespace* for Doctrine that includes the *directory* of our project. Since Ansistrano always deploys into a new releases/ directory, each deploy has its own, unique namespace.

When provisioning finishes, commit the new config changes:

```
$ git add -u
$ git commit -m "Doctrine caching"
```

Push them!

```
$ git push origin master
```

Now, deploy the site:

```
$ ansible-playbook ansible/deploy.yml -i ansible/hosts.ini --ask-vault-pass
```

While we're waiting, find the server again and run:

```
$ php -i | grep apcu
```

Yes! *Now* APCu is installed and working. Without doing *anything* else, Symfony's cache.system service is already using it. And when the deploy finishes, thanks to the Doctrine caching, we should have the fastest version of our site yet.

Except... for one more, fascinating cache issue. Actually, let's not call it a cache issue. Let's call it a cache opportunity!

Chapter 18: Priming cache.app

Watch closely: our production site is *super* slow! It takes a few *seconds* to load! What!? It's *especially* weird because, locally in the dev environment, it's way faster: just a few hundred milliseconds!

Why? Open src/AppBundle/Controller/DefaultController.php:

```
134 lines | src/AppBundle/Controller/DefaultController.php
... lines 1 - 9
class DefaultController extends Controller
{
... lines 12 - 14
    public function indexAction()
    {
... lines 17 - 20
        // Caching
        $uploadsItem = $this->getAppCache()->getItem('total_video_uploads_count');
        if (!$uploadsItem->isHit()) {
            $uploadsItem->set($this->countTotalVideoUploads());
            $uploadsItem->expiresAfter(60);
            // defer cache item saving
            $this->getAppCache()->saveDeferred($uploadsItem);
        }
        $totalVideoUploadsCount = $uploadsItem->get();

        $viewsItem = $this->getAppCache()->getItem('total_video_views_count');
        if (!$viewsItem->isHit()) {
            $viewsItem->set($this->countTotalVideoViews());
            $viewsItem->expiresAfter(60);
            // defer cache item saving
            $this->getAppCache()->saveDeferred($viewsItem);
        }
        $totalVideoViewsCount = $viewsItem->get();

        // save all deferred cache items
        $this->getAppCache()->commit();
... lines 42 - 48
    }
... lines 50 - 132
}
```

On the homepage, we show the total number of videos and the total number of video views. To get these, we first look inside a cache: we look for total_video_uploads_count and total_video_views_count. If they are *not* in the cache, *then* we calculate those and *store* them in the cache.

To calculate the number of videos, we call $this->countTotalVideoUploads():

```
134 lines | src/AppBundle/Controller/DefaultController.php
... lines 1 - 9
class DefaultController extends Controller
{
... lines 12 - 93
    /**
     * @return int
     */
    private function countTotalVideoUploads()
    {
        sleep(1); // simulating a long computation: waiting for 1s

        $fakedCount = intval(date('Hms') . rand(1, 9));

        return $fakedCount;
    }
... lines 105 - 132
}
```

That's a private method in this controller. It generates a random number... but has a sleep() in it! I added this to simulate a slow query. The countTotalVideoViews() *also* has a sleep():

```
134 lines | src/AppBundle/Controller/DefaultController.php
... lines 1 - 9
class DefaultController extends Controller
{
... lines 12 - 105
    /**
     * @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 117 - 132
}
```

So why is our site so slow? Because I put a sleep() in our code! I'm sabotaging us! But more importantly, for some reason, it seems like the cache system is failing. Let's find out why!

Hello cache.app

First, look at the getAppCache() method:

```
134 lines | src/AppBundle/Controller/DefaultController.php
... lines 1 - 9
class DefaultController extends Controller
{
... lines 12 - 125
    /**
     * @return AdapterInterface
     */
    private function getAppCache()
    {
        return $this->get('cache.app');
    }
}
```

To cache things, we're using a service called cache.app. This service is *awesome*. We already know about the system.cache service: an internal service that's used to cache things that make the site functional. The cache.app service is for *us*: we can use it to cache whatever we want! And unlike system.cache, it is *not* cleared on each deploy.

So why is this service failing? Because, by default, it tries to cache to the *filesystem*, in a var/cache/prod/pools directory:

```
$ ls var/cache/prod/pools
```

On production, we know that this directory is *not* writable. So actually, I'm surprised the site isn't broken! This service should not be able to write its cache!

Caching Failing is not Critical

To understand what's going on, lets mimic the issue locally. First, run:

```
$ bin/console cache:clear
```

This will clear and warm up the dev cache. Then, run:

```
$ sudo chmod -R 000 var/cache/dev/pools
```

Now, our *local* site won't be able to cache either.

Let's see what happens. Refresh! Huh... the site works... but it's *slow*. And the web debug toolbar is reporting a few warnings. Click to see those.

Woh! There are two warnings:

> Failed to save key total_video_uploads_count

and

> Failed to save key total_video_views_count

Of course! If caching *fails*, it's not fatal... it just makes our site slow. This is what's happening on production.

Let's fix the permissions for that directory:

```
$ sudo chmod -R 777 var/cache/dev/pools
```

Production Caching with Redis

So how can we fix this on production? We *could* make that directory writable, but there's a much better way: change cache.app to *not* use the file system! We already installed Redis during provision, so let's use that!

How? Open app/config/config.yml. Actually, use config_prod.yml, to only use this in production. Add framework, cache and app set to cache.adapter.redis:

30 lines | app/config/config_prod.yml

```
... lines 1 - 3
framework:
    cache:
        app: cache.adapter.redis
... lines 7 - 30
```

cache.adapter.redis is the id of a *service* that Symfony automatically makes available. You can also use cache.adapter.filesystem - which is the default - doctrine, apcu, memcached or create your own service. If you need to configure Redis, use the default_redis_provider key under app, set to redis:// and then your connection info:

```
# app/config/config_prod.yml
framework:
    cache:
        app: cache.adapter.redis
        default_redis_provider: redis://ConnectionInfo
```

There are similar config keys to control the other cache adapters.

Since we just changed our code, commit that change:

```
$ git add -u
$ git commit -m "using Redis cache"
```

Then, push and dance!

```
$ git push origin master
```

And then deploy!

```
$ ansible-playbook ansible/deploy.yml -i ansible/hosts.ini --ask-vault-pass
```

When the deploy finishes... try it! The first refresh should be slow: it's creating the cache. Yep... slow... Try again. Fast! Super fast! Our cache system is fixed!

Do Not Clear cache.app on Deploy

As we talked about earlier, the cache.system cache is effectively cleared on each deploy automatically. But cache.app is *not* cleared on deploy... and that's good! We're caching items that we do *not* want to automatically remove.

Actually... in Symfony 3.3, that's not true: when you run cache:clear, this *does* empty the cache.app cache. This is actually a *bug*, and it's fixed in Symfony 3.4. If you need to fix it for Symfony 3.3, open app/config/services.yml and override a core service:

```yaml
# app/config/services.yml
... lines 1 - 5
services:
... lines 7 - 36
    # Prevents cache.app from being cleared on cache:clear
    # this bug is fixed in Symfony 3.4
    cache.app_clearer:
        class: Symfony\Component\HttpKernel\CacheClearer\Psr6CacheClearer
```

The details of this aren't important, and if you're using Symfony 3.4 or higher, you don't need this.

Oh, and if you *do* want to clear cache.app, use:

```
$ bin/console cache:pool:clear cache.app
```

Chapter 19: Cleanup & GitHub OAuth Token

It's time to *polish* our deploy. Right now, you can surf to the /app_dev.php script on production. You can't really *access* it... but that file should not be deployed.

Back on the Ansistrano docs, look at the workflow diagram. So far, we've been hooking into "After Symlink Shared", because that's when the site is basically functional but not yet live. To delete app_dev.php, let's hook into "Before Symlink". It's basically the same, but this is the last opportunity to do something *right* before the deploy becomes live.

Scroll down to the variables section and copy ansistrano_before_symlink_tasks_file. In deploy.yml, paste that and set it to a new file: before-symlink.yml:

58 lines | ansible/deploy.yml

```
---
- hosts: aws
... lines 3 - 14
  vars:
... lines 16 - 48
    # Hooks: custom tasks if you need them
... line 50
    ansistrano_before_symlink_tasks_file: "{{ playbook_dir }}/deploy/before-symlink.yml"
... lines 52 - 58
```

In the deploy/ directory, create that! We only need one new task: "Remove sensitive scripts from web/ dir". Use the file module:

9 lines | ansible/deploy/before-symlink.yml

```
---
- name: Remove sensitive scripts from web/ dir
  file:
... lines 4 - 9
```

For path, first go back to deploy.yml, create a new variable release_web_path and set it to {{ ansistrano_release_path.stdout }}/web:

58 lines | ansible/deploy.yml

```
---
- hosts: aws
... lines 3 - 14
  vars:
... lines 16 - 18
    release_web_path: "{{ ansistrano_release_path.stdout }}/web"
... lines 20 - 58
```

Copy that variable and get back to work! Set path to {{ release_web_path }}/{{ item }}:

9 lines | ansible/deploy/before-symlink.yml

```
---
- name: Remove sensitive scripts from web/ dir
  file:
    path: '{{ release_web_path }}/{{ item }}'
... lines 5 - 9
```

We're *also* going to delete this config.php script:

423 lines | web/config.php

```
<?php

/*
 * ************** CAUTION **************
 *
 * DO NOT EDIT THIS FILE as it will be overridden by Composer as part of
 * the installation/update process. The original file resides in the
 * SensioDistributionBundle.
 *
 * ************** CAUTION **************
 */
... lines 12 - 423
```

Set state to absent and add with_items. Delete 2: app_dev.php and config.php:

9 lines | ansible/deploy/before-symlink.yml

```
---
- name: Remove sensitive scripts from web/ dir
  file:
    path: '{{ release_web_path }}/{{ item }}'
    state: absent
  with_items:
    - app_dev.php
    - config.php
```

Oh, and since I never deployed my services.yml change, let's commit these changes, push, and deploy to the cloud!

```
$ ansible-playbook ansible/deploy.yml -i ansible/hosts.ini --ask-vault-pass
```

Composer GitHub Access Token

While we're waiting, there is one thing that *could* break our deploy: GitHub rate limiting. If composer install accesses the GitHub API too often, the great and powerful GitHub monster will kill our deploy! This *shouldn't* happen, thanks to Composer's caching, but it *is* possible.

Tip

Actually, a change made to Composer in 2016 effectively fixed the rate limiting problem. But the fix (GitHub OAuth token) we will show will allow you to install dependencies from private repositories.

Google for "Composer GitHub token" to find a spot on their troubleshooting docs called API rate limit and OAuth tokens. All we need to do is create a personal access token on GitHub and then run this command on the server. This will please and pacify the GitHub monster, and the rate limiting problem will be gone.

Click the Create link and then "Generate new token". Think of a clever name and give it repo privileges.

Setting the GitHub Token in Ansible

Perfect! We *could* run the composer config command manually on the server. But instead, let's do it in our provision playbook: ansible/playbook.yml.

This is pretty easy... except that we *probably* don't want to hardcode my access token. Instead, we'll use the Ansible vault: a *new* vault just for playbook.yml. As soon as the deploy finishes, create it:

```
$ ansible-vault create ansible/vars/provision_vault.yml
```

Use the normal beefpass as the password. And then, add just one variable: vault_github_oauth_token set to the new access token:

```
# ansible/vars/provision_vault.yml

vault_github_oauth_token: 146f9e4f876164866d5afd956843d9141c4c6c47
```

Save and close! Whenever I have a vault, I also like to create a simple variables file. Create provision_vars.yml. And inside, set github_oauth_token to vault_github_oauth_token:

3 lines | ansible/vars/provision_vars.yml

```
---
github_oauth_token: "{{ vault_github_oauth_token }}"
```

Finally, in playbook.yml, let's include these! Include ./vars/provision_vault.yml and then ./vars/provision_vars.yml:

182 lines | ansible/playbook.yml

```
---
- hosts: webserver

  vars_files:
    - ./vars/provision_vault.yml
    - ./vars/provision_vars.yml
    - ./vars/vars.yml
... lines 8 - 182
```

We now have access to the github_oauth_token variable.

We have a few tasks that install the Composer executable:

```yaml
182 lines | ansible/playbook.yml
---
- hosts: webserver
... lines 3 - 35
  tasks:
... lines 37 - 100
    - name: Check for Composer
      stat:
        path: /usr/local/bin/composer
      register: composer_stat

    - name: Download Composer
      script: scripts/install_composer.sh
      when: not composer_stat.stat.exists

    - name: Move Composer globally
      become: true
      command: mv composer.phar /usr/local/bin/composer
      when: not composer_stat.stat.exists

    - name: Set permissions on Composer
      become: true
      file:
        path: /usr/local/bin/composer
        mode: "a+x"

    - name: Make sure Composer is at its latest version
      composer:
        working_dir: "/home/{{ ansible_user }}"
        command: self-update
      register: composer_self_update
      changed_when: "not composer_self_update.stdout|search('You are already using composer version')"
... lines 127 - 182
```

After those, create a new one: "Set GitHub OAuth token for Composer". Use the composer module and set command to config:

```
182 lines | ansible/playbook.yml
---
- hosts: webserver
... lines 3 - 35
  tasks:
... lines 37 - 100
    - name: Check for Composer
      stat:
        path: /usr/local/bin/composer
      register: composer_stat

    - name: Download Composer
      script: scripts/install_composer.sh
      when: not composer_stat.stat.exists

    - name: Move Composer globally
      become: true
      command: mv composer.phar /usr/local/bin/composer
      when: not composer_stat.stat.exists

    - name: Set permissions on Composer
      become: true
      file:
        path: /usr/local/bin/composer
        mode: "a+x"

    - name: Make sure Composer is at its latest version
      composer:
        working_dir: "/home/{{ ansible_user }}"
        command: self-update
      register: composer_self_update
      changed_when: "not composer_self_update.stdout|search('You are already using composer version')"

    - name: Set GitHub OAuth token for Composer
      composer:
        command: config
... lines 131 - 182
```

The docs show the full command we need. Copy the arguments and set arguments to that string. Replace the <oauthtoken> part with {{ github_oauth_token }}:

```
182 lines | ansible/playbook.yml
---
- hosts: webserver
... lines 3 - 35
  tasks:
... lines 37 - 127
    - name: Set GitHub OAuth token for Composer
      composer:
        command: config
        arguments: '-g github-oauth.github.com "{{ github_oauth_token }}"'
... lines 132 - 182
```

Also set working_dir to /home/{{ ansible_user }}... the composer module requires this to be set. And at the end, add a tag: github_oauth:

```
182 lines | ansible/playbook.yml
---
- hosts: webserver
... lines 3 - 35
  tasks:
... lines 37 - 127
    - name: Set GitHub OAuth token for Composer
      composer:
        command: config
        arguments: '-g github-oauth.github.com "{{ github_oauth_token }}"'
        working_dir: "/home/{{ ansible_user }}"
      tags:
        - github_oauth
... lines 135 - 182
```

Why the tag? Because I *really* don't want to re-run my *entire* provision playbook *just* for this task. Translation: I'm being lazy! Run the provision playbook, but with an extra -t github_oauth, just this one time:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini --ask-vault-pass -l aws -t github_oauth
```

Use beefpass! Great! So... is this working?

On GitHub, you can see that the token has *never* been used. When we deploy, composer install *should* now use it. But first, back on the server, run composer clear-cache:

```
$ composer clear-cache
```

to make sure it actually makes some API requests and doesn't just load everything from cache.

Now, deploy!

```
$ ansible-playbook ansible/deploy.yml -i ansible/hosts.ini --ask-vault-pass
```

As soon as this executes the "Composer install" task, our access key should be used. There it is... and yes! The key was used within the last week. Now we will never have rate limiting issues.

Chapter 20: Deploying to Multiple Servers

I just got great *great* news from our investors: the cows are practical *stampeding* to our site. They're watching our workout videos in herds. They're crying for mooooore!

Ahem.

But now, our site is starting to have issues! We need to go from one server to *multiple*. And guess what? We've already done most of the hard work to do this! Congrats!

Ready to make it happen? First, find your local terminal and use the aws.yml playbook to create a new EC2 instance:

```
$ ansible-playbook ./ansible/aws.yml -i ./ansible/hosts.ini --ask-vault-pass
```

This should only take a few second while it boots the server and waits for it to become available. Perfect!

Provision (Literally) all the Servers!

So why is it so easy to deploy to multiple servers? Because we've done *everything* via playbooks, and Ansible is *built* to work on many servers. Copy the public IP of the new server and open your ansible/hosts.ini file. Under the aws group, paste the new IP:

14 lines | ansible/hosts.ini

```
... lines 1 - 6
[aws]
54.80.32.36
54.81.150.15
... lines 10 - 14
```

Now, provision *both* servers by using the playbook.yml file. Add -l aws so we *only* provision those hosts:

```
$ ansible-playbook ./ansible/playbook.yml -i ./ansible/hosts.ini --ask-vault-pass -l aws
```

This won't make many changes to our existing server, but it *will* do *everything* to the new one. Translation... this may take awhile...

Tip

To speed things up, you could use your playbook to provision a server, and then create a custom image (AMI). For new servers, you can then boot from this image, to nearly-instantly get a provisioned server.

Dynamic Inventory

Oh, and you may have noticed that I hardcoded the IP address in my hosts.ini file:

14 lines | ansible/hosts.ini

```
... lines 1 - 6
[aws]
54.80.32.36
54.81.150.15
... lines 10 - 14
```

For a *truly* scalable architecture, you can use Ansible's Dynamic inventory, which works *extremely* well with EC2. You could, for example, use it to automatically use all EC2 instances with a specific *tag*.

Once the provision... finally... finishes.... let's deploy! Use deploy.yml:

```
$ ansible-playbook ./ansible/deploy.yml -i ./ansible/hosts.ini --ask-vault-pass
```

While that works, we need to talk about a few interesting things.

First, if you write the number pi to 2 decimal places, it spells the word pie backwards! Super interesting!

Second, Google for "Ansible serial" to find a spot on their docs called "Delegation, Rolling Updates, and Local Actions". On your playbook, Ansible allows you to set a serial option. If you have *many* servers, like 100, then if serial is set to 3, Ansible will deploy to only 3 servers at a time. The effect is a *rolling* deploy: your updated code reaches your 100 servers little-by-little. That's overkill for MooTube... well, at least today! But, it *is* a cool feature!

One *possible* issue with a rolling update involves the release directory name. You guys already know that each release is in a timestamped directory. In a serial deploy, that timestamp will be different on earlier servers versus later servers. For our app, that's no problem! But, if you used the directory name as part of some cache keys - like many apps do - then this *would* be a problem: different servers would be using different cache keys to get the same data.

In the Ansistrano docs, if you search for "serial", they mention this. By setting a variable, you can control the release directory name and make sure it's the same on all servers.

Preparing for Multiple Servers

Look back at the deploy. The database migrations *just* ran: for the first server, it reported "OK", because there were no migrations to run. But the second server *did* have migrations to run.

Wait... that's kinda weird... shouldn't all servers use the same database? Yep! There are a few things you need to do in order to be ready for multiple servers. The most obvious is that all your servers need to use the *same* database, or database cluster. For MooTube.com, each app is using their own *local* database. We're not going to fix that, but you *do* need to fix this in real life. The other really common thing you need to change is session storage: instead of storing sessions locally, you need store them in a shared place, like the database. In fact, that's the rule: when you have multiple servers, never store anything locally, except for temporary files or cache that help your code actually function.

Migrations & run_once

But wait... if all our servers shared one database... there would *still* be a problem! *Every* server would try to execute the migrations. That means that the same migration scripts would execute *multiple* times. Oh no!

Back in the Ansible docs, there is an option called run_once to fix this. It's pretty simple: if this is set, the task only runs on one server.

Ok! We are now deployed to *both* servers. Right now, mootube.example.com points directly to the *original* server. Copy the IP address to the new server. Then, open /etc/hosts and change mootube.example.com to point to that:

```
# /etc/hosts
# ...
#54.80.32.36 mootube.example.com
54.81.150.15 mootube.example.com
```

To test it, open a new Incognito Window to avoid caching and visit mootube.example.com. It works! Yes! We did *not* load the fixtures on the new server... which means we have a convenient way to know which server is being hit.

So, if you're using a playbook for provision and deploy, using multiple servers isn't a big deal. You *will* need to update your code a little bit - like to share session data - but almost everything is the same.

Next, let's go a step further and add a load balancer!

Chapter 21: Load Balancer & Reverse Proxy Setup

We have 2 servers! Yay! So... how do we configure things so that our users hit each server randomly? Why, a load balancer of course! Setting up a load balancer has nothing to do with Ansible, but let's take a quick tour anyways!

Creating an Elastic Load Balancer

I've already loaded up my EC2 Control Panel. Click "Load Balancers" on the left to create an "Elastic Load Balancer". To keep things simple, I'll create a "Classic" load balancer, but you should use an "Application Loader" balancer. That type is better, but a little more complex and beyond what I want to cover in this tutorial.

Give it a name - "MooTube-LoadBalancer" and make it respond only to HTTP traffic. You can also configure your load balancer to allow HTTPS traffic... which is *amazing*, because AWS can handle the SSL certificate automatically. Ultimately, the entire SSL process is resolved by the load balancer, and *all* requests - including *secure* requests - will be forwarded to our servers on port 80, as HTTP. This means we get HTTPS with basically no setup.

For the health check, keep index.html - I'll talk about why in a minute. I'm going to lower the interval and healthy threshold, but you can keep this: I'm only doing this so that the load balancer will see our new servers faster.

Finally, select the 2 running instances: "MooTube recording" is our original server - I renamed it manually - and "MooTube instance" is the new server we just launched.

Health Checks

Ok, create it! Look at the "Instances" tab: both servers are listed as "OutOfService". That's normal: it's testing to make sure the servers work! How does it test? By making a request to the IP address of each server, /index.html.

Go copy the IP address to one of the servers and try this! Woh! It's the "Welcome to Nginx" page from the default virtual host. *This* is why our health check will pass.

A *better* setup might be to make MooTube our *default* virtual host, so that you see the site *even* when you go to the IP address. That would be really nice because, right now, even though the health check will pass, it doesn't actually mean that MooTube is working on this server. It would be nicer to health check the *actual* app.

Go back to the "Instances" tab. Yes! Both instances are now "InService".

Testing the Load Balancer

So how can we test this? Every load balancer has a public DNS name. Copy that... then try it in a browser! Oh... it's that same "Welcome to Nginx" page! Our load balancer *is* sending our traffic to one of the servers... but since the host name is *not* mootube.example.com, we see the *default* virtual host.

In a real situation, we would configure our DNS to point to this load balancer. The Route 53 service in AWS let's you do this really easily. The tricky thing is that, as you can see, it does *not* list an IP address for the load balancer! What!? That's because the IP address might change at any time. In other words, you can rely on the DNS name, but not the IP address.

Since this is a fake site... we can't setup the DNS properly. So, to test this, we're going to cheat! Go to your terminal and ping the DNS name:

```
$ ping MooTube-ELB-Practice-21925007.us-east-1.elb.amazonaws.com
```

The ping will fail, but yes! There is the IP address to the load balancer. Like I said, do *not* rely on this in real life. But for temporary testing, it's fine! Edit your /etc/hosts file, and point this IP address to mootube.example.com:

```
# /etc/hosts
# ...
#54.80.32.36 mootube.example.com
#54.81.150.15 mootube.example.com
54.221.225.196 mootube.example.com
```

Ok, let's try it! Open a new Incognito window and go to http://mootube.example.com. Yes! It works! With no videos, this must be the new server! Refresh a few more times. I *love* it: you can see the load balancer is randomly sending us to one of the two servers.

Reverse Proxy & X-Forwarded-* Headers

Now that we're behind a load balancer... we have a new, minor, but important problem. Suppose that, in our app, we want to get the user's IP address. So, $request->getClientIp(). Guess what? That will now be the *wrong* IP address! Instead of being the IP of the user, it will *always* be the IP of the load balancer!

In fact, a *bunch* of things will be wrong. For example, $request->isSecure() will return false, even if the user is accessing our site over https. The port and even the host might be wrong!

This is a *classic* problem when you're behind a proxy, like a load balancer. When the load balancer sends the request back to our server, it *changes* a few things: the REMOTE_ADDR header is changed to be the *load balancer's* IP address. And if the original request was an https connection on port 443, the new request will *appear* insecure on port 80. That's because the load balancer handled the SSL stuff.

To help us, the load balancer sets the *original* information on a few headers: X-Forwarded-For holds the original IP address and X-Forwarded-Proto will be set to http or https.

Tip

There are some standards, but the exact headers used can vary from proxy to proxy.

This means that our app needs to be smart enough to read *these* headers, instead of the normal ones. Symfony doesn't do this automatically, because it could be a security risk. You need to configure it explicitly.

Setting Trusted Proxies

Google for "Symfony reverse proxy". Ok! In our front controller - so app.php in Symfony 3, we need to call setTrustedProxies() and pass it all possible IP addresses of our load balancer. Then, when a request comes into the app from a trusted IP address, Symfony knows it's safe to use the X-Forwarded headers and will use them automatically.

But... AWS is special... because we do *not* know the IP address of the load balancer! It's always changing! In that case, copy the second code block. Open web/app.php and - right after we create the request - paste it:

37 lines | web/app.php

```
... lines 1 - 22
$request = Request::createFromGlobals();

// Real client IP is under HTTP_X_FORWARDED_FOR for requests through AWS ELB,
// i.e. REMOTE_ADDR holds AWS ELB IP instead
Request::setTrustedProxies(
    // trust *all* requests
    ['127.0.0.1', $request->server->get('REMOTE_ADDR')],
    // if you're using ELB, otherwise see https://symfony.com/doc/current/request/load_balancer_reverse_proxy.html
    Request::HEADER_X_FORWARDED_AWS_ELB
);
... lines 33 - 37
```

Thanks to this code, we're going to trust *every* request that enters our app. Wait, what!? Doesn't that defeat the security mechanism? Yes! I mean... maybe! When you trust *all* proxies like this, you *must* configure your servers to *only* accept port 80 traffic from your load balancer. In other words, you need to configure your EC2 instances so that you *cannot* access them directly from the public web. The details of doing that are out of the scope of this tutorial. But once you've done this, then it *is* safe to trust *all* requests, because your load balancer is the *only* thing who can access your server.

Next! Let's get crazy, setup continuous integration, and auto-deploy our code after the tests pass! Nice!

Chapter 22: CircleCI: Auto-Deploy my Code!

This is not a tutorial about testing... but we couldn't resist! Our project actually *does* have a small test suite. Find your local terminal. To run them, execute:

```
$ ./vendor/bin/simple-phpunit
```

This is a wrapper around PHPUnit. It will install some dependencies the first time you try it and then... go tests go! They pass! Despite our best efforts, we haven't broken anything.

So here is my lofty goal: I want to configure our project with continuous integration on CircleCI and have CircleCI deploy *for* us, if the tests pass. Woh.

CircleCI Setup

In your browser, go to https://circleci.com and login. I'll make sure I'm under my own personal organization. Then go to projects and add a new project: our's is called ansistrano-deploy.

To use CircleCI, we will need a config.yml file. Don't worry about that yet! Live dangerously and just click "Start Building": this will activate a GitHub webhook so that each code push will automatically create a new CircleCI build. The power!

Actually, this starts our first build! But since we *don't* have that config.yml file yet, it's not useful.

Creating .circleci/config.yml

Head back to your editor. If you downloaded the "start" code for the course, you should have a tutorial/ directory with a circleci-config.yml file inside. To make CircleCI use this, create a new .circleci directory and paste it there: but call it just config.yml:

87 lines | .circleci/config.yml

```
version: 2
jobs:
  build_and_test:
    working_directory: ~/mootube
    docker:
      - image: php:7.1
      - image: mysql:5.7
        environment:
          MYSQL_ALLOW_EMPTY_PASSWORD: yes
    steps:
      # Installation
      - run:
          name: Install System Packages
          command: apt-get update && apt-get -y install git unzip zlib1g-dev
      - run:
          name: Install PHP Extensions
          command: docker-php-ext-install pdo pdo_mysql zip
      - run:
          name: Install Composer
          command: |
            php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');" && \
            php -r "if (hash_file('SHA384', 'composer-setup.php') === '544e09ee996cdf60ece3804abc52599c22b1f40f4323403c44d44fdfc
            php composer-setup.php && \
            php -r "unlink('composer-setup.php');" && \
            chmod +x ./composer.phar && \
            mv ./composer.phar /usr/local/bin/composer

      # Dependencies
      - checkout
      - restore_cache:
          key: mootube-{{ .Branch }}-{{ checksum "./composer.lock" }}-v1
      - run: composer install --prefer-dist --no-interaction
      # Force pulling Simple PHPUnit dependencies to be able to cache them as well
      - run: ./vendor/bin/simple-phpunit --version
      - save_cache:
          key: mootube-{{ .Branch }}-{{ checksum "./composer.lock" }}-v1
          paths:
            - '/root/.composer/cache'
            - './vendor'

      # Database
      - run: ./bin/console doctrine:database:create --env=test
      - run: ./bin/console doctrine:schema:create --env=test
      - run: ./bin/console hautelook_alice:doctrine:fixtures:load --no-interaction

      # To use server:start we need to install pcntl extension
      - run:
          name: Run web server in background
          command: ./bin/console server:run
          background: true

      # Testing
      - run: ./bin/console lint:yaml app/config
      - run: ./bin/console lint:twig app/Resources
      - run: ./vendor/bin/simple-phpunit

  deploy:
    working_directory: ~/mootube
    docker:
      - image: ansible/ansible:ubuntu1604
    steps:
      # Installation
      - run:
          name: Install System Packages
          command: pip install --upgrade pip && pip install ansible

      # Dependencies
      - checkout
      - restore_cache:
          key: mootube-{{ .Branch }}-{{ checksum "./ansible/requirements.yml" }}-v1
      - run: ansible-galaxy install -r ansible/requirements.yml
      - save_cache:
          key: mootube-{{ .Branch }}-{{ checksum "./ansible/requirements.yml" }}-v1
          paths:
            - '/root/.ansible/roles'

      # @TODO Deploy to AWS here...

workflows:
```

```
workflows:
  version: 2
  build_test_and_deploy:
    jobs:
      - build_and_test
      - deploy:
          requires:
            - build_and_test
```

We *will* talk about this file in a minute... but heck! Let's get crazy and just try it first! Back on your local terminal, add that directory and commit:

```
$ git add .circleci/
$ git commit -m "Adding CircleCI config"
```

Push wrecklessly to master! This should create a new build... there it is! It's build #7... because - to be *totally* honest - I was doing a bit of practicing before recording. I *usually* try to hide that... but I'm busted this time...

Anyways, click into the build. Ah, we're on some "Workflow" screen, and you can see two different builds: build_and_test and deploy.

Builds and Workflows in config.yml

Go back to config.yml. Under jobs, we have one called build_and_test:

87 lines | .circleci/config.yml

```
version: 2
jobs:
  build_and_test:
... lines 4 - 87
```

It sets up our environment, installs composer, configures the database and... eventually, runs the tests!

87 lines | .circleci/config.yml

```
version: 2
jobs:
  build_and_test:
    working_directory: ~/mootube
    docker:
      - image: php:7.1
      - image: mysql:5.7
        environment:
          MYSQL_ALLOW_EMPTY_PASSWORD: yes
    steps:
      # Installation
      - run:
          name: Install System Packages
          command: apt-get update && apt-get -y install git unzip zlib1g-dev
      - run:
          name: Install PHP Extensions
          command: docker-php-ext-install pdo pdo_mysql zip
      - run:
          name: Install Composer
          command: |
            php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');" && \
            php -r "if (hash_file('SHA384', 'composer-setup.php') === '544e09ee996cdf60ece3804abc52599c22b1f40f4323403c44d44fdf
            php composer-setup.php && \
            php -r "unlink('composer-setup.php');" && \
            chmod +x ./composer.phar && \
            mv ./composer.phar /usr/local/bin/composer

      # Dependencies
      - checkout
      - restore_cache:
          key: mootube-{{ .Branch }}-{{ checksum "./composer.lock" }}-v1
      - run: composer install --prefer-dist --no-interaction
      # Force pulling Simple PHPUnit dependencies to be able to cache them as well
      - run: ./vendor/bin/simple-phpunit --version
      - save_cache:
          key: mootube-{{ .Branch }}-{{ checksum "./composer.lock" }}-v1
          paths:
            - '/root/.composer/cache'
            - './vendor'

      # Database
      - run: ./bin/console doctrine:database:create --env=test
      - run: ./bin/console doctrine:schema:create --env=test
      - run: ./bin/console hautelook_alice:doctrine:fixtures:load --no-interaction

      # To use server:start we need to install pcntl extension
      - run:
          name: Run web server in background
          command: ./bin/console server:run
          background: true

      # Testing
      - run: ./bin/console lint:yaml app/config
      - run: ./bin/console lint:twig app/Resources
      - run: ./vendor/bin/simple-phpunit
... lines 56 - 87
```

But we also have a *second* job: deploy:

87 lines | .circleci/config.yml

```
version: 2
jobs:
... lines 3 - 56
  deploy:
... lines 58 - 87
```

The *whole* point of this job is to install Ansible and get ready to run our Ansistrano deploy. We're *not* actually doing this yet... but the environment should be ready:

87 lines | .circleci/config.yml

```
version: 2
jobs:
... lines 3 - 56
  deploy:
    working_directory: ~/mootube
    docker:
      - image: ansible/ansible:ubuntu1604
    steps:
      # Installation
      - run:
          name: Install System Packages
          command: pip install --upgrade pip && pip install ansible

      # Dependencies
      - checkout
      - restore_cache:
          key: mootube-{{ .Branch }}-{{ checksum "./ansible/requirements.yml" }}-v1
      - run: ansible-galaxy install -r ansible/requirements.yml
      - save_cache:
          key: mootube-{{ .Branch }}-{{ checksum "./ansible/requirements.yml" }}-v1
          paths:
            - '/root/.ansible/roles'

      # @TODO Deploy to AWS here...
... lines 78 - 87
```

The *real* magic is down below under workflows:

87 lines | .circleci/config.yml

```
... lines 1 - 78
workflows:
  version: 2
  build_test_and_deploy:
    jobs:
      - build_and_test
      - deploy:
          requires:
            - build_and_test
```

The *one* workflow lists both builds. *But*, thanks to the requires config, the deploy job will *only* run if build_and_test is successful. That's *super* cool.

Back on CircleCI, that job *did* finish successfully, and deploy automatically started. This *should* setup our Ansible-friendly environment... but it will *not* actually deploy yet.

CircleCI Environment Vars and the Vault Pass

It's time to fix that! In config.yml, under deploy, run the normal deploy command:
ansible-playbook ansible/deploy.yml -i ansible/hosts.ini --ask-vault-pass:

```
88 lines | .circleci/config.yml
version: 2
jobs:
... lines 3 - 56
  deploy:
... lines 58 - 76
      # Deploy
      - run: ansible-playbook ansible/deploy.yml -i ansible/hosts.ini --ask-vault-pass
... lines 79 - 88
```

And in theory... that's all we need! But... do you see the problem? Yep: that --ask-vault-pass option is *not* going to play well with CircleCI.

We need a different solution. Another option you can pass to Ansible is --vault-password-file that points to a *file* that holds the password. That's *better*... but how can we put the password in a file... without committing that file to our repository?

The answer! Science! Well yes, but more specifically, environment variables!

Back in CircleCI, configure the project. Find "Environment Variables" and add a new one called ANSIBLE_VAULT_PASS set to beefpass. Back in config.yml, before deploying, we can echo that variable into a file: how about ./ansible/.vault-pass.txt:

```
90 lines | .circleci/config.yml
version: 2
jobs:
... lines 3 - 56
  deploy:
... lines 58 - 76
      # Deploy
      - run: echo $ANSIBLE_VAULT_PASS > ./ansible/.vault-pass.txt
... lines 79 - 90
```

Use that on the next line: --vault-password-file= and then the path:

```
90 lines | .circleci/config.yml
version: 2
jobs:
... lines 3 - 56
  deploy:
... lines 58 - 76
      # Deploy
      - run: echo $ANSIBLE_VAULT_PASS > ./ansible/.vault-pass.txt
      - run: ansible-playbook ansible/deploy.yml -i ansible/hosts.ini --vault-password-file=./ansible/.vault-pass.txt
... lines 80 - 90
```

To be extra safe, delete it on the next line:

```
90 lines | .circleci/config.yml
version: 2
jobs:
... lines 3 - 56
  deploy:
... lines 58 - 76
      # Deploy
      - run: echo $ANSIBLE_VAULT_PASS > ./ansible/.vault-pass.txt
      - run: ansible-playbook ansible/deploy.yml -i ansible/hosts.ini --vault-password-file=./ansible/.vault-pass.txt
      - run: rm ./ansible/.vault-pass.txt
... lines 81 - 90
```

And... I'll fix my ugly YAML.

Setting Ansible Variables

Ok, problem solved! Time to deploy, right!? Well... remember how we added that prompt at the beginning of each deploy? Yep, that's going to break things too! No worries: Ansible gives us a way to set variables from the *command* line. When we do that, the prompt will *not* appear. How? Add a -e option with: git_branch=master:

90 lines | .circleci/config.yml

```
version: 2
jobs:
... lines 3 - 56
  deploy:
... lines 58 - 78
      - run: ansible-playbook ansible/deploy.yml -i ansible/hosts.ini -e "git_branch=master" --vault-password-file=./ansible/.vault-pass.txt
... lines 80 - 90
```

Disabling Host Key Checking

Ready to deploy... now!? Um... not so fast. Scroll up a little. Under the docker image, we need to add one environment variable: ANSIBLE_HOST_KEY_CHECKING set to no:

92 lines | .circleci/config.yml

```
version: 2
jobs:
... lines 3 - 56
  deploy:
    working_directory: ~/mootube
    docker:
      - image: ansible/ansible:ubuntu1604
        environment:
          ANSIBLE_HOST_KEY_CHECKING: no
... lines 63 - 92
```

Whenever you SSH to a machine for the first time, SSH prompts you to verify the fingerprint of that server. This disables that. If you have a highly sensitive environment, you may need to look into actually *storing* the fingerprints to your servers instead of just disabling this check.

Finally... I think we're ready! Go back to your local terminal, commit the changes, and push!

Adding ssh Keys

Go check it out. Ah, here is the new build: the build_and_test job starts off immediately. Let's fast-forward. But watch, when it finishes.... yes! Visually, you can see it activate the second job: deploy.

Inside this job, it sets up the environment first. When it starts running *our* tasks... woh! It fails! Ah:

> Failed to connect to host via ssh... no such identity... permission denied

Of course! CircleCI is trying to SSH onto our servers, but it does not have access. This works on our local machine because, when we deploy to the aws hosts, the group_vars/aws.yml file is loaded. This tells Ansible to look for the SSH key at ~/.ssh/KnpU-Tutorial.pem:

6 lines | ansible/group_vars/aws.yml

```
---
... line 2
ansible_ssh_private_key_file: ~/.ssh/KnpU-Tutorial.pem
... lines 4 - 6
```

That path does *not* exist in CircleCI.

So... Hmmm... We *could* leverage environment variables to create this file... but great news! CircleCI gives us an easier way. Open up the key file and copy all of its contents. Then, in CircleCI, configure the project and look for "SSH Permissions". Add a new one: paste the key, but leave the host name empty. This will tell CircleCI to use this key for all hosts.

We are ready! In CircleCI, I'll just click rebuild. It skips straight to the deploy job and starts setting up the environment. Then... yes! It's running our playbook! OMG, go tell your co-workers! The machines are deploying the site to the other machines! It takes a minute or two... but it *finishes*! CircleCI *just* deployed our site automatically.

There's no visible difference, but we are setup!

Next, let's talk about some performance optimizations that we need to make to our deploy.

Chapter 23: Optimizing Performance!

What about performance? Is our server optimized? Could our deploy somehow make our code faster? Why actually... yes!

Google for Symfony performance to find an article in the docs *all* about this.

Optimized Autoloader

Scroll down a little: I want to start at the section about Composer's Class Map. It turns out... autoloading classes - which happens *hundreds* of times on each request... is kinda slow! Composer even has its own documentation about optimizing the autoloader. Fortunately, making it fast is super simple: we just need to pass a few extra flags to composer install.

Open up the Ansible documentation for the composer module. It has an option called optimize_autoloader which actually *defaults* to true! In other words, thanks to the module, we're already passing the --optimize flag as well as the --no-dev flag.

The only missing option is --classmap-authoritative, which gives a minor performance boost. But hey, I *love* free performance!

Open up after-symlink-shared.yml and find the Composer install task. Add arguments set to --classmap-authoritative. I'll also set optimize_autoloader to true... but that's the default anyways:

```
50 lines | ansible/deploy/after-symlink-shared.yml
---
... lines 2 - 6
- name: Install Composer dependencies
  composer:
    command: install
    arguments: --classmap-authoritative
    optimize_autoloader: yes
    working_dir: '{{ ansistrano_release_path.stdout }}'
... lines 13 - 50
```

Oh, and there *is* one *other* way to optimize the autoloader: with a --apcu-autoloader flag. This is meant to be used instead of --classmap-authoritative... but I'm not sure if the performance will be much different. If you *really* care, you can test it, and let me know.

Guarantee OPCache

Back on the performance docs, at the top, the *first* thing it mentions is using a byte code cache... so OPcache. If you ignore *everything* else I say, at *least* make sure you have this installed. We already do. But, to be sure, we can open playbook.yml and - under the extensions - add php7.1-opcache:

183 lines | ansible/playbook.yml

```yaml
---
- hosts: webserver
... lines 3 - 35
  tasks:
... lines 37 - 68
    - name: Install PHP packages
      become: true
      apt:
        name: "{{ item }}"
        state: latest
      with_items:
... lines 75 - 79
        - php7.1-opcache
... lines 81 - 183
```

opcache.max_accelerated_files

Ok, what other performance goodies are there? Ah yes, opcache.max_accelerated_files. This defines how many files OPcache will store. Since Symfony uses a lot of files, we recommend setting this to a higher value.

On *our* server, the default is 10,000:

```
$ php -i | grep max_accelerated_files
```

But the docs recommend 20,000. So let's change it!

We already have some code that changes the date.timezone php.ini setting:

183 lines | ansible/playbook.yml

```yaml
---
- hosts: webserver
... lines 3 - 35
  tasks:
... lines 37 - 84
    - name: Set date.timezone for CLI
      become: true
      ini_file:
        path: /etc/php/7.1/cli/php.ini
        section: Date
        option: date.timezone
        value: UTC

    - name: Set date.timezone for FPM
      become: true
      ini_file:
        path: /etc/php/7.1/fpm/php.ini
        section: Date
        option: date.timezone
        value: UTC
      notify: Restart PHP-FPM
... lines 101 - 183
```

In that case, we modified *both* the cli *and* fpm config files. But because this is just for performance, let's only worry about FPM. Copy the previous task and create a new one called: Increase OPcache limit of accelerated files:

```yaml
192 lines | ansible/playbook.yml

---
- hosts: webserver
... lines 3 - 35
  tasks:
... lines 37 - 92
    - name: Set date.timezone for FPM
... lines 94 - 101
    - name: Increase OPcache limit of accelerated files
      become: true
      ini_file:
        path: /etc/php/7.1/fpm/php.ini
        section: opcache
... lines 107 - 108
      notify: Restart PHP-FPM
... lines 110 - 192
```

The section will be opcache. Why? On the server, open up the php.ini file and hit / to search for max_accelerated_files:

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

This is the setting we want to modify. And if you scroll up... yep! It's under a section called [opcache]:

```
# /etc/php/7.1/fpm/php.ini

# ...
[opcache]
# ...
; The maximum number of keys (scripts) in the OPcache hash table.
; Only numbers between 200 and 1000000 are allowed.
;opcache.max_accelerated_files=1000
```

Tell the ini_file module to set the option opcache.max_accelerated_files to a value of 20000:

```yaml
192 lines | ansible/playbook.yml

---
- hosts: webserver
... lines 3 - 35
  tasks:
... lines 37 - 101
    - name: Increase OPcache limit of accelerated files
      become: true
      ini_file:
        path: /etc/php/7.1/fpm/php.ini
        section: opcache
        option: opcache.max_accelerated_files
        value: 20000
      notify: Restart PHP-FPM
... lines 110 - 192
```

The Mysterious realpath_cache_size

There is just *one* last recommendation I want to implement: increasing realpath_cache_size and realpath_cache_ttl. Let's change them first... and explain later.

Go back to the php.ini file on the server and move *all* the way to the top. The standard PHP configuration all lives under a section called PHP:

```
# /etc/php/7.1/fpm/php.ini

[PHP]

;;;;;;;;;;;;;;;;;;;
; About php.ini   ;
;;;;;;;;;;;;;;;;;;;
# ...
```

If you looked closely enough, you would find out that the two "realpath" options *indeed* live here:

```
# /etc/php/7.1/fpm/php.ini

[PHP]
# ...
; Determines the size of the realpath cache to be used by PHP. This value should
; be increased on systems where PHP opens many files to reflect the quantity of
; the file operations performed.
; http://php.net/realpath-cache-size
;realpath_cache_size = 4096k

; Duration of time, in seconds for which to cache realpath information for a given
; file or directory. For systems with rarely changing files, consider increasing this
; value.
; http://php.net/realpath-cache-ttl
;realpath_cache_ttl = 120
```

Copy the previous task and paste. Oh, and I'll fix my silly typo. Name the new task "Configure the PHP realpath cache". This time, we want to modify *two* values. So, for option, use the handy {{ item.option }}. And for value, {{ item.value }}:

204 lines | ansible/playbook.yml

```
---
- hosts: webserver
... lines 3 - 35
  tasks:
... lines 37 - 101
    - name: Increase OPcache limit of accelerated files
... lines 103 - 110
    - name: Configure the PHP realpath cache
      become: true
      ini_file:
        path: /etc/php/7.1/fpm/php.ini
        section: PHP
        option: '{{ item.option }}'
        value: '{{ item.value }}'
      notify: Restart PHP-FPM
... lines 119 - 204
```

Hook this up by using with_items. Instead of simple strings, set the first item to an array with option: realpath_cache_size and value, which should be 4096K. Copy that and change the second line: realpath_cache_ttl to 600:

204 lines | ansible/playbook.yml

```yaml
---
- hosts: webserver
... lines 3 - 35
  tasks:
... lines 37 - 110
    - name: Configure the PHP realpath cache
      become: true
      ini_file:
        path: /etc/php/7.1/fpm/php.ini
        section: PHP
        option: '{{ item.option }}'
        value: '{{ item.value }}'
      notify: Restart PHP-FPM
      with_items:
        - { option: 'realpath_cache_size', value: '4096K' }
        - { option: 'realpath_cache_ttl', value: '600' }
... lines 122 - 204
```

All about the realpath_cache

We *did* make one small change to our deploy playbook, but let's just trust it works. The more interesting changes were to playbook.yml. So let's re-provision the servers:

```
$ ansible-playbook ansible/playbook.yml -i ansible/hosts.ini --ask-vault-pass -l aws
```

While that works, I want to explain all this realpath_cache stuff... because I don't think many of us *really* know how it works. Actually, Benjamin Eberlei even wrote a blog post about these settings. Read it to go deeper.

But here's the tl;dr: each time you require or include a file - which happens *many* times on each request - the "real path" to that file is cached. This is useful for symlinks: if a file lives at a symlinked location, then PHP figures out the "real" path to that file, then *caches* a map from the original, symlinked path, to the final, real path. That's the "Real Path Cache".

But even if you're *not* using symlinks, the cache is *great*, because it prevents IO operations: PHP does not even need to *check* if the path is a symlink, or get other information.

The point is: the realpath cache rocks and makes your site faster. And that's *exactly* why we're making sure that the cache is big enough for the number of files that are used in a Symfony app.

The realpath_cache_ttl is where things get *really* interesting. We're using a symlink strategy in our deploy. And *some* sources will tell you that this strategy plays *really* poorly with the "realpath cache". Why? Well, just think about: suppose your web/app.php file requires app/AppKernel.php. Internally. app.php will ask:

> Hey Realpath Cache! What is the *real* path to /var/www/project/current/app/AppKernel.php?

If we've recently deployed, then that path *may* already exist in the "realpath cache"... but still point to the *old* release directory! In other words, the "realpath cache" will continue to think that a bunch of our files still *truly* live in the *old* release directory! This will happen until all the cache entries hit their TTL.

So here's the big question: if the "realpath cache" is such a problem... then why are we *increasing* the TTL to an even *higher* value!?

Because... I lied... a little. Let me show you why the "realpath cache" is *not* a problem. On your server, open up /etc/nginx/sites-available/mootube.example.com.conf. Search for "real". Ah yes:

```
# /etc/nginx/sites-available/mootube.example.com.conf

# ...
location ~ ^/(app_dev|config)\.php(/|$) {
    # ...
    fastcgi_param SCRIPT_FILENAME $realpath_root$fastcgi_script_name;
    # ...
}
```

This line helps pass information to PHP. The *key* is the $realpath_root part. Thanks to this, by the time PHP executes, our code - like web/app.php *already* knows that it is in a releases directory. That means that when it tries to require a file - like app/AppKernel.php, it *actually* says:

> Hey Realpath Cache! What is the *real* path to /var/www/project/releases/2017XXXXX/app/AppKernel.php?

The *symlink* directory - current/ is *never* included in the "realpath cache"... because our own code thinks that it lives in the resolved, "releases" directory. This is a long way of explaining that the "realpath cache" just works... as long as you have this line.

Check on the provision. Perfect! It just finished updating the php.ini file. Go check that out on the server and look for the changes. Yep! max_accelerated_files looks perfect... and so do the realpath settings:

```
# /etc/php/7.1/fpm/php.ini

# ...
opcache.max_accelerated_files = 20000
# ...
realpath_cache_size = 4096K
# ...
realpath_cache_ttl = 600
# ...
```

Next! Let's talk about rolling back a deploy. Oh, we of course *never* make mistakes... but... ya know... let's talk about rolling back a deploy anyways... in case someone *else* messes up.

Chapter 24: When things go wrong: Rollback

Yea... things go wrong. But, but, but! Ansistrano is *cool* because *if* something fails, the symlink will never change and the site will *continue* using the old code. If you're following our "Safe Migrations" philosophy, then deploying is even *safer*: if your migrations run, but the deploy never finishes, your non-destructive migrations won't hurt the current code.

But sometimes... a deploy *will* finish... only for you to have the sudden, horrible realization, that something is now *massively* wrong with the site. Like, there are zombies all over it or something.

At this moment, fate presents you with 3 options:

1. Run for your life!
2. Quickly make a new commit to fix things and re-deploy.
3. Rollback to the previous deploy.

Other than running out of the building screaming, rolling back is the *fastest* way to escape the problem. And fortunately, Ansistrano has a *second* role all about... rolling back: ansistrano.rollback.

To install it, open the requirements.yml file and add an entry. For the version... let's see. The latest version right now is 2.0.1. Let's use that:

9 lines | ansible/requirements.yml

```
... lines 1 - 6
- src: ansistrano.rollback
  version: 2.0.1
```

Tip

Due to the changes in Ansible Galaxy, Ansistrano is installed now via ansistrano.rollback instead of the old carlosbuenosvinos.ansistrano-rollback.

To install the role, on your local terminal, run:

```
$ ansible-galaxy install -r ansible/requirements.yml
```

Creating the Rollback Playbook

The rollback process will be its own, simple playbook. Create it: rollback.yml. I'll open deploy.yml so we can steal things... starting with the host. Then, of course, we need to include the new role: ansistrano.rollback:

6 lines | ansible/rollback.yml

```
---
- hosts: aws

  roles:
    - ansistrano.rollback
```

Rolling back is *way* simpler than deploying, but it works in the same way: there are a few *stages* and we override variables to control things. The *only* variable we *need* to override is ansistrano_deploy_to. In deploy.yml, we imported a vars_files called vars.yml, and used it to help set this.

Let's do basically the same thing here. Copy part of the vars_files section, paste it, and just import vars.yml: we don't need the vault:

13 lines | ansible/rollback.yml

```yaml
---
- hosts: aws

  vars_files:
    - ./vars/vars.yml
... lines 6 - 13
```

Back in deploy.yml, also steal ansistrano_deploy_to and add that to rollback.yml:

13 lines | ansible/rollback.yml

```yaml
---
- hosts: aws

  vars_files:
    - ./vars/vars.yml

  vars:
    # Ansistrano vars
    ansistrano_deploy_to: "{{ project_deploy_dir }}" # Base path to deploy to.
... lines 10 - 13
```

Rollback!

And... yea... that's basically it! So... let's try it! On the server, I'm already in /var/www/project. My current symlink is set, and releases has 3 directories inside.

Back on your local terminal... rollback!

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

That's it. It should only take a few seconds. It *does* delete the old release directory, but this - like most things - can be controlled with a variable.

Done! Back on the server... yes! The symlink *changed*. And one of our releases is gone!

Running Down Database Migrations

So rolling back is pretty easy. The most *common* issue involves migrations. Again, if you follow our "safe migrations" philosophy, you have nothing to worry about. But, if you're a bit more reckless - hey, no judgment - then you may need to manually run the *down* direction on some of your migrations after a rollback.

Let's add a little "opportunity" for us to do that. Let me show you: copy the ansistrano_before_symlink_tasks_file variable. In rollback.yml, paste this and set it to a new rollback/before-symlink.yml:

16 lines | ansible/rollback.yml

```yaml
---
- hosts: aws
... lines 3 - 6
  vars:
... lines 8 - 10
    # Hooks: custom tasks if you need them
    ansistrano_before_symlink_tasks_file: '{{ playbook_dir }}/rollback/before-symlink.yml'
... lines 13 - 16
```

Now, create a new rollback/ directory with that file inside. Here, we'll add just one task: "Pause to run migrations manually down". Use the pause module to *freeze* the playbook and put up a message:

5 lines | ansible/rollback/before-symlink.yml

```
---
- name: Pause to run migrations manually down
  pause:
    prompt: "Please, run 'bin/console doctrine:migrations:execute YYYYMMDDHHMMSS --down' manually in '{{ ansistrano_release_path
```

This is *our* opportunity to manually execute any *down* migrations we want.

Hey, I know: it's not automated and it's not awesome. But, things have gone wrong, so it's time for us to take over.

Let's rollback *one* more time:

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

Here's the pause: it shows us the directory we should go into. Hit enter and it keeps going. Oh, and cool! It *failed* on one of the servers! That was unplanned... but awesome! That's the new server, and apparently we've only deployed there *two* times. So, there was *no* old version to rollback to. Ansistrano smartly prevented the rollback... instead of rolling back to nothing.

Ok guys... we're done! Woh! This tutorial was a *crazy* ride - I *loved* it! And I hope you did too. You can now deploy a *killer* Symfony site - or *any* site with Ansistrano. If this was all interesting but felt like a lot of work, don't forget about the platform-as-a-service options like Heroku or Platform.sh. You don't have *quite* as much flexibility, and they're sometimes a bit more expensive, but a lot of what we learned is handled for you and you can get started *really* quickly.

Whatever you choose, go forth and build something awesome! Ok guys, I'll seeya next time!