Technology Conversations

Continuous Integration, Delivery or Deployment with Jenkins, Docker and Ansible

This article tries to provide one possible way to set up the Continuous Integration, Delivery or Deployment pipeline. We'll use Jenkins, Docker, Ansible and Vagrant to set up two servers. One will be used as a Jenkins server and the other one as an imitation of production servers. First one will checkout, test and build applications while perform deployment and post-deployment tests.

You'll need Vagrant and Git installed. The rest of the tools will be set up as part of the exercises in this article.

CI/CD Environment

We'll set up Jenkins environment using Vagrant and Ansible. Vagrant will create a VM with Ubuntu and run the bootstrap.sh script. The only purpose of that script is to install Ansible. Once that is done, Ansible will make sure that Docker is installed and Jenkins process is running.

As everything else in this article, Jenkins itself is packed as a Docker container and deployed with Ansible. Please consult the Continuous Deployment: Implementation with Ansible and Docker article for more info.

If you haven't already, please clone the GitHub repo jenkins-docker-ansible. Once repo is cloned, we can fire up Vagrant for the **cd** machine.

If you run into issues with Ansible complaining about executable permissions, try modifying the `Vagrantfile`'s `synced_folder` entry from `config.vm.synced_folder ".", "/vagrant"` to `config.vm.synced_folder ".", "/vagrant", mount_options: ["dmode=700,fmode=600"]`. You'll find an example in the Vagrantfile.

```
1 git clone https://github.com/vfarcic/jenkins-docker-ansible.git
2 cd jenkins-docker-ansible
3 vagrant up cd
```

This might take a while when run for the first time (each consecutive run will be much faster) so let's use this opportunity to go through the setup while waiting for the creation of VM and configuration to finish.

Two key lines in the Vagrantfile are:

```
...
cd.vm.provision "shell", path: "bootstrap.sh"
cd.vm.provision :shell, inline: 'ansible-playbook /vagrant/ansible/cd.yml -c lo
...
```

First one runs the bootstrap.sh script that installs Ansible. We could use the Vagrant Ansible Provisioner but that would require Ansible to be installed on the host machine. That is unnecessary dependency, especially for Windows users who would have a hard time to set up Ansible. Moreover, we'll need Ansible inside the VM to perform deployment from the **cd** to **prod** VM.

Once bootstrap.sh is executed, Ansible playbook cd.yml is run.

```
- hosts: localhost
  remote_user: vagrant
  sudo: yes
  roles:
    - java
    - docker
    - registry
    - jenkins
```

It will run java, docker, registry and jenkins roles. Java is the Jenkins dependency required for running slaves. Docker is needed for building and running containers. All the rest will run as Docker processes. There will be no other dependency, package or application that will be installed directly. Registry role runs Docker Registry. Instead of using the public one in hub.docker.com, we'll push all our containers to the private registry running on port 5000. Finally, jenkins role is run. This one might require a bit more explanation.

Here's the list of tasks in the jenkins role.

```
- name: Directories are present
  file: path="{{ item }}" state=directory
  with_items: directories

- name: Config files are present
  copy: src='{{ item }}' dest='{{ jenkins_directory }}/{{ item }}'
  with_items: configs

- name: Plugins are present
  get_url: url='https://updates.jenkins-ci.org/{{ item }}' dest='{{ jenkins_di
  with_items: plugins

- name: Build job directories are present
  file: path='{{ jenkins_directory }}/jobs/{{ item }}' state=directory
  with_items: jobs

- name: Build jobs are present
  template: src=build.xml.j2 dest='{{ jenkins_directory }}/jobs/{{ item }}/con
  with_items: jobs

- name: Deployment job directories are present
  file: path='{{ jenkins_directory }}/jobs/{{ item }}-deployment' state=directo
  with_items: jobs

- name: Deployment jobs are present
  template: src=deployment.xml.j2 dest='{{ jenkins_directory }}/jobs/{{ item }
  with_items: jobs
```

```
28
29  - name: Container is running
30    docker: name=jenkins image=jenkins ports=8080:8080 volumes=/data/jenkins:/va
31
32  - name: Reload
33    uri: url=http://localhost:8080/reload method=POST status_code=302
34    ignore_errors: yes
```

First we create directories where Jenkins plugins and slaves will reside. In order to speed up building containers, we're also creating the directory where ivy files (used by SBT) will be stored on host. That way containers will not need to download all dependencies every time we build Docker containers.

Once directories are created, we copy Jenkins configuration files and few plugins.

Next are Jenkins jobs. Since all jobs are going to do the same thing, we have two templates (**build.xml.j2** and **deployment.xml.j2**) that will be used to create as many jobs as we need.

Finally, once Jenkins job files are in the server, we are making sure that Jenkins container is up and running.

Full source code with Ansible Jenkins role can be found in the jenkins-docker-ansible repository.

Let's go back to Jenkins job templates. One template is for building and the other one for deployment. Build jobs will clone the code repository from GitHub and run few shell commands.

Following is the key part of the build.xml.j2 template:

```
1  sudo docker build -t 192.168.50.91:5000/{{ item }}-tests docker/tests/
2  sudo docker push 192.168.50.91:5000/{{ item }}-tests
3  sudo docker run -t --rm -v $PWD:/source -v /data/.ivy2:/root/.ivy2/cache 192.168
4  sudo docker build -t 192.168.50.91:5000/{{ item }} .
5  sudo docker push 192.168.50.91:5000/{{ item }}
```

Each **{{ item }}** from above will be replaced with values from Ansible variables. Since all build jobs will do the same procedure, we can use the same template for all of them and simply provide a list of values. In this article, variables from the **main.yml** are following:

```
1  jobs:
2    - books-service
```

When Ansible is run, each **{{ item }}** will be replaced with **books-service**. **jobs** variable could have as many items as we need. They don't need to be added all at once but gradually according to our needs.

Later on it could look something like:

```
1  jobs:
2    - books-service
3    - authentication-service
4    - shopping-cart-service
5    - books-ui
```

Commands from the template, when deployed with Ansible, are following.

```
sudo docker build -t 192.168.50.91:5000/books-service-tests docker/tests/
sudo docker push 192.168.50.91:5000/books-service-tests
sudo docker run -t --rm -v $PWD:/source -v /data/.ivy2:/root/.ivy2/cache localh
sudo docker build -t 192.168.50.91:5000/books-service .
sudo docker push 192.168.50.91:5000/books-service
```

First we build the test container and push it to the private registry. Then we runs tests. If there are no failures, we'll build the books-service container and push it to the private registry. From here on, books-service is tested, built and ready to be deployed.

Before Docker, all my Jenkins servers ended up with a huge number of jobs. Many of them were different due to variety of frameworks, languages and libraries required to build all the applications. Managing a lot of different jobs easily becomes very tiring and prone to errors. And it's not only jobs that become complicated very fast. Managing slaves and dependencies they need to have often requires a lot of time.

With Docker comes simplicity. If we can assume that each project will have its own tests and application containers, all jobs can do the same thing. Build the test container and run it. If nothing fails, build the application container and push it to the registry. Finally, deploy it. All projects can be exactly the same if we can assume that each of them have their own Docker files. Another advantage is that there's nothing to be installed on servers (besides Docker). All we need is Docker that will run containers.

Unlike build jobs that are always the same (build with the specification from Dockerfile), deployments tend to get a bit more complicated. Even though applications are immutable and packed in containers, there are still few environment variables, links and/or volumes to be set. That's where Ansible comes in handy. We can have every Jenkins deployment job the same with only name of the Ansible playbook differing. Deployment jobs simply run Ansible role that corresponds to the application we're deploying. It's still fairly simple in most cases. The difference when compared to deploying applications without Docker is huge. While with Docker we need to think only about data (application and all dependencies are packed inside containers), without it we would need to think what to install, what to update and how those changes might affect the rest of applications running on the same server or VM. That's one of the reasons why companies tend not to change their technology stack and, for example, still stick with Java 5 (or worse).

As an example, books-service Ansible tasks are listed below.

```
- name: Directory is present
  file:
    path=/data/books-service/db
    state=directory

- name: Latest container is pulled
  shell: sudo docker pull 192.168.50.91:5000/books-service

- name: Container is absent
  docker:
    image=192.168.50.91:5000/books-service
```

```
    name=books-service
    state=absent

- name: Container is running
  docker:
    name=books-service
    image=192.168.50.91:5000/books-service
    ports=9001:8080
    volumes=/data/books-service/db:/data/db
    state=running
```

We're making sure that directory where data will be stored is present, pulling the latest version of the container, removing the running process and starting the new one.

Let's get back to the **cd** VM we started creating at the beginning of this article! If **vagrant up cd** command finished executing, whole VM with Jenkins, Docker and Registry is up and running.

Now we can open http://localhost:8080 and (almost) use Jenkins. Ansible tasks did not create credentials so we'll have to do that manually.

- Click **Manage Jenkins** > **Manage Nodes** > **CD** > **Configure**.
- Click **Add** button in the **Credentials** Section.
- Type **vagrant** as both username and password and click the **Add** button.
- Select the newly created key in the **Credentials** section.
- Click the **Save** and, finally, the **Launch slave agent** buttons

This could probably be automated as well but, for security reasons, I prefer doing this step manually.

Now the CD slave is launched. It's pointing to the **cd** VM we created with Vagrant and will be used for all our jobs (even for deployments that will be done on a separate machine).

We are ready to run the books-service job that was explained earlier. From the Jenkins home page, click **books-service** link. First build already started (can be started manually by pressing **Build Now**. Progress can be seen in the **Build History** section. **Console Output** inside the build (in this case #1) can be used to see logs. Building Docker containers for the first time can take quite some time. Once this job is finished it will run the **books-service-deployment** job. However, we still don't have the production environment VM and the Ansible playbook run by the Jenkins job will fail to connect to it. We'll get back to this soon. At the moment we're able to checkout the code, run tests, build Docker containers and push them to the private registry.

Major advantages to this kind of setup is that there is no need to install anything besides Docker on the **cd** server since everything is run through containers. There will be no headache provoked by installations of all kinds of libraries and frameworks required for compilation and execution of tests. There will be no conflicts between different versions of the same dependency. Finally, Jenkins jobs are going to be very simple since all the logic resides in Docker files in the repositories of applications that should be built, tested and deployed. In other words, simple and painless setup that will be easy to maintain no matter how many projects/applications Jenkins will need to manage.

If naming conventions are used (as in this example), creating new jobs is very easy. All there is to be done is to add new variables to the Ansible configuration file **ansible/roles/jenkins/defaults/main.yml** and run **vagrant provision cd** or directly **ansible-playbook /vagrant/ansible/cd.yml -c local** from the CD VM.

Here's how to apply changes to the CD server (includes adding new Jenkins jobs).

[from the host directory where this repo is cloned]

```
vagrant provision cd
```

or

```
vagrant ssh cd
ansible-playbook /vagrant/ansible/cd.yml -c local
exit
```

books-service job is scheduled to pull the code from the repository every 5 minutes. This consumes resources and is slow. Better setup is to have a GitHub hook. With it the build would be launched almost immediately after each push to the repository. More info can be found in the GitHub Plugin page. Similar setup can be done for almost any other type of code repository.

Production Environment

In order to simulate closer to reality situation, production environment will be a separate VM. At the moment we don't need anything installed on that VM. Later on, Jenkins will run Ansible that will make sure that the server is set up correctly for each application we deploy. We'll create this environment in the same way as the previous one.

[from the host directory where this repo is cloned]

```
vagrant up prod
```

Unlike the **cd** VM that required setup, **prod** has only the Ubuntu OS. Packages and additional dependencies are not required.

Now, with the **prod** environment up and running, all that's missing is to generate SSH keys and import them to the **cd** VM.

[from the host directory where this repo is cloned]

```
vagrant ssh prod
ssh-keygen # Simply press enter to all questions
exit
vagrant ssh cd
ssh-keygen # Simply press enter to all questions
ssh-copy-id 192.168.50.92 # Password is "vagrant"
exit
```

That's about it. Now we have an production VM where we can deploy applications. We can go back to Jenkins (http://localhost:8080) and run the **books-service-deployment** job. If, the **books-service** job did not finish before you reached this part, please wait until it's over and **books-service-deployment** will start automatically. When finished, service will be up and running on the port 9001.

Let's put few entries to our recently deployed **books-service**

[from the host directory where this repo is cloned]

```
vagrant ssh prod
curl -H 'Content-Type: application/json' -X PUT -d '{"_id": 1, "title": "My Firs
curl -H 'Content-Type: application/json' -X PUT -d '{"_id": 2, "title": "My Sec
curl -H 'Content-Type: application/json' -X PUT -d '{"_id": 3, "title": "My Thi
exit
```

Let's check whether the service returns correct data. Open http://localhost:9001/api/v1/books from your browser. You should see the three books that were previously inserted with **curl**.

Our service has been deployed and is up and running. Every time there is a change in the code, the same process will be repeated. Jenkins will clone the code, run tests, build the container, push it to the registry and, finally, run that container in the destination server.

VM creation, provisioning, building and deploying them took a lot of time. However, from now on most of the things (Docker images, IVY dependencies, etc) are already downloaded so each next run will be very fast. Only new Docker images will be created and pushed to the registry. From this moment on, speed is what matters.

Summary

With Docker we can explore new ways to build, test and deploy applications. One of the many benefits of containers is simplicity due to their immutability and self sufficiency. There are no reasons any more to have servers with huge number of packages installed. No more going through the hell of maintaining different versions required by different applications or spinning up a new VM for every single application that should be tested or deployed.

But it's not only servers provisioning that got simplified with Docker. Ability to provide Docker file for each application means that Jenkins jobs are much easier to maintain. Instead of having tens, hundreds or even thousands of jobs where each of them is specific to the application it is building, testing or deploying, we can simply make all (or most of) Jenkins jobs the same. Build with the Dockerfile, test with the Dockerfile and, finally, deploy Docker container(s) with Ansible (or some other tool like Fig).

We didn't touch the subject of post-deployment tests (functional, integration, stress, etc) that are required for successful Continuous Delivery and/or Deployment. We're also missing the way to deploy the application with zero-downtime. Both will be the subject of one of the next articles. We'll

continue where we left and explore in more depth what should be done once the application is deployed.

Source code for this article can be found in jenkins-docker-ansible repository.

This entry was posted in Continuous Integration, Delivery and Deployment, Docker and tagged Ansible, CD, CI, Continuous delivery, Continuous integration, Continuous Integration, Delivery and Deployment, Docker, jenkins, Vagrant on February 11, 2015 [http://technologyconversations.com/2015/02/11/continuous-integration-delivery-or-deployment-with-jenkins-docker-and-ansible/] by Viktor Farcic.

---

41 thoughts on “Continuous Integration, Delivery or Deployment with Jenkins, Docker and Ansible”

Pingback: Continuous Deployment: Implementation with Ansible and Docker | Technology

eyalgo
March 11, 2015 at 7:21 am

Hi,
I am trying to work with the tutorial on windows environment.
When I run “vagrant up cd” , it is stuck with:
“cd: Verifying Hyper-V is enabled…”

should I install anything else prior to running vagrant?

Thanks

Viktor Farcic Post author
March 11, 2015 at 12:51 pm

Sorry… I forgot to directly answer your question. The only prerequisite to running Vagrant (besides VMs being supported by your BIOS) is VirtualBox (even though it can run with VMWare as well).

eyalgo
March 14, 2015 at 11:04 pm

Hi,
Thanks for helping out.
Yes, I now understand that in order to use Vagrant, I need to have VB installed.

Thanks

Viktor Farcic Post author
March 11, 2015 at 1:02 pm

Please take a look at http://docs.vagrantup.com/v2/hyperv/index.html. It might help.

Viktor Farcic Post author
March 11, 2015 at 12:50 pm

I had a similar problem long time ago and it was fixed by enabling VMs in BIOS. I'm not sure whether that's the cause of what's happening in your case but that's where I'd start. You might also import any 64 bits box and try it out (http://docs.vagrantup.com/v2/getting-started/boxes.html). Just bare bones OS to see whether VMs can be created...

Liping Huang
April 15, 2015 at 8:58 am

Awesome!!! This is really a great article, Thanks.

And a question is about the "Production Environment" section, in this section, there is only little character to describe the deploy book-service, I means what exact the job "books-service-deployment" do.

Viktor Farcic Post author
April 15, 2015 at 10:56 am

Deployment used in this article is very simplified with intention only to provide a very basic understanding of how Ansible works. It doesn't The code used to deploy books-service is in https://github.com/vfarcic/jenkins-docker-ansible/blob/master/ansible/roles/books-service/tasks/main.yml. I've been working on a much more elaborated example that I presented in http://www.boosterconf.no/talks/413 and will present soon in http://craft-conf.com/2015#workshops/ViktorFarcic. You can find presentation in http://vfarcic.github.io/cd-workshop/ and the source code in https://github.com/vfarcic/cd-workshop. I have pending to write a set of articles (at least 5) that would guide people from the begining to the end of the process. However, I'm currently too ocupied writing a book on TDD, hosting workshops and moving to a new company in Silicon Valley. In other words, I have too many things on my plate and those articles will probably need to wait for 2 months. Sorry for that

While writing articles takes a lot of time I can do the workshop that goes through the whole process online without charge. If you organize, let's say, at least 10 people, I'll be happy to do a few hours session. You can contact me via email if you're interested (you'll find it in the "about" section).

Fernando Cruz
July 24, 2015 at 1:13 am

Hi, I am very interested in the article, but I can figure the role of Vagrant in the picture. Could you help me?

Viktor Farcic Post author
July 24, 2015 at 1:19 am

I'm using Vagrant in this article as a convenient way to set up multiple servers. If you already have two (or more) servers you can use, there is no need for Vagrant. In "real projects", I do not use Vagrant but deploy directly to servers. Does this answer your question? I might have misunderstood...

Milan Simonovic
August 4, 2015 at 12:02 am

trying to read the code but can't figure out how come the build job can execute docker commands, when is docker installed inside the Jenkins container? The image is vfarcic/jenkins, suppose it's this one https://github.com/vfarcic/jenkins-docker? Looking at the Dockerfile, there's no docker client installed, nor is any of the docker jenkins plugins installed (https://github.com/vfarcic/jenkins-docker-ansible/blob/master/ansible/roles/jenkins/defaults/main.yml). What am I missing here?

Viktor Farcic Post author
August 4, 2015 at 6:59 am

Docker commands are not executed inside the Jenkins container but through the "CD" node (that in this article is on the same server).

Milan Simonovic
August 19, 2015 at 6:07 pm

yeah, jenkins doesn't run inside a docker container. I've managed to get it packaged as a container, and put the docker executable on the path so the build jobs execute docker commands against the docker daemon running on the docker host (-H docker_host_ip:port). Guess it should also be possible with the docker-build-step plugin but i run into some issues there...

docker
August 18, 2015 at 5:47 am

how to use just docker(-compose) with ansible provision without vagrant?
Our workflow configured with vagrant and ansible provison. And want to migrate to docker with our pre-configured ansible tasks.
any suggestions about this?

Viktor Farcic Post author
August 18, 2015 at 9:25 am

If you check the code from the repository used in this article, you'll see the ansible/hosts/prod file. It contains IPs of each product. You can change them to point to the servers you're planning to deploy to. It doesn't matter much whether those servers are virtual machines or not. The only important thing is that you should import SSH keys of those servers. You can do that with the following command:

ssh-copy-id [USER]@[IP]

You don't need to use only one (in this case prod) file. You can have, for example, testing, preproduction, and so on.

In the article, we're running ansible-playbook with "-c local" that tells it to run on the same machine. In the "real world" you'll probably want to run Ansible on one machine (it can even be your laptop) and tell it to operate against remote servers. This is where ansible/hosts files start getting useful. You can, for example, run:

ansible-playbook my-playbook.yml -i hosts/my_production

Please let me know if this answered your questions. If not, you can contact me via HangOuts, send me an email (info is in the About section of the blog) or post another comment.

hbakhtiyor
August 18, 2015 at 11:21 am

Hi Viktor,

Thanks for your reply, sorry for that explanation. I wanted to say our current dev workflow configured with vagrant+vbox and ansible provision running locally, and want to migrate without using vagrant+vbox, instead we'll use several containers running locally which ansible will provision inside of containers

Viktor Farcic Post author
August 18, 2015 at 11:43 am

I'm not sure I understood what you meant by "inside of containers". If I remove the last part and stick with "...will provision." (without "inside of containers"), then the answer would be following.

The flow with Ansible is the same locally or inside an VM. You either use "-c local" argument to tell Ansible to run locally (locally means inside the machine where Ansible is installed be it VM, laptop or a server) or you can use "-i" to specify a file with IPs where Ansible should do something. I guess that this does not answer your question since it's almost the same as the previous one and I probably misunderstood you.

macsens
September 2, 2015 at 3:41 pm

Hi Victor, nice article. Thanks! I needed to 'fix' to things to make this work though:

in ansible/roles/dockers/tasks/main.yml add to top (as apt-get needs updating):

– name: Add Docker repository and update apt cache
apt_repository:
repo: deb https://get.docker.com/ubuntu docker main
update_cache: yes
state: present
tags: [docker]

In article, the second option to provision cd server refers to incorrect yml
vagrant ssh cd
ansible-playbook /vagrant/ansible/prod.yml -c local
exit

Should be:

vagrant ssh cd
ansible-playbook /vagrant/ansible/cd.yml -c local
exit

Cheers, Michiel

Viktor Farcic Post author
September 2, 2015 at 4:22 pm

You're right. Repository should indeed be added. Actually, apt task name should also be changed to docker.io to lxc-docker. It's been a long time since I wrote this article and a lot changed with Docker.

You're right about prod.yml. I updated the article.

Thank you for noticing those two problems.

pinto
September 8, 2015 at 4:26 pm

error:
==> cd: TASK: [registry | Container is running] **
==> cd: failed: [localhost] => {"failed": true, "parsed": false}
==> cd: BECOME-SUCCESS-xmccoadyqxwnznvlmajgskdgcrdrawgj
==> cd: Traceback (most recent call last):
==> cd: File "/root/.ansible/tmp/ansible-tmp-1441721935.04-98001605293810/docker", line 3185, in
==> cd: main()
==> cd: File "/root/.ansible/tmp/ansible-tmp-1441721935.04-98001605293810/docker", line 1540, in main
==> cd: started(manager, containers, count, name)
==> cd: File "/root/.ansible/tmp/ansible-tmp-1441721935.04-98001605293810/docker", line 1400, in started
==> cd: created = manager.create_containers(delta)
==> cd: File "/root/.ansible/tmp/ansible-tmp-1441721935.04-98001605293810/docker", line 1282, in create_containers
==> cd: params['host_config']['Memory'] = mem_limit
==> cd: KeyError: 'host_config'
==> cd:
==> cd:
==> cd: FATAL: all hosts have already failed — aborting
==> cd:
==> cd: PLAY RECAP
****
==> cd: to retry, use: –limit @/root/cd.retry

==> cd:
==> cd: localhost : ok=8 changed=7 unreachable=0 failed=1
The SSH command responded with a non-zero exit status. Vagrant
assumes that this means the command failed. The output for this command
should be in the log above. Please read the output to determine what
went wrong.

Viktor Farcic Post author
September 8, 2015 at 4:30 pm

Is this happening while running the “vagrant up cd” command? Can you send me the contents of the /root/.ansible/tmp/ansible-tmp-1441721935.04-98001605293810/docker file?

pinto
September 9, 2015 at 2:27 pm

Hi Victor

Thanks for replying

This is correct it happens on “vagrant up cd” command.
Previously it was working ok with your old git commit.
removed all dirs cloned git and reran
$ vagrant up cd
Bringing machine ‘cd’ up with ‘virtualbox’ provider…
==> cd: Importing base box ‘ubuntu/trusty64’…

==> cd: failed: [localhost] => {“failed”: true, “parsed”: false}
==> cd: BECOME-SUCCESS-gvjbdijpdiciligiqgkaejydeharkylz
==> cd: Traceback (most recent call last):
==> cd: File “/root/.ansible/tmp/ansible-tmp-1441800976.6-143869077974865/docker”, line 3185, in

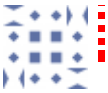
$ vagrant ssh cd
Welcome to Ubuntu 14.04.3 LTS (GNU/Linux 3.13.0-63-generic x86_64)

root@cd:~/.ansible/tmp# pwd
/root/.ansible/tmp

root@cd:~/.ansible/tmp# ls -l
total 0

Looks liek file does not exist. not rebooted system.
Maybe you can try on a different system

Thanks



Viktor Farcic Post author
September 9, 2015 at 8:40 pm

I found the problem. It's a bug introduced in Ansible 1.9.3. I changed the bootstrap.sh script and now it should work. Can you please try it out and let me know whether the latest commit fixed it?

pinto
September 10, 2015 at 1:39 pm

We finally got there.

with your updated reposirory we see messages like:
==> cd: msg: '/usr/bin/apt-get -y -o "Dpkg::Options::=–force-confdef" -o "Dpkg::Options::=–force-confold" install 'openjdk-7-jdk" failed: E: Failed to fetch http://archive.ubuntu.com/ubuntu/pool/main/g/gvfs/gvfs-common_1.20.3-0ubuntu1.1_all.deb 404 Not Found [IP: 91.189.91.23 80]
==> cd:

to resolve this:
vagrant box list ubuntu/trusty64
vagrant box remove ubuntu/trusty64 –box-version=20150908.0.0
vagrant box update ubuntu/trusty64

tested these 2 and they are ok:
bootstrap.sh
#pip install ansible==1.9.1
pip install ansible==1.9.2

as you saw there is indeed bug in 1.9.3

Tried again and failed
==> cd1: changed: [localhost] => {“changed”: true, “gid”: 0, “group”: “root”, “mode”: “0755”, “owner”: “root”, “path”: “/data/registry”, “size”: 4096, “state”: “directory”, “uid”: 0}
==> cd1:
==> cd1: TASK: [registry | Container is running] **
==> cd1: failed: [localhost] => {“failed”: true, “parsed”: false}
==> cd1: BECOME-SUCCESS-gnatllpijowrqylvrysxlxglxfamflay
==> cd1: Traceback (most recent call last):
==> cd1: File “/root/.ansible/tmp/ansible-tmp-1441883754.33-222862175761366/docker”, line 3185, in
==> cd1: FATAL: all hosts have already failed — aborting

Thank you for putting this example together.

Viktor Farcic Post author
September 10, 2015 at 1:50 pm

Thank you for letting me know about the problem. I tend to test the code in different settings before posting it as part of an article. However, once the article is published, it’s very hard for me to know that something got broken due to a bug in software releases made later on. I should probably setup some kind of periodic testing of all the repos but, as things usually go, there’s very little time and too much to do.

Anyways, I really appreciate when readers report problems. Thanks again.

Pingback: 使用Jenkins，Docker和Ansible进行持续集成和交付_ChinaSparker

Rogerio
September 23, 2015 at 1:51 pm

Hi Viktor,
First I must thank you for your articles, they’re an excellent read and time saving.
Sadly I´ve come accross with a problem uppon building the books-service as after setting the credentials (tried to build a couple of times with same failure):

Step 10 : CMD /source/run_tests.sh
—> Using cache

—> 28afc9634097
Successfully built 28afc9634097
+ sudo docker push 192.168.50.91:5000/books-service-tests
Error response from daemon: invalid registry endpoint https://192.168.50.91:5000/v0/: unable to ping registry endpoint https://192.168.50.91:5000/v0/
v2 ping attempt failed with error: Get https://192.168.50.91:5000/v2/: EOF
v1 ping attempt failed with error: Get https://192.168.50.91:5000/v1/_ping: EOF. If this private registry supports only HTTP or HTTPS with an unknown CA certificate, please add `--insecure-registry 192.168.50.91:5000` to the daemon's arguments. In the case of HTTPS, if you have access to the registry's CA certificate, no need for the flag; simply place the CA certificate at /etc/docker/certs.d/192.168.50.91:5000/ca.crt
Build step 'Execute shell' marked build as failure
Warning: you have no plugins providing access control for builds, so falling back to legacy behavior of permitting any downstream builds to be triggered
Finished: FAILURE

I know this article was written some time ago, but hope you can share some ideas on how I can solve this?
Thanks!

Viktor Farcic Post author
September 24, 2015 at 10:29 am

Hi,

I fixed it. Can you please pull the latest code from the repo and re-provision the cd VM. Something like:

cd jenkins-docker-ansible
git pull
vagrant provision cd

From there on, it should work. Can you please let me know how it went?

Rogerio
September 28, 2015 at 6:50 pm

Hi,
Thanks for you quick reply… (I was hoping to get an email alert from your reply). I' ll describe what I did:
1- Pulled from git, re-provisioned the vm and got an error while 'upping' the CD vm – destroyed and removed all vagrant and virtual box files and cloned into a new dir and 'upped' the CD vm and got the same error as the one after pulled:

==> cd:
==> cd: TASK: [docker | Files are present] **
==> cd: changed: [localhost] => {"changed": true, "checksum": "326f2e9dcc9b5fd0b2baae05315dbd9d7f4a36a1", "dest": "/etc/default/docker", "gid": 0, "group": "root", "md5sum": "f04f9078c3d8af2df8b5e70b65a7c02a", "mode": "0644", "owner": "root", "size": 65, "src": "/root/.ansible/tmp/ansible-tmp-1443453945.56-57082013926945/source", "state": "file", "uid": 0}
==> cd:
==> cd: TASK: [docker | Docker service is restarted] **********************************
==> cd: failed: [localhost] => {"failed": true}
==> cd: msg: * Docker is managed via upstart, try using service docker
==> cd: * Docker is managed via upstart, try using service docker
==> cd:
==> cd:
==> cd: FATAL: all hosts have already failed — aborting
==> cd:
==> cd: PLAY RECAP
****
==> cd: to retry, use: –limit @/root/cd.retry
==> cd:
==> cd: localhost : ok=8 changed=5 unreachable=0 failed=1
The SSH command responded with a non-zero exit status. Vagrant
assumes that this means the command failed. The output for this command
should be in the log above. Please read the output to determine what
went wrong.

2 – Executed 'vagrant reload –provision cd' and everything went as expected.
3 – Configured node credentials and books-service build job completed successfully
4 – Performed the ssh-keygen steps and got following error on the books-service-deployment job (even after trying to re-build):

Started by upstream project "books-service" build number 1
originally caused by:
Started by an SCM change
Building remotely on CD (cd) in workspace /data/jenkins/slaves/cd/workspace/books-service-deployment
[books-service-deployment] $ /bin/sh -xe /tmp/hudson7665963082914977600.sh

+ ansible-playbook /vagrant/ansible/books-service.yml -i /vagrant/ansible/hosts/prod
ERROR: The file /vagrant/ansible/hosts/prod is marked as executable, but failed to execute correctly. If this is not supposed to be an executable script, correct this with `chmod -x /vagrant/ansible/hosts/prod`.
Build step 'Execute shell' marked build as failure
Finished: FAILURE

5 – Tried the curl requests and got:
curl: (7) Failed to connect to localhost port 9001: Connection refused
– Accessing from browser also gives error. So I'm stuck again

Hope it helps.

Viktor Farcic Post author
September 28, 2015 at 10:07 pm

Hi,

I'll answer using the same numbering as your questions.

1. Ansible 1.9.1 introduced an error that prevents it from using services (at least with this version of Ubuntu). The step that failed was supposed to restart the Docker service whenever the configuration changed. The reason it worked after `vagrant reload` is that the configuration already changed (the "Files are present" task) in the previous playbook run so the "Docker service is restarted" task did not run the second time and thus did not fail. Same thing happened to me last time and that's the reasons I haven't noticed the problem. This time I started over by destroying the VMs first and same thing happened. I just fixed this by upgrading Ansible version to 1.9.2. As a side note, last couple of versions of Ansible are very buggy. I think that they are working hard on the version 2.0 and somehow neglected smaller updates. For example, the latest version (1.9.3) also fails (but for a different reason).

2. With the changed version of Ansible, now it should work. To be on the safe side, please destroy the VM and bring it up again.

```
git pull
vagrant destroy -f
vagrant up
```

1. OK.

2. That's strange. It's as if Jenkins thinks those are two commands (`ansible-playbook`

`/vagrant/ansible/books-service.yml -i` and `/vagrant/ansible/hosts/prod`. In reality, both lines are one single command. Can you please double check the configuration of the job and make sure that everything is in one line. When I run it from scratch, it worked. The command is `ansible-playbook /vagrant/ansible/books-service.yml -i /vagrant/ansible/hosts/prod` (all in one line). Maybe it's somehow related to the first problem (even though I fail to see the connection). Please let me know how it went. You can contact me on HangOuts and we can debug it together.

3. Until the container is deployed with the previous step, this one won't work.

Rogerio
September 29, 2015 at 1:20 pm

Hi,
Once again thank you for availability for this.

Followed your instructions: git pulled, vagrant destroyed and vagrant upped
and all went ok until the build of the `books-service-deployment` that outputted the same error as before:

```
Started by upstream project "books-service" build number 1
originally caused by:
Started by an SCM change
Building remotely on CD (cd) in workspace /data/jenkins/slaves/cd/workspace/books-service-deployment
[books-service-deployment] $ /bin/sh -xe /tmp/hudson7096597554297840616.sh
+ ansible-playbook /vagrant/ansible/books-service.yml -i /vagrant/ansible/hosts/prod
ERROR: The file /vagrant/ansible/hosts/prod is marked as executable, but failed to execute correctly. If this is not supposed to be an executable script, correct this with chmod -x /vagrant/ansible/hosts/prod.
Build step 'Execute shell' marked build as failure
Finished: FAILURE
```

Remembered this could be related to being trying this on a windows host – permissions can be a pain on shared vagrant dirs with linux guests – so I copied all files from `/vagrant` to the vagrant's user home dir on the CD vm; fixed permissions on files as it went complaining. In the end needed to fix permissions on the following 2 files to become:

```
-rw-rw-r-- 1 vagrant vagrant 116 Sep 29 09:27 /home/vagrant/ansible/hosts/prod
-rw------- 1 vagrant vagrant 1702 Sep 29 09:41
/home/vagrant/machines/prod/virtualbox/private_key
```

Then edited file `/home/vagrant/ansible/hosts/prod` to point to `/home/vagrant/machines/prod/virtualbox/private_key`. And edited the Jenkin's job configuration to point to `/home/vagrant/ansible/hosts/prod` (the command was in a single line – probably the 2 lines in my last comment was due to the size of the browser's window as I copied & pasted from the Jenkins console output into a text editor before I pasted here).

The solution is now running as it should – I'm crossing my fingers hoping the future releases of Ansible will be 'bugg free' in case I need to start again .
Thank you again for all your help and for sharing this great work.
Cheers.

Viktor Farcic Post author
September 29, 2015 at 6:18 pm

I'm glad it worked at the end. Lately I started testing the code used in articles on Linux, Windows and Mac. However, I started doing that quite after this article was written and most were tested only on my laptop with Ubuntu. The theory was that with VMs and Vagrant everything should work everywhere but that doesn't seem to be the case with shared folders. I should probably go back and rewrite this article so that it works correctly in Windows. It's just that there is never enough time... If you feel like you're up to it, I'll give you the write access to the article.

Rogerio
October 1, 2015 at 11:02 am

Hi,
So I've tried again from the start and this time applied the 'fix' to the Vagrantfile that you have on some of your articles:

```
config.vm.synced_folder ".", "/vagrant", mount_options: ["dmode=700,fmode=600"]
```

And that solved the problem.

Viktor Farcic Post author
October 1, 2015 at 11:30 am

Thank you for reminding me of the fix. Since I don't use Windows and don't experience the same problem I forgot about it. I updated the article so that other can skip the pain you went through.

Peter
November 3, 2015 at 3:44 pm

Am I the only one who is getting error during image building from book-service repository? In dockerfile there are non-existing debs addded.

Viktor Farcic Post author
November 3, 2015 at 7:40 pm

I think that it was a temporary failure of one of the Debian repositories. Yesterday I noticed some failures with other containers I'm working on. I built the containers from this article few minutes ago and everything went fine. Can you try again and let me know how it went? If you still have problems, please contact me on HangOuts or email and we'll try to solve it together. You can find my info in the "About" section of this blog.

Peter
November 4, 2015 at 10:17 am

Still have issues with extra debs

To run your code I had to add pre task with apt-get update to install java and so on. I had to added fonts-liberation package for Dockerfile.test in books serivce and removed these extra debs from Dockerfile. To make thoses changes I forked your book-serivce repo. Thank you very much for such detailed example of CD

Viktor Farcic Post author
November 4, 2015 at 5:41 pm

Can you make a pull request?

Piotr Żmijewski
November 4, 2015 at 11:10 pm

I changes also distribution to ubuntu, so I will have to check if it is working with debian:jessie and then I will pull request

aieeeeeeeee
November 26, 2015 at 12:58 pm

Thanks for this wonderful tutorial!!! Juste one question, is the deployment job triggered automatically as soon as there is a new successful build of books-service?
Ideally I would like to make sure that each time I commit to github it triggers a build of books-service (via a hook) and then it deploys everything in production if successfully build...

Thanks again!

Viktor Farcic Post author
November 26, 2015 at 1:12 pm

The Jenkins job checks the repository every 5 minutes and if it finds changes triggers the build. This is not a very efficient way of doing things and it would be better to create a trigger/hook in your repository that calls the Jenkins job. However, since the example in this article is running on local network and uses public GitHub, such a trigger would not work. In “real world” situations, such a problem would not exist and trigger would be preferable.