Dots and Brackets: Code Blog

Blog about DevOps, distributed applications and microservices

How to use Vagrant to create Consul cluster

Last two articles about Consul service discovery involved one simple but extremely boring manual task: creating and configuring a cluster. In fact, I had to do it twice. I had to create three virtual machines, download and unpack Consul on them, find out their IP addresses, add configuration files and finally launch the binaries.

It's dull. It's boring. Humans shouldn't do that kinds of things by hand. Seeing how easily we can automate creation of Docker containers with Dockerfile and docker-compose makes me wonder if we can do the same for hosts.

In fact, we can. Vagrant is a tool to do exactly that. It has it's own Vagrantfile which can store configuration for one or more hosts and then bring them up to life with simple `vagrant up`.

What's Vagrant

As official documentation says, Vagrant is a tool for building and managing virtual environments. If we ever need to configure a virtual machine either for development, for tests, or for some production needs, we can do that manually (and forget in a week how exactly we did that), or we can put VM configuration steps into a Vagrantfile. Not only we'll be able to create a VM from that as many times as we need, we also could add it to version control system, share with the teammates, or test a VM locally before bringing identical machine to production. What's cool, the same Vagrantfile used for creating a VM in, let's say, VirtualBox on local machine, with little to no modifications can be used for bringing up AWS or Azure host.

The plan for today

Knowing how that I already had to create three almost identical VMs twice, it's safe to expect that it might happen again, so.. let's automate that! Let's create a Vagrantfile to bring up three VirtualBox VMs with one Consul server and two regular agents on them. Obviously, server and agents will work as a cluster. Should we begin?

Prerequisites

As usual, before we can start we need to do some installation first. Mighty Google can point to download pages for both Vagrant and VirtualBox, and installation process itself is totally painless. I'm running both of the tools on Mac, but Windows and main flavors of Linux are also supported.

Step 0. Creating blank virtual machine

Creating new virtual machine with Vargrant starts with creating new Vagrantfile. It's fairly easy to do:

```
vagrant init ubuntu/xenial64 --minimal
#A `Vagrantfile` has been placed in this directory. You are now
#ready to `vagrant up` your first virtual environment! Please read
#the comments in the Vagrantfile as well as documentation on
#`vagrantup.com` for more information on using Vagrant.
```

Everything is pretty straightforward. `init` obviously creates the file. `ubuntu/xenial64` is a box to use, and the box itself is like a base image in Docker. In our case the box is "64 bit Ubuntu 16.04 LTS", but there're many,

many others. Finally, as `init` command tends to produce huge Vagrantfile with three meaningful lines and tons of comments, `--minimal` is the way to keep the file compact.

Now, type `vagrant up` and few minutes later you'll have fully functional Ubuntu VM:

```
vagrant up
#Bringing machine 'default' up with 'virtualbox' provider...
#==> default: Importing base box 'ubuntu/xenial64'...
#..
```

Vagrant used VirtualBox as default VM provider, but we could choose something else. E.g. Google Compute Engine would require `--provider=google` flag.

After VM is created we can get into it with `vagrant ssh`:

```
vagrant status
#Current machine states:
#
#default                   running (virtualbox)
#...

vagrant ssh
#Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.4.0-72-generic x86_64)
#...
```

Step 1. Provisioning the VM

Now it's time for installing and configuring Consul agents, and starting with Consul server seems like a logical choice. Here's what we need to do:

1. Give the VM a meaningful name, e.g. `consul-server`
2. Download and unzip Consul binaries
3. Assign static IP address (otherwise how other cluster members are going to find it?)
4. Register and start `consul` as Ubuntu service

Step 1.1. Setting up a VM name

For this task we'll need to make some changes in Vagrantfile. This is what `init --minimal` command created for us:

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/xenial64"
end
```

Setting up a VM name is just one more line in Ruby language:

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/xenial64"
  config.vm.hostname = "consul-server"
end
```

Now let's reload the VM with `vagrant reload`, SSH back into it and confirm that hostname indeed has changed:

```
vagrant reload
#==> default: Attempting graceful shutdown of VM...
#...
vagrant ssh
#...
ubuntu@consul-server:~$ hostname
#consul-server
```

Yup, it all worked.

Step 1.2. Shell provisioner for installing Consul

This task will be a little bit harder. The process of configuring the server is called provisioning, and Vagrant supports all sorts of ways to perform that. For instance, it supports `shell` provisioner for executing shell files, `file` for copying a file or folder, or even `ansible`, `chef` or `puppet` to do virtually anything.

For our task `shell` provisioner will do:

```
#...
config.vm.provision "shell", path: "provision.sh"
#...
```

`path` points to the file with provisioning steps. We need to create that one as well:

```
#!/bin/bash

# update and unzip
apt-get -y update && apt-get install -y unzip

# install consul
cd /home/ubuntu
```

```
version='0.8.0'
wget https://releases.hashicorp.com/consul/${version}/consul_${version}_linux_amd64.
unzip consul.zip
rm consul.zip

# make consul executable
chmod +x consul
```

However, if you try to call `vagrant reload` this time, the changes won't be applied. The thing is Vagrant does provisioning only during VM creation. Or when it's told to do so. So type `vagrant reload --provision` (or `vagrant provision` if VM was already reloaded), and enjoy newly deployed Consul:

```
vagrant reload --provision
vagrant ssh
ubuntu@consul-server:~$ ./consul version
# Consul v0.8.0
```

Step 1.2.1. Optional: make provision script idempotent

There's old Jedi trick allowing to apply provisioning script multiple times without nasty side effects or errors. It comes in handy when we add new features to provisioning stage and we want to make sure that existing ones don't break anything during reprovisioning.

At the moment our provisioning script does only two things:

- installs unzip
- installs consul

If `provision.sh` will try to install unzip or consul only if they aren't installed already, it will become idempotent.

```
#..
# update and unzip
dpkg -s unzip &>/dev/null || {
    apt-get -y update && apt-get install -y unzip
}

# install consul
if [ ! -f /home/ubuntu/consul ]; then
#...
fi
```

Now reprosivioning VM will skip the steps that already has been taken thus avoiding 'already exists' sort of errors.

Step 1.3. Assigning static IP

Piece of cake. Two more lines in Vagrantfile and it’s ready:

```
#...
config.vm.hostname = "consul-server"

serverIp = "192.168.99.100"
config.vm.network "private_network", ip: serverIp
#...
```

Step 1.4. Starting Consul as Ubuntu service

This is actually hard. Not a rocket science, but also not entirely trivial.

In order to make Consul a service we have to declare it as systemd service. We can download sample service definition file from here and then copy it to `/etc/systemd/system/` directory during provisioning. `consul` executable location itself also has to be changed from home folder to something more appropriate. I’ll skip most of the details (sources are available at github), but here’re some main points from this step.

1. Vagrant mounts the whole directory with Vagrantfile to `/vagrant` path at guest OS, so we can use this mount for copying configuration files during provisioning. E.g.

```
cp /vagrant/consul.service /etc/systemd/system/consul.service
```

2. Consul agent as a service reads config files from `/etc/systemd/system/consul.d/`. It’s convenient, as now we can configure server and agents by putting some sort of `init.json` in there.

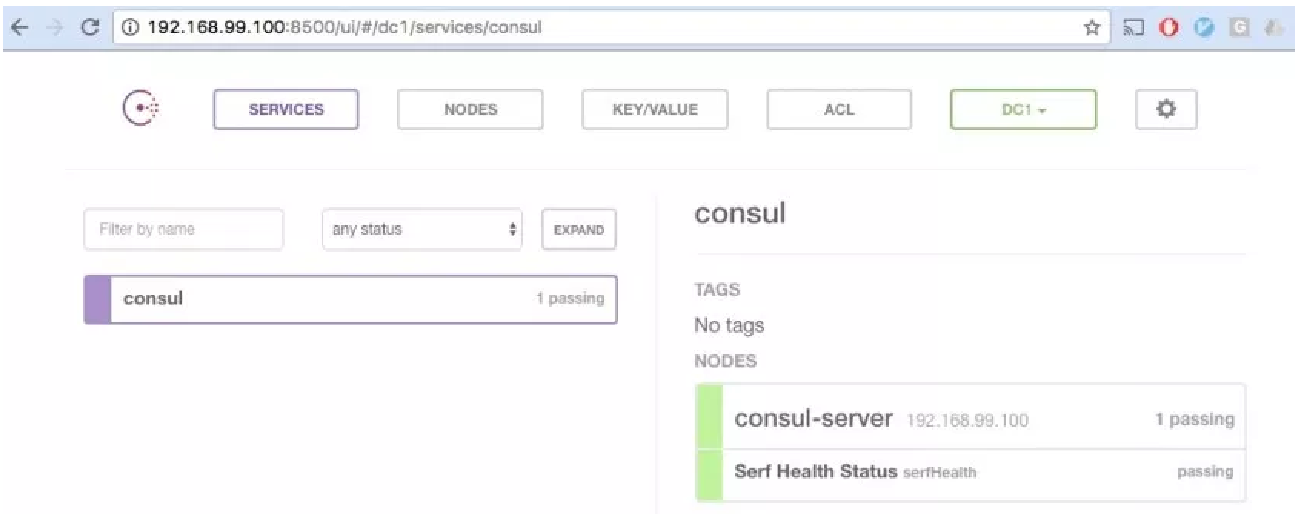
```
#...
  serverInit = %(
    {
        "server": true,
        "ui": true,
        "advertise_addr": "#{serverIp}",
        "client_addr": "#{serverIp}",
        "data_dir": "/tmp/consul",
        "bootstrap_expect": 1
    }
  )

  config.vm.provision "shell", inline: "echo '#{serverInit}' > /etc/systemd/syste
#...
```

3. Finally, service needs to be started. That’s one more provisioning line:

```
#...
config.vm.provision "shell", inline: "service consul start"
#...
```

Step 1.5. Test run

If you haven't fallen asleep so far, this is the right time to type `vagrant reload --provision` and go to `192.168.99.100:8500` in your browser of choice. There's something new in there:



consul 1 passing

consul

TAGS
No tags
NODES

consul-server 192.168.99.100 1 passing
Serf Health Status serfHealth passing

It's true Consul server. You can stop the VM any time. You can even delete it. But as long as you have Vagrantfile, restoring ready to use Consul server is as easy as running `vagrant up`.

Step 2. Adding more VMs to the cluster

Vagrantfile is not limited to only one VM. By using `define` function we can declare as many VMs as we like:

```
config.vm.define "host1" do |host|
  host.vm.hostname = "first-host"
end

config.vm.define "host2" do |host|
  host.vm.hostname = "second-host"
end
```

As Vagrantfile itself is written in Ruby, we can use loops and functions to make multiple hosts creation easier. But before that happens, destroy existing `consul-`

`server` with `vagrant destroy -f`, as we're totally going to change the configuration file.

I think some refactoring also won't do any harm. After all, configuring Consul server and regular agents is almost identical, so why not try to reuse the code:

```
    #...
    def create_consul_host(config, hostname, ip, initJson)
      config.vm.define hostname do |host|

        host.vm.hostname = hostname
        host.vm.provision "shell", path: "provision.sh"

        host.vm.network "private_network", ip: ip
        host.vm.provision "shell", inline: "echo '#{initJson}' > /etc/systemd/system/c
        host.vm.provision "shell", inline: "service consul start"
      end
    end
    #...
```

This will allow to create Consul server with one function call:

```
#...
  create_consul_host config, "consul-server", serverIp, serverInit
#...
```

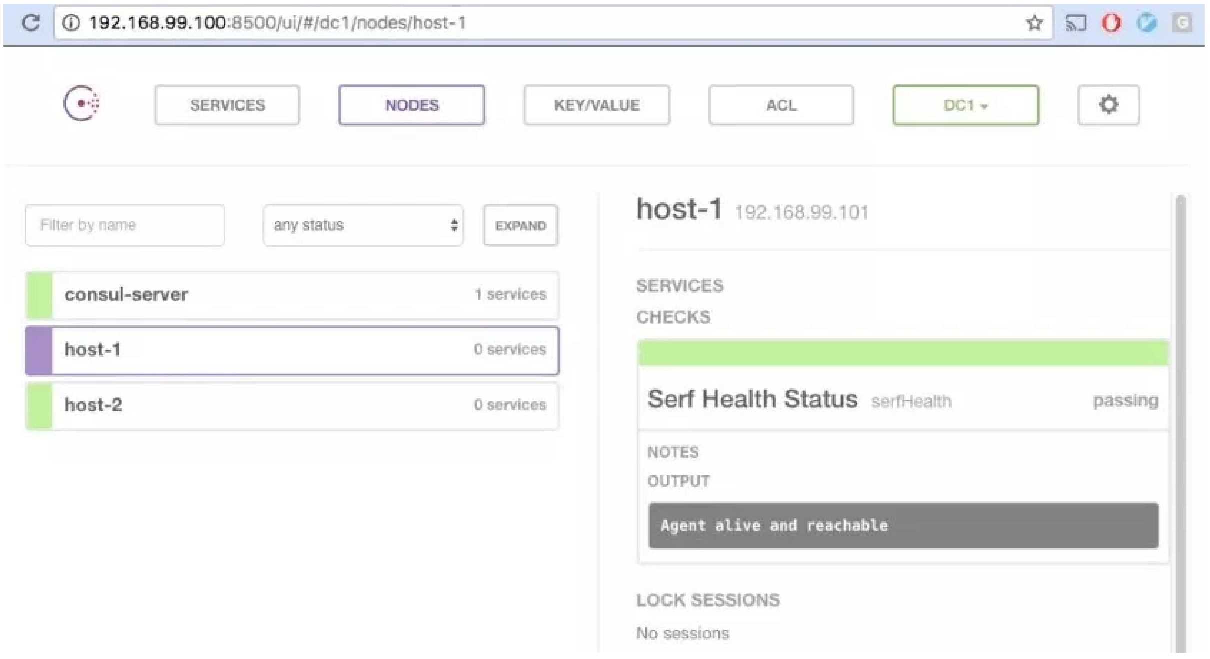
And now just take a look what it takes to create two more virtual machines with fully functional Consul agents in them:

```
#...
  for host_number in 1..2
    hostname="host-#{host_number}"
    clientIp="192.168.99.10#{host_number}"

    clientInit = %(
      {
        "advertise_addr": "#{clientIp}",
        "retry_join": ["#{serverIp}"],
        "data_dir": "/tmp/consul"
      }
    )

    create_consul_host config, hostname, clientIp, clientInit
  end
#...
```

Isn't that cool? We're creating and configuring virtual machines in a loop!

But let's check that new Vagrantfile indeed worked:

```
vagrant up
#Bringing machine 'consul-server' up with 'virtualbox' provider...
#Bringing machine 'host-1' up with 'virtualbox' provider...
#Bringing machine 'host-2' up with 'virtualbox' provider...
```



consul-server 1 services
host-1 0 services
host-2 0 services

host-1 192.168.99.101

SERVICES
CHECKS

Serf Health Status serfHealth passing

NOTES
OUTPUT

```
Agent alive and reachable
```

LOCK SESSIONS
No sessions

The whole cluster was created from configuration file. I don't have to do that manually anymore.

Here should come the conclusion…

But I can't come up with any.

Well, maybe one thought. Somehow moving from manual to automatic creation and provisioning of VMs and hosts is hard. Not from technology point of view but from some sort of psychological inertia. "VMs are big and heavy, how can we automate them?". Or even my favorite: "I know it's one time job, I won't need to do that again". But there's always the second time. At some point handcrafted environments start pushing us back, as they are not easily reproducible, often outdated and honestly, most of the time we have no idea what exactly is installed on them and how they manage to work.

Putting VM and host configuration to a file and creating new instances from it changes everything. Those hosts are predictable, always up to date and it's not a big deal to create a new one. Vagrant is one of the tools that makes this magic possible. You can find all the source code from this post at github.

Share this:

pav / April 19, 2017 / Development / consul, quick guide, service discovery, ubuntu, Vagrant, virtualization

4 thoughts on “How to use Vagrant to create Consul cluster”

Pingback: Using Vagrant for Windows VMs provisioning – Dots and Brackets: Code Blog

Pingback: Building VM image with Packer – Dots and Brackets: Code Blog

Pingback: Provisioning Vagrant VM with Ansible – Dots and Brackets: Code Blog

Pingback: How to unit test.. a server with goss – Dots and Brackets: Code Blog