SEE TEXT ONLY

BY STEVE WILLS

Vagrant (https://www.vagrantup.com/) is a command-line utility for building and managing portable development environments such as virtual machines (VMs). It creates, starts up, provisions, and destroys VMs easily. It is often used as a tool for setting up development and test environments such as on a laptop.

VAGRANT

Vagrant allows you to set up these environments in an automated and reproducible way, which ultimately increases productivity and allows you to share these automated environments with others. Vagrant, written in Ruby (https://www.ruby-lang.org/), is a cross-platform utility. It easily runs on Mac OS, Windows, Linux, and FreeBSD. By supporting multiple hypervisors—including VirtualBox, VMWare, Bhyve (with some work)—as well cloud hosting—such as Amazon AWS, Google Cloud Compute, Azure, and many others—Vagrant offers a lot of flexibility. In terms of provisioning or initial VM setup, it supports a variety of tools, such as Ansible, Chef, Puppet, Salt, and even shell scripts.

Hypervisor and provisioner support is provided by plugins, so end users can easily develop plugins for particular providers or provisioners. There are many plugins available (https://github.com/mitchellh/vagrant/wiki/Available-Vagrant-Plugins).

Why Use Vagrant?

You can create Vagrant environments and easily share them with your coworkers regardless of which OS they use on their laptops or desktops. And you can bring up a single or multi-host setup with a single `vagrant up` command. By making development environments reproducible, you can avoid the common "works on my machine" type of problem.

Vagrant is typically used on laptops or desktop workstations for setting up development and test environments, but it can be used elsewhere. For example, one interesting use case is for creating build nodes or temporary hosts for automated build environments or tools such as Jenkins. Generally, Vagrant is used by software developers or testers. It can be useful for operations folks, too, such as for modeling production environment setups.

Vagrant is a very useful tool meant for reproducible, ephemeral development and testing environments. However, it is not meant for managing production hosts with critical data. Vagrant's simple design makes it far too easy to destroy everything you have created with it for that type of usage.

How to Get Vagrant

You can install Vagrant on FreeBSD using the command `pkg install vagrant`. This will install Vagrant and all of its dependencies. You will also want to install a hypervisor. For the purposes of this article, I will demonstrate both VirtualBox and Bhyve as hypervisors. To install VirtualBox on FreeBSD, use `pkg install virtualbox-ose`. Be certain to follow any instructions about system setup that are listed.

Components of a Vagrant Project

There are two main components to Vagrant environments. The first is what is called the Vagrant "box". This is the wrapper of a disk image containing an installed guest OS and meta-data about that disk image and OS. Users can create these themselves—manually—and share them or use pre-created boxes created by other users or groups. Prebuilt boxes can be found at https://atlas.hashicorp.com/boxes/search. The `vagrant` command will search this location by default for boxes that do not already exist locally. Boxes can be manually added to the local Vagrant cache using the `vagrant box add` command, such as in the following example:

```
vagrant box add hashicorp/precise64
```

`You can list cached boxes with`

```
vagrant box list.
```

The second main component is the `Vagrantfile`. This file lists the VMs that are in your Vagrant project and how to set up those VMs. While the Vagrantfile is written in Ruby, editing it does not require extensive knowledge of Ruby. It is intended to be simple. More information on the `Vagrantfile` is available at https://www.vagrantup.com/docs/vagrantfile/.

The Vagrantfile also serves to mark for Vagrant where the root of your project lives. This helps Vagrant find other files that may be referenced in the Vagrantfile using relative file reference.

Setting up a Project

The simplest way to set up a project is to run the following command:

```
vagrant init hashicorp/precise64
```

This will create the Vagrantfile in the current directory. Next run:

```
vagrant up
```

This command will download the box to a local directory (`~/.vagrant.d/boxes/` by default) and start up the VM. After this, you can run:

```
vagrant ssh
```

to ssh into the VM and begin using it. When you are done, you can stop the VM using:

```
vagrant halt
```

Or if you want to get rid of the VM completely, run:

```
vagrant destroy
```

This will properly remove the instance of the VM with its local changes, but leave the cached copy of the box in `~/.vagrant/boxes/`.

In our example VM, notice we did not specify any provisioning or configuration steps. We can add steps to the Vagrantfile to configure the VM. For example, uncomment these lines in the

Vagrantfile:

```
config.vm.provision "shell", inline: <<-SHELL
    apt-get update
    apt-get install -y apache2
SHELL
```

The next time you run `vagrant up` or `vagrant provision`, Vagrant will run the `apt-get` commands to install the Apache web server.

More information on provisioning is available at https://www.vagrantup.com/docs/getting-started/provisioning.html.

One interesting feature of Vagrant is the synced folder feature. It will sync files to and from the host operating system to the guest operating system, making it easier, for example, to edit code you want to build or run in the VM. The VM has a default network connection that allows connecting to the VM with `ssh`. You can add additional port forwards to the VM on any additional ports needed.

Creating a New Vagrant Project

As an example, we will create a new Vagrant project running a web server. The first step of creating a new Vagrant project is creating a new directory and running `vagrant init` in that directory. For example:

```
mkdir journal
cd journal
vagrant init freebsd/FreeBSD-11.0-RELEASE-p1
```

This will create a Vagrantfile in our new directory. Note that we will want to make a number of changes to this default file before using it. Due to defaults of Vagrant, you will want to add the following line to our new Vagrantfile:

```
config.vm.guest = :freebsd
```

Also, due to a bug in the FreeBSD box (for which a fix is available and will be committed soon), you will need to add the following line to the new Vagrantfile:

```
config.vm.base_mac = "080027FC8C33"
```

To avoid a boot timeout during the first boot, as the system performs its first boot update, we need to add the following line to our Vagrantfile:

```
config.vm.boot_timeout = 3600
```

There is another change to make before we can try to initialize our box, as we need to disable the default shared folders:

```
config.vm.synced_folder ".", "/vagrant", id: "vagrant-root", disabled: true
```

Also, we'll want to expose a port for our web server by adding this line:

```
config.vm.network :forwarded_port, guest: 80, host: 8080
```

And, since we do not have `bash` installed by default on FreeBSD, we need to configure Vagrant to use `sh` as its ssh shell:

```
config.ssh.shell = "sh"
```

The default amount of RAM specified is 512MB, which we should increase to 1GB. And we will allocate two CPUs to the VM:

```
config.vm.provider :virtualbox do |vb|
  vb.customize ["modifyvm", :id, "--memory", "1024"]
  vb.customize ["modifyvm", :id, "--cpus", "2"]
end

config.vm.provider :bhyve do |vm|
  vm.memory = "1024M"
  vm.cpus = "2"
end
```

Notice how we have separate blocks for each hypervisor provider. Let's also add additional disk to the VM for use with ZFS. This will be hypervisor specific as well. In the section for virtualbox, we add:

```
 file_to_disk = File.realpath( "." ).to_s + "/extradisk.vdi"

 if ARGV[0] == "up" && ! File.exist?(file_to_disk)
   vb.customize [
   'createhd',
   '--filename', file_to_disk,
   '--format', 'VDI',
   '--size', 3 * 1024 # 3 GB
   ]
  end
  vb.customize [
    'storageattach', :id,
    '--storagectl', 'IDE Controller', # The name may vary
    '--port', 1, '--device', 0,
    '--type', 'hdd', '--medium',
    file_to_disk
    ]
```

And in the section for bhyve, we add:

```
  vm.disks = [{name: "extradisk", size: "3G", format:"raw",}]
```

Our next step is to install the web server process and ensure that it is running.

```
  config.vm.provision "shell", inline: <<-SHELL
    pkg install -y apache24
    sysrc apache24_enable=yes
    service apache24 start
  SHELL
```

So, our Vagrantfile, minus comments, should look like the following:

```
Vagrant.configure("2") do |config|
  config.vm.box = "freebsd/FreeBSD-11.0-RELEASE-p1"
  config.vm.guest = :freebsd
  config.vm.base_mac = "080027FC8C33"
  config.vm.boot_timeout = 3600
  config.vm.network :forwarded_port, guest: 80, host: 8080
  config.vm.synced_folder ".", "/vagrant", id: "vagrant-root", disabled: true
  config.ssh.shell = "sh"

  config.vm.provider :virtualbox do |vb|
   vb.customize ["modifyvm", :id, "--memory", "1024"]
   vb.customize ["modifyvm", :id, "--cpus", "2"]
   file_to_disk = File.realpath( "." ).to_s + "/extradisk.vdi"
```

CODE CONTINUES NEXT PAGE

CODE CONTINUED FROM PREVIOUS PAGE

```
  if ARGV[0] == "up" && ! File.exist?(file_to_disk)
    vb.customize [
    'createhd',
    '--filename', file_to_disk,
    '--format', 'VDI',
    '--size', 3 * 1024 # 3 GB
    ]
  end
  vb.customize [
    'storageattach', :id,
    '--storagectl', 'IDE Controller', # The name may vary
    '--port', 1, '--device', 0,
    '--type', 'hdd', '--medium',
    file_to_disk
    ]
  end

  config.vm.provider :bhyve do |vm|
    vm.memory = "1024M"
    vm.cpus = "2"
    vm.disks = [{name: "extradisk", size: "3G", format:"raw",}]
  end

  config.vm.provision "shell", inline: <<-SHELL
    pkg install -y apache24
    sysrc apache24_enable=yes
    service apache24 start
  SHELL
end
```

Now we can start our VM using `vagrant up`. When Vagrant creates and starts our VM, it creates it as a headless VM that is not visible. You may want to open the VirtualBox Manager so that you can see the VM in the VM list. You can also click the VM and then click the "Show" button to see the VM console. While the shell provisioning script is running, we can view the output of the commands. After this finishes, we should be able to bring up the default website:

```
http://localhost:8080/
```

To access the box, you can `ssh` in using the previously mentioned `vagrant ssh` command.

Existing Projects

One of the strongest powers of Vagrant comes from using the automation to create boxes and Vagrantfiles and sharing them with other users. So, let's look at some existing projects. Note these examples were tested with the VirtualBox hypervisor.

Minio

Minio (https://minio.io/) is a "distributed object storage server built for cloud applications and devops." It is compatible with Amazon's S3 and can be pretty useful. For this example, I have created a Vagrantfile that uses Ansible to configure Minio in FreeBSD. First, install Ansible by running `pkg install ansible`; then clone or download this project (https://github.com/swills/minio.vagrant). You can then run the `vagrant up` command and have a working Minio server with just a few simple commands.

```
mkdir minio
cd minio
git clone https://github.com/swills/minio.vagrant.git .
vagrant up
```

The `vagrant up` step will take a few minutes, as the FreeBSD VM does an automated `freebsd-update` on first boot. After it finishes, `ssh` into the VM and get the authorization keys for Minio by running `vagrant ssh`. Then in the VM, execute the following commands:

```
sudo grep accessKey /usr/local/etc/minio/.minio/config.json
  "accessKey": "KO2E80M9H87CDIJKP6A1",
sudo grep secretKey /usr/local/etc/minio/.minio/config.json
  "secretKey": "2mMkACOoq7v8oUrZqZs3+S9gRwqNLws5MZTNjgnE"
```

You can then login to Minio at:

```
http://localhost:9000/minio/login
```

You will need to use the accessKey and secretKey found in the VM. (These are generated at the time the service is first started and will vary.)

Gluster

GlusterFS (http://www.gluster.org/documentation/About_Gluster/) is a free software parallel distributed filesystem, capable of scaling to several petabytes. Similar to the previous example, I've created a Vagrantfile that uses Ansible to configure Gluster on a pair of FreeBSD VMs. You can clone this project (https://github.com/swills/okapi) and run `vagrant up` to get started.

```
mkdir gluster
cd gluster
git clone https://github.com/swills/okapi.git .
vagrant up
```

Next, run the `vagrant ssh server1` command. Once you are in that VM, run the following command:

```
sudo gluster peer probe server2
sudo gluster volume create gv0 replica 2 server1:/gluster/gv0 server2:/gluster/gv0
sudo gluster volume start gv0
sudo gluster volume info
sudo mount_glusterfs server1:/gv0 /mnt
```

That will set up Gluster. Then you can `vagrant ssh server2` and in that VM run:

```
sudo mount_glusterfs server1:/gv0 /mnt
```

From this point, you should have a working Gluster cluster. Any file created or modified in `/mnt` on one should be available in the other. (The `gluster` commands are run manually so that they happen after both machines are up.)

Poudriere

Finally, and perhaps of particular interest to anyone interested in doing FreeBSD ports development, there is a Vagrantfile and set of Ansible scripts that will set up a complete `poudriere` development environment. This includes a complete ports tree for doing ports development and multiple jails for testing builds. Note that setting up this one will take significantly longer to provision and will require more disk space due to the jail build process and the ports tree checkout.

```
mkdir poudriere_vagrant
cd poudriere_vagrant
git clone https://github.com/swills/gerenuk.git .
vagrant up
```

After this finishes successfully, you can `vagrant ssh` into the VM and work with ports and run `poudriere` to test builds. You can edit files in `/usr/local/poudriere/ports/default` and build them using standard poudriere commands, for example:

```
sudo poudriere bulk -j 110-amd64 shells/zsh
```

The provisioning scripts have also set up NGINX to serve the poudriere statuspages at:

```
http://localhost:10080/
```

Using Vagrant with Bhyve

Vagrant can be used with Bhyve via a project called vagrant-bhyve (https://github.com/jesa7955/vagrant-bhyve) that was created during the Google Summer of Code project in 2016. In order to use it, you will probably want another project that helps convert VMs to formats that vagrant-bhyve can understand, called vagrant-mutate (https://github.com/sciurus/vagrant-mutate). You can install both on FreeBSD by running:

```
pkg install rubygem-vagrant-bhyve rubygme-vagrant-mutate
```

After installing the plugin as we did above, we need to ensure vagrant is using it by running the following commands:

```
vagrant plugin install vagrant-mutate
vagrant plugin install vagrant-bhyve
```

As mentioned previously, we can list currently locally cached boxes with the `vagrant box list` command. Our output might look like this:

```
freebsd/FreeBSD-11.0-RELEASE-p1 (virtualbox, 2016.09.29)
hashicorp/precise64              (virtualbox, 1.1.0)
```

We can use `vagrant mutate` to convert these boxes for use with Bhyve. To convert the box, use the following commands:

```
vagrant mutate ubuntu/xenial64 bhyve
vagrant mutate freebsd/FreeBSD-11.0-RELEASE-p1 bhyve
```

We can now create a VM using our previously generated journal Vagrantfile by doing the following:

```
vagrant up --provider bhyve
```

This will, as always, update on first boot, so it may take some time. Due to the automatic reboot after update, the VM will shut down and Vagrant will indicate that the VM failed to come up, but if you rerun the `vagrant up --provider bhyve` command, it should work fine. After this, you can run the provisioning step by running `vagrant provision`. And then, just as when we were using VirtualBox, ssh in via `vagrant ssh`. The vagrant-bhyve plugin does not currently support setting up the port forwarding, so that will have to be done manually in this case. You can view the console of the VM by running the `cu -l /dev/nmdm0B` command.

Conclusion

I hope you enjoyed this brief tour of Vagrant, and, when you try it out, I hope you find it as useful as I do!

STEVE WILLS is a husband and father living in North Carolina. He is a FreeBSD ports committer with a focus on Ruby and other programming languages.