Automate Your Development Environment with Vagrant

Get introduced to Vagrant—a tool for managing virtual machines. By taking care of these machines, Vagrant allows developers, operations engineers and designers to get on with their core work.

Vagrant is a tool that helps to manage virtual environments, which means it works with virtual machines (VMs). It provides a simple and easy-to-use command line client for managing these environments, and an interpreter for the text-based definitions of what each environment looks like, called Vagrantfiles. Vagrant is open source, which means that anyone can download it, modify it, and share it freely.

Many virtual machine hypervisors provide their own command line interfaces and, technically, the provisioning of virtual machines through these programs can be done directly or through shell scripts. What Vagrant provides, by adding a layer, is simplicity across multiple systems and a more consistent approach, which could theoretically be used with any virtual environment running on top of any other system.

By providing a common text-based format to work with virtual machines, the environment can be defined in code, making it easy to back up, modify, share and manage with revision control systems like Git. It also means that, rather than sharing a whole virtual machine image, which could be many GBs, every time a change is made to the configuration, a simple text file of just a few kBs can be shared instead.

Why and how Vagrant helps you

It's worth mentioning that Vagrant makes things really easy. I mean, there's no tangle of wires or the need to use Vim, apart from the loads of annoying command line stuff. Vagrant makes it easy to run a development environment. While, at its core, Vagrant provides a rather simple function, it may be useful to a wide range of people working on different kinds of tasks.

For developers, Vagrant makes it easy to create a local environment that mimics the environment upon which your code will eventually be deployed. You can make sure you have the same libraries, dependencies and processes installed, as well as the same operating system and version, and many other details, without having to sacrifice the way your local machine is set up, and without the lag or cost of creating an external development environment and connecting to it.

For operations engineers, Vagrant gives a disposable environment and consistent workflow for developing and testing infrastructure management scripts. You can quickly

test things like shell scripts, Chef cookbooks, Puppet modules, and more, using local virtualisation such as VirtualBox or VMware. Then, with the same configuration, you can test these scripts on remote clouds such as AWS or RackSpace with the same workflow. Ditch your custom scripts to recycle EC2 instances, stop juggling SSH prompts to various machines, and start using Vagrant to bring some sanity into your life.

For designers, Vagrant will automatically set up everything that is required for a particular Web app in order for you to focus on doing what you do best: design. Once Vagrant is configured, developers do not ever need to worry about how to get that app running. You will no longer need to bother other developers to help you fix your environment so that you can test your designs. Just check out the code, type 'vagrant up' and start designing.

With the implication that all designers could get others to do the configuring, I think you'll agree that the "Just check out the code… and start designing" premise is very compelling.

You don't need Vagrant to develop your Web applications on virtual machines. All you need is a virtualisation software package, something like VMware Workstation or VirtualBox, and some code. Download the half-gigabyte operating system image you want and install it. Then download and configure the stack you'll be working with—let's say it is Apache, MySQL or PHP. Next, install some libraries (CuRL and ImageMagick maybe) and, finally, configure the ability to easily copy files from your machine to the new virtual one, somewhat like Samba, or install an FTP server. Once all this is done, copy the code over, import the database, configure Apache's virtual host, restart and keep your fingers crossed.

If you're a bit weird like me, then the above tasks will be pretty easy to do and, secretly, quite a bit of fun.

Or you could use Vagrant. It allows you, or someone else, to specify in plain text how the machine's virtual hardware should be configured and what should be installed on it. It also makes it insanely easy to get the code on the server. So check out your project, type 'vagrant up' and start work.

Installing Vagrant

Step 1 - Installing VirtualBox: First, download VirtualBox and install it. On *nix systems (Mac OSX, Linux, etc), you will need to modify your *.bash_profile* (or *.zsh_profile*) to extend your *$PATH* variable:

```
PATH=$PATH:/Applications/VirtualBox.app/Contents/MacOS/
export PATH
```

This will allow Vagrant to know where VirtualBox is installed, and this, of course, will vary for different operating systems. For Windows, installing VirtualBox is enough.

Step 2 - Installing Vagrant: To install Vagrant, just go to *https://www.vagrantup.com/downloads.html* and download the installer, which will automatically add Vagrant to your system path so that it is available in the terminals. If it is not found, please try logging out and logging back into your system (this is particularly necessary sometimes for Windows).

Configuring Vagrant

Vagrant will create a custom virtual machine. To do this, Vagrant will need a template, or base VM, to copy and customise. The Vagrant development team is nice enough to provide a number of base Linux VMs. To install one, run the following command:

```
$ vagrant box add precise32 http://files.vagrantup.com/
precise32.box
```

This will download a base VM of Ubuntu 12.04 32-bit, which is about 230MB in size and will be referred to by the name'precise32'. Next, you'll need a project to work on, and a development environment configuration. Here's a Node.js demo; clone this repository, and move into the project directory, as follows:

```
$ git clone https://github.com/Srijancse/Vagrant-Node
$ cd Vagrant-Node
```

Take a look at the directory:

```
vagrant-node-mongo /
   README.md
   Vagrantfile
   /app
   /cookbooks
```

Inside the 'app' sub-directory, there is a simple *node.js* app that returns some text output to an HTTP request. In 'cookbooks' there are some Chef cookbooks, or collections of recipes that define automated configuration commands to install Node.js, MongoDB, apt-get, and other essential tools. Let's take a closer look at the file named *Vagrantfile*. This is the Vagrant configuration file, which defines how to set up your VM project. It should look like what's shown below:

```
Vagrant::Config.run do |config|
  config.vm.box = "precise32"
  config.vm.forward_port 3000, 3000
  config.vm.share_folder "app", "/home/vagrant/app", "app"
  # allow for symlinks in the app folder
  config.vm.customize ["setextradata", :id, "VBoxInternal2/
SharedFoldersEnableSymlinksCreate/app", "1"]
  config.vm.customize ["modifyvm", :id, "--memory", 512]
  config.vm.provision :chef_solo do |chef|
    chef.cookbooks_path = "cookbooks"
    chef.add_recipe "apt"
```

```
    chef.add_recipe "mongodb"
    chef.add_recipe "build-essential"
    chef.add_recipe "nodejs::install_from_package"
    chef.json = {
      "nodejs" => {
        "version" => "0.8.0"
        # uncomment the following line to force
        # recent versions (> 0.8.4) to be built from
        # the source code
        # , "from_source" => true
      }
    }
  end
end
```

Let's go through the *config* file to understand what it actually does. First, it tells Vagrant to use the base box named 'precise32' -- the one which we downloaded. Then we configure port forwarding between the Vagrant VM and our host over port 3000, and a shared folder in which we will put our application code. Then we configure some VM hardware settings, such as RAM configuration. Finally, we configure the automation system (*chef_solo*, in this case), and add the recipes from our */cookbooks* directory, defining the applications and services we want to install in the Vagrant VM. Note that I can also pass in specific configurations in JSON format—I'm using this to specify the exact version of Node.js (0.8.0) I want to install.

To power on your custom VM, from inside your application directory issue the following command:

```
$ vagrant up
```

If this is the first time you have issued this command for a new project, Vagrant will actually create the VM from scratch, which may take some time. The next time you enter this command, it will just power the VM on, which is substantially faster. When the 'vagrant up' command is issued, Vagrant works in the background to make the VM required for that environment stand up (as defined in the *Vagrantfile*). If you open up the VirtualBox UI you can actually watch the VM appear, but there's no real reason to do so -- you won't need the UI to interact with your VM. Instead, once the VM is up, type the following command:

```
$ vagrant ssh
```

This will automatically give you an SSH session in Vagrant without the need to enter a hostname or credentials. The console output looks similar to what is shown in Figure 1.

Now, to verify the installation of Node, NPM and MangoDB, use the following commands:

```
$ node-v
```

```
mbp15:vagrant-node-mongo constantinecois$ vagrant ssh
Welcome to Ubuntu 12.04 LTS (GNU/Linux 3.2.0-23-generic-pae i686)

 * Documentation:  https://help.ubuntu.com/

121 packages can be updated.
51 updates are security updates.

Welcome to your Vagrant-built virtual machine.
Last login: Fri Sep 14 06:22:31 2012 from 10.0.2.2
vagrant@precise32:~$
```

Figure 1: Vagrantoutput

```
$ npm -v
$ mongo -version
```

If you use an *ls*, you will also notice a directory */app* located in the Vagrant home directory. Inside will be the application code -- these files are actually located on the host system, but are visible on the Vagrant VM as a VirtualBox shared directory. This is where you will run the code. The shared directory means you can pull up Atom in the host OS, and edit the files that will be run in the Vagrant VM. You could start the test application now to see it running, but to get an optimal workflow, you may want something that will restart your application when changes are made to the code. To get this functionality in Node.js, go ahead and install a supervisor on your Vagrant VM from the SSH command line:

```
$ sudo npm install supervisor -g
```

Now, to launch the application, use the command shown below:

```
$ supervisor app/app.js
```

Open up a text editor or IDE in the host system, and open the *vagrant-node/app/app.js* file for editing. Now here's where Vagrant plays a cool part. Open a browser in the host system, and navigate to *http://localhost:3000*. Since the *node.js* app is running in your Vagrant VM, the result will be what's shown in Figure 2.

The application is running in the Vagrant VM, but one can test it in the host browser, as if it was running locally! Now, go to your text editor and modify the *app.js* code to read as follows:

```
var http = require('http');

server = http.createServer(function(req, res) {
  res.writeHead(200, {"Content-Type": "text/html"});
  res.write("<html><body><h1>Hello, from Vagrant!</h1></
body></html>");
  res.end();
}).listen(3000);
```

Refresh the browser and you can see the changes made in the running application.

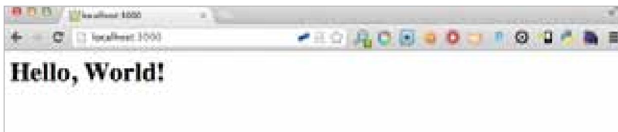

Figure 2: Vagrant-hello-world

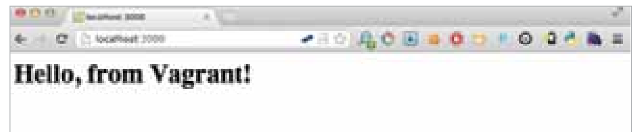

Figure 3: Vagrant-modified

This is the real power of Vagrant – you can code on the local machine using local tools. You can also run the application on a custom controlled Vagrant VM with short commands, and test the apps using your favourite browsers and testing tools. Vagrant handles the full management of the VM. When you're done working, just exit the Vagrant SSH session with the following command:

```
$ exit
```

…and instruct Vagrant to power down the VM as follows:

```
$ vagrant halt
```

Then you can move onto another project. You can have a Vagrant VM for every project.

This is almost a revolution for team workflows. One can check *Vagrantfile* and cookbooks into source control, just as we did with the GitHub application, and can rest assured that each member of the team will develop code with exactly the same runtime environment. If you change something in the runtime environment for the application, each developer issues a *vagrant destroy* command, as shown below:

```
$ vagrant destroy
```

It then creates a new VM with the updated configurations.

By using Vagrant you can be absolutely certain that you'll never again hear, "It works on my machine," because you'll know that every machine has exactly the same code as yours. END

References

[1] *https://en.wikipedia.org/wiki/Vagrant_(software)*
[2] *https://www.vagrantup.com/*

By: Srijan Agarwal

The author is a developer at WikiToLearn, and an open source enthusiast. You can contact him at *srijanagarwal.cse@gmail.com*.