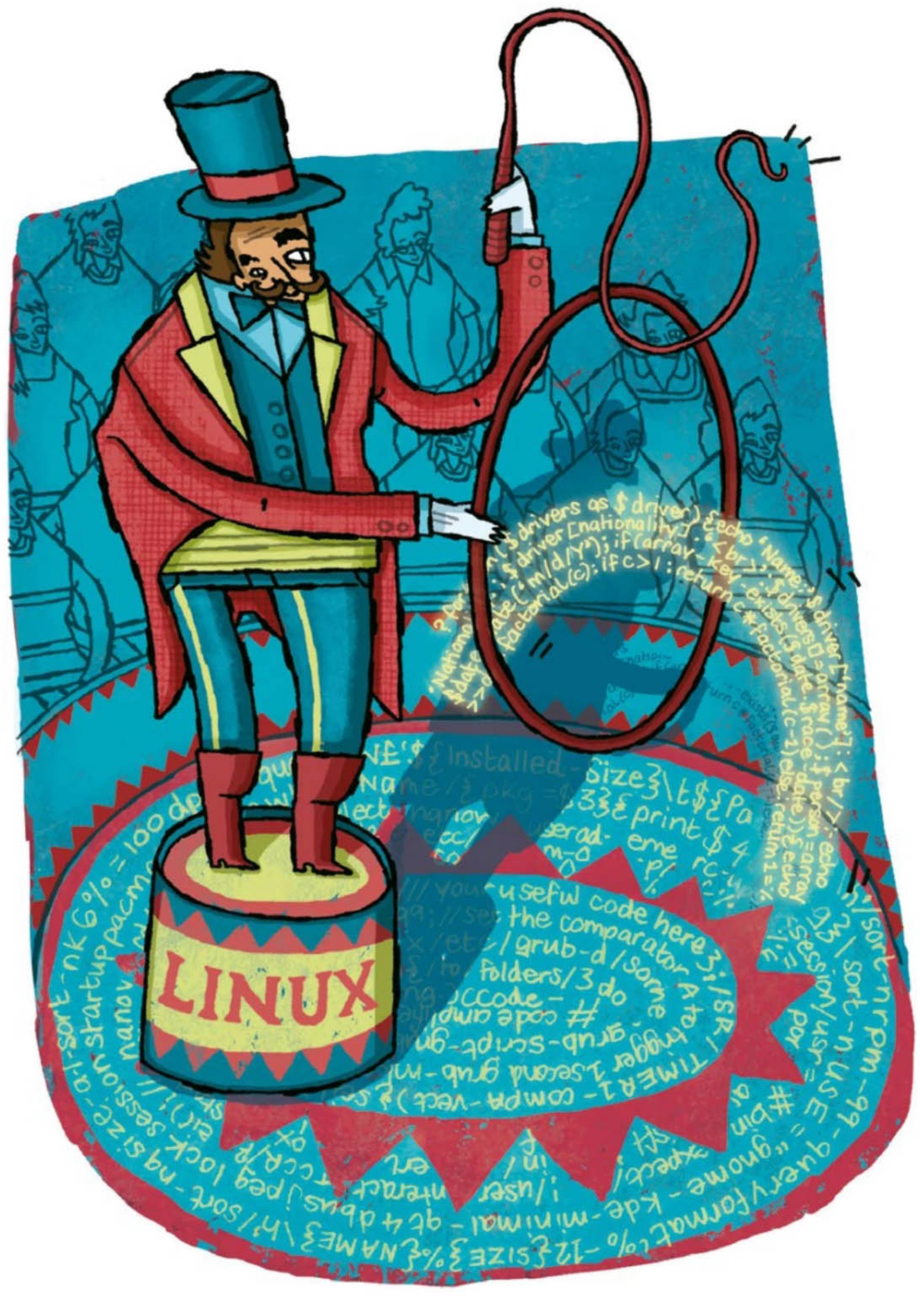
Vagrant: VMs made easy

Bobby Moss shows you how to try the latest distros and run non-Linux applications without risking your home setup.

Our expert

Bobby Moss develops cloud microservices for a global IT consultancy and its clients. In his spare time he works on free software projects and tinkers with old hardware.



When you hear the name 'Vagrant', you might be wondering what rough sleepers have to do with managing virtual machines (or VMs, as they're known). You would be in good company, because as far as strange project names go, this has to rank favourably with competition like 'Kafka', 'Avogato' and 'Mustache'.

To clarify, Vagrant is a common scripting language for managing and provisioning virtual machines across different virtualisation tools like *Virtualbox* and *VMware Workstation*. It relies on drivers to translate the instructions you provide in the configuration (such as your choice of operating system, number of processor cores, amount of RAM or a specific suite of software you need), and uses these instructions to create VMs that match your expectations. Anything you can do manually on these platforms can be fully automated so long as Vagrant has a driver for it. Some of these are supported directly by Hashicorp (the company behind Vagrant) but most are developed by the wider community.

As you might imagine, this platform-neutrality makes it a very popular tool in IT departments across the globe.

Quick tip

If Vagrantfiles seem familiar, it's because they follow the same syntax as the Ruby programming language. Luckily, you don't need to be an experienced developer to change your VM settings since you will simply be adding, removing and editing key-value pairs.

Being able to spin up multiple VMs and bring them down again across different virtualisation tools and cloud environments on demand with a common framework is very useful to sysadmins, testers and software developers.

In this tutorial, however, we will be focusing on how you can leverage the power and flexibility of Vagrant at home to test out the latest Linux distributions, run software your system wasn't originally designed for, and even make those pesky Windows applications you just can't do without work flawlessly and safely without risking the setup that you use to play games, surf the web, edit documents and complete other everyday tasks.

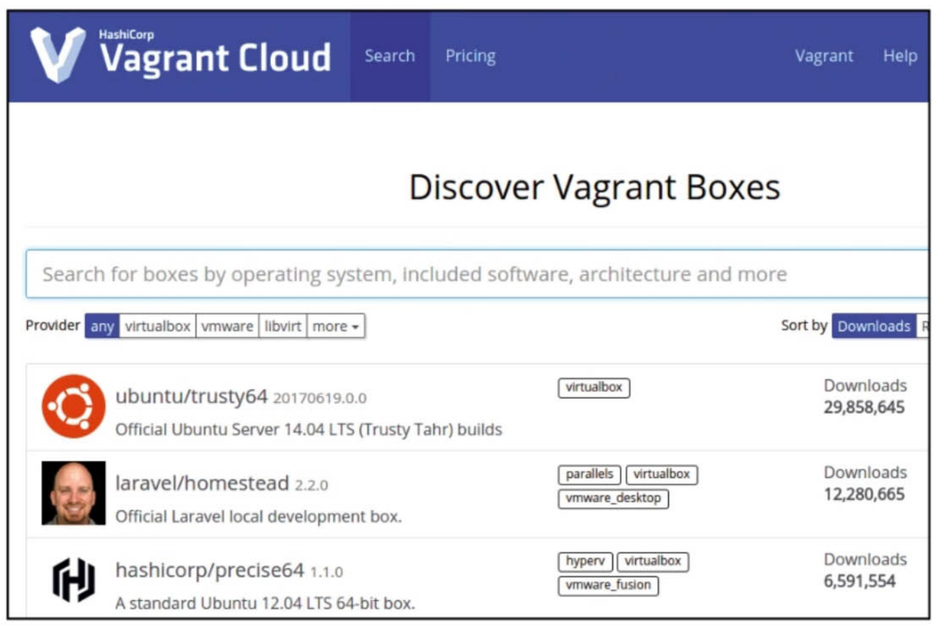
Building boxes

Before we begin, it's important to note that the versions of Vagrant and *Virtualbox* you use through this tutorial will be relevant. This is because the driver the former needs to be updated, for it to support recent versions of the latter, so you might find that the tools you download through your distro's usual repositories are not compatible with each other.

If you want all the latest and greatest 'bleeding edge' features, it is recommended you download the relevant packages directly from the project websites at **http://bit.ly/1XvPzB6** and **http://bit.ly/1fPLZt5**. You will also need to ensure you install *Virtualbox*'s optional extension pack to benefit from all of Vagrant's features, such as automatic folder sharing and support for a wider array of hardware configurations.

Once these are successfully installed, you should fire up your terminal and type in the following lines:

```
$ mkdir test
```



› You can find boxes to download for almost any purpose from Hashicorp themselves and the wider community.

Vagrant vs Docker

Docker is a containerisation technology that isolates applications from each other while using the same kernel. The main benefit, if you're not a developer, is that it can enable you to run an application packaged specifically for Ubuntu on Fedora without compatibility problems. However, it is worth bearing in mind that it can still make changes to the host system from within docker, so you should be mindful about the packages you choose to run this way on your home setup.

Much like Vagrant there is such a thing as a "DockerFile" that works in a similar way to a Vagrantfile. You can find out more information about this via **http://dockr.ly/1PallBK**.

Docker is one of the virtualisation technologies that Vagrant supports, but you can also use your Vagrantfile to deploy several Docker containers onto a single Virtualbox VM. This means that rather than choosing between the two, you can benefit from extra flexibility and lower hardware resource requirements by running them both in tandem. In that setup you would only need one guest OS to run multiple applications, but it does mean that if the VM stops then Docker does too.

› Vagrant doesn't just automate the installation of virtual machines, it enables you to try out shiny new Linux distributions without breaking your bootloader.

```
$ cd test
$ vagrant init
```

You will notice that inside this new test folder you created, there is a 'Vagrantfile'. This is the main configuration file that will determine the attributes of all the virtual machines you create from this particular directory. If you want to make changes to a VM, this is the file you should edit, and if you want to spawn more than one VM, you would create a new directory and Vagrantfile for each.

The full anatomy of a Vagrantfile is described in Figure 1 (see *bottom-right*), but the file you generate will probably look a little different because it will include all kinds of extra comment lines we omitted for brevity. But in this file you can add port forwarding if you are hosting some specific networked software like *Apache*, add extra folder shares between guest and host and define any of the hardware attributes of the VM you generate. Later in this tutorial we will cover how to use it to deploy extra software to the guest too using configuration management tools such as *Ansible*.

As we are using *Virtualbox* you will notice in our example that there is an extra `modifyvm` line for functionality that has yet to be included in the driver. This works because Vagrant is offering a common wrapper for the VBoxManage service, which itself provides a command line interface for all the functionality that is normally available through the GUI. You can see a full reference for all the commands that are available to you from **http://bit.ly/1HUjufW**, and if you want to make use of it this any additional lines would always follow the same pattern.

Now we understand how a basic Vagrantfile works, let's move over to a completely new directory and run the following commands:

```
$ vagrant init jhcook/fedora25
$ vagrant up
```

This Vagrantfile will look a little different to the last one. The `config.vm.box` line now contains the value `jhcook/fedora25` instead of `base` because we have specified the box we are interested in generating at the end of our "init" line. For the purposes of our tutorial we are initially launching an instance of Fedora Workstation 25. You can search for boxes with all kinds of different operating systems and pre-defined purposes from **http://bit.ly/2v7xTUe**.

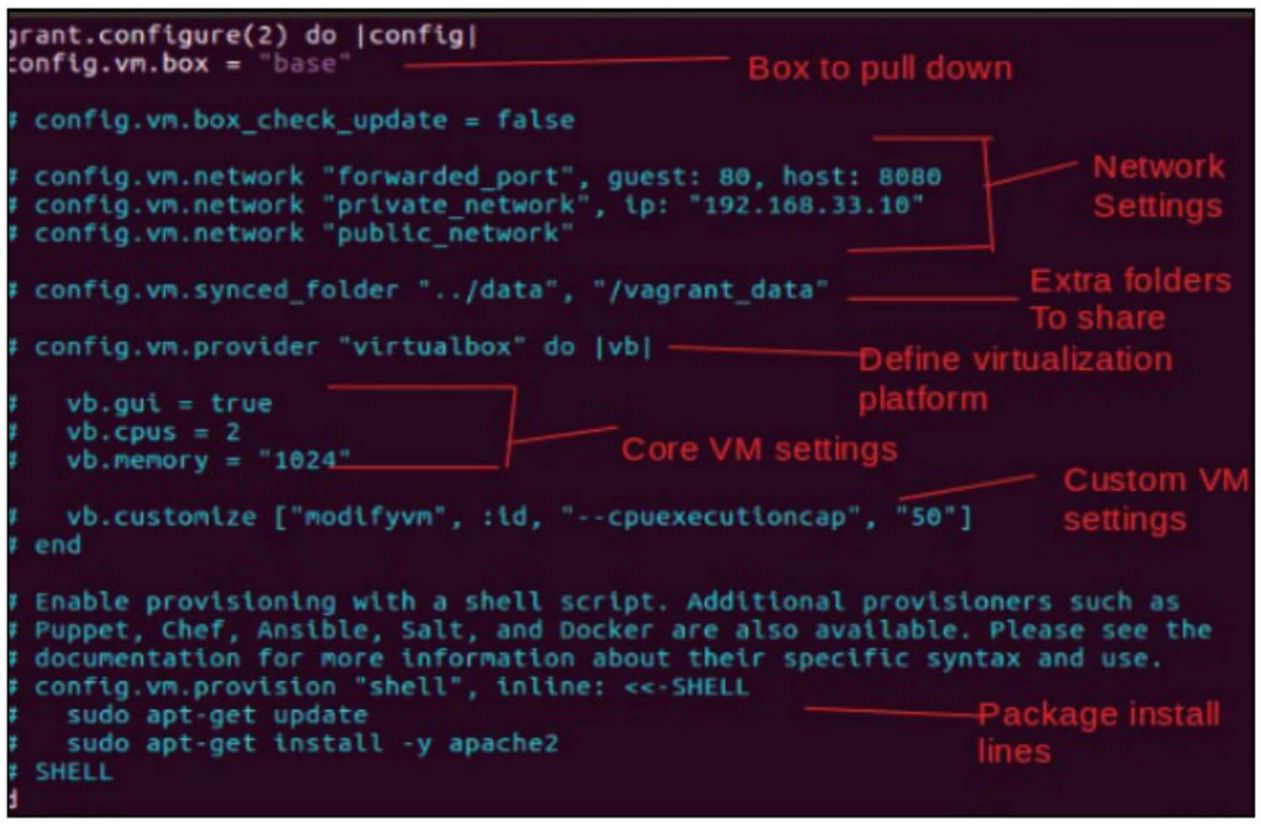
The first thing you will notice is the initial launch does take quite a while. This is because Vagrant is downloading a "box" file that contains the VM and its default settings from a remote server, so your mileage will vary according to your broadband connection.

However, if you open up the Virtualbox GUI you will see a very curious thing has happened. The virtual machine is clearly there, it's turned on and is definitely running even though the GUI never launched!

This is because the Virtualbox GUI disabled by default, and with the current Vagrantfile config you would have to manually open it from *Virtualbox* itself. While this is very helpful for developers creating web services, this is far from ideal for our purposes. Thankfully you can fix this quickly and easily by uncommenting some lines in your Vagrantfile so it contains the following code:

```
config.vm.provider "virtualbox" do |vb|
  vb.gui = true
end
```

Save and close the file, then restart the virtual machine by running the following command from the same directory as »



› Figure 1: This annotated image displays the full anatomy of a Vagrantfile that's used to create VM attributes.

your new Vagrantfile:

```
$ vagrant reload
```

You should now find when the VM restarts that Fedora's familiar Gnome desktop automatically pops up for you when it is ready to use. You will find that the default 'vagrant' user's password is the same as its username.

You can do much more than just launch and restart a virtual machine with vagrant commands. The following hibernates, wakes up and shuts down the virtual machine:

```
$ vagrant suspend
$ vagrant resume
$ vagrant halt
```

It is recommended you run the "suspend" or "halt" command before you shut down or restart your PC to avoid corrupting the guest operating system. If you're in any doubt about the current running state of the virtual machine that you're running, simply type the following command to find out what is happening:

```
$ vagrant status
```

Finally if you want to completely delete the virtual machine regardless of its current running state, then you should use the following command:

```
$ vagrant destroy
```

You will notice that the second time you create your VM will be much quicker than the first because Vagrant is smart enough to store an extracted version of the "box" file you downloaded earlier to your hard drive. If you set the correct option in the Vagrantfile it will scan the server for updates.

It is also worth noting that if you copy any file into the same folder as your Vagrantfile then so long as the guest operating system is Linux you should find that same file shared in the **/vagrant** folder. You can either verify this through the desktop file manager of the VM you just created or by using SSH from the host machine:

```
$ vagrant ssh
$ sudo su -
$ ls -lrt /vagrant
```

Simply type "exit" twice to end the SSH session. Whenever you are logged in to the guest operating system you can elevate vagrant user to root without providing a password when you run the second command.

This is the default behaviour because the virtual machines you generate are designed to be disposable and used only for as long as you need them.

This makes sense given you can always spin them back up again after you destroy them, thanks to the Vagrantfile configuration that you have created.

If you want to keep the VM running for much longer periods it is recommended you change the vagrant user's password and create a new user account for everyday use.

Ansy for Ansible

Vagrant isn't restricted to simply creating virtual machines and switching them on and off. We can take things a step further by automatically installing useful applications on top of our base box when we run the "vagrant up" command.

There are a number of ways you can do this. You could add some shell commands at the bottom of the Vagrantfile, but this usually needs to be altered for different distributions because they will use different package managers or store files in odd locations. Thankfully, we have other options.

In the world of DevOps there are systems that can centrally manage the installation and configuration of different packages across servers and VMs in a more platform-neutral way. The most common of these are Chef, Puppet and Ansible. We'll talk more about these systems later on in this tutorial.

The easiest way to enable Ansible in your Vagrantfile is to add the code below before the final `end` statement:

```
config.vm.provision "ansible_local" do lansiblel
  ansible.playbook = "playbook.yml"
end
```

Because we are using `"ansible_local"` rather than `"ansible"` you don't need to have it installed already on your home setup. Vagrant is smart enough to try and install it for you on the guest operating system if it isn't there by default when you create the VM.

As you might be able to guess from the middle line you will need to create a YAML file called **playbook.yml** in the same directory as your Vagrantfile. To make it useful you should add the following three lines to the top of the file:

```
---
- hosts: all
  become_user: root
  become: yes
```

Now we have a basic Ansible confguration, we can start creating "tasks" for the system to do. Let's destroy your Fedora VM and change the following line in your Vagrantfile:

› Scripted installations from Ansible let you quickly install and configure packages on your box. This is particularly useful if you break your VM!

Ansible and Fedora

Unfortunately, if you're using Fedora 23 or later as a guest OS Ansible support is slightly broken. This is because the distro no longer ships with Python 2 and Ansible needs this for the RPM install module to work. As a consequence, if you try to run Ansible you will simply get error messages about missing Python modules.

Until Ansible migrates to Python 3 (a release supporting it was under active development at the time of writing) you can work around this problem by sending raw commands to the package manager instead:

```
- name: Upgrade all packages
  raw: dnf -y update
- name: Install Chromium
  raw: dnf -y install chromium
```

The downside to this approach is Ansible has no awareness of whether your applications have been installed successfully or not, and it also has no idea whether your system is running the latest versions. As a result it will launch these tasks every time you run 'vagrant up', which is far from ideal! This is why you should always prefer proper package manager tasks over sending raw commands.

```
config.vm.box = "boxcutter/ubuntu1604-desktop"
```

This will pull down a shiny new Ubuntu desktop, but the box it is built from might be out of date. It makes sense therefore to update all the packages and (as an added bonus) install *Chromium* on our new VM. You can do this by adding some more lines to your YAML file:

```
tasks:
  - name: Upgrade all packages
    apt: upgrade=dist
  - name: Install Chromium
    apt: name=chromium-browser state=latest
```

Once the new file has been saved and closed, simply run the following command to apply your changes:

```
$ vagrant provision
```

This can also work with Yum-based distros if you swap *apt* for *rpm* and shorten *chromium-browser* to *chromium*.

While your Ansible tasks should automatically run when you create a new box, you can also manually force it to happen with the following line:

```
$ vagrant up --provision
```

Spin your own

If you are more than a little squeamish about the idea of downloading unknown boxes from Vagrant community websites then you will be glad to hear that it is very straightforward to roll your own Vagrant boxes for *Virtualbox* VMs. Let's walk through how to make this happen.

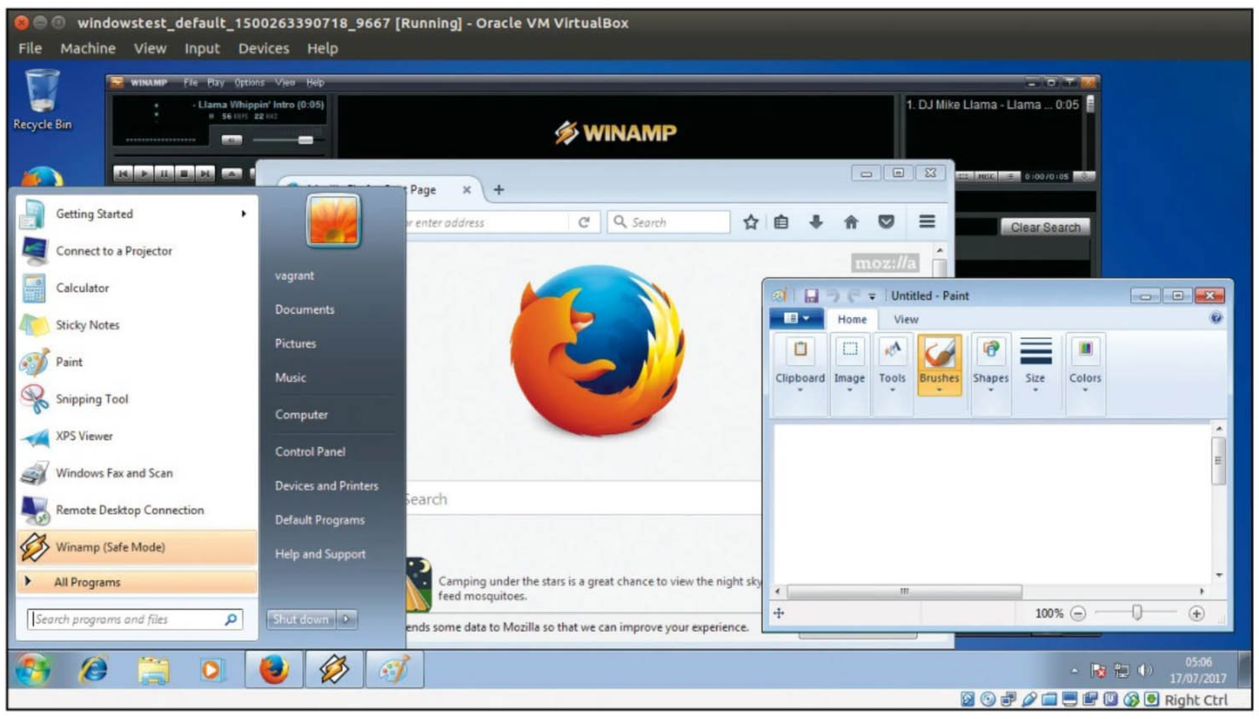
We mentioned before that you could potentially use Vagrant to manage Windows VMs you want to run old software on. Because of license restrictions non-server versions of Windows are not made freely available on Vagrant boxes by the community anyway, so this seems as good an opportunity as any to make our own!

First you will need to create a virtual machine called **windowsvm** through the VirtualBox GUI. If you are using Windows 7 or above you may want to ensure you allocate at least 1GB (ideally 2GB) of RAM and create a 25GB virtual hard disk for it. Feel free to add more than one core and allocate as much video memory as you can spare.

You will need to run through the usual setup steps for your installation media. Once Windows is running, you should ensure you install any relevant updates and services packs, as well as 'Guest Additions' from Virtualbox's 'Devices' menu.

There are some extra steps that you can perform to ensure your box is ready for Vagrant to make the best use of it. A full list of recommendations, such as disabling UAC, setting certain registry entries and fully configuring WinRM (Windows' own management interface) is available via **http://bit.ly/2vsLepQ**.

Once you have shut down your virtual machine, you should find the files you need in **~/Virtualbox VMs/windowsvm**. You can clear out any files that don't have a .vdi or .vbox or .vmdk extension, since the only things you need now are you hard disk file and the core configuration.



When only Windows will do and Wine isn't compatible enough to run nasty proprietary software, managing a Windows virtual machine (VM) with Vagrant can at least save you time and hassle.

The next step is to create a new file called "metadata.json". You only need to add the following line to it so Vagrant will know which virtualization platform you are targeting:

```
{"provider":"virtualbox"}
```

You should also create an empty file called "Vagrantfile" in the same directory. We will be using Vagrant's own packaging command so won't need to set anything (such as MAC addresses) manually inside it. You should now navigate to your home directory and run the following command:

```
$ vagrant package --base windowsvm
```

Once the export is complete you should be able to add the box to your current Vagrant install and run it for the first time.

```
$ vagrant box add package.box --name windowsvm
$ mkdir windowstest
$ cd windowstest
$ vagrant init windowsvm
$ vagrant up
```

If everything went well then Vagrant should create a brand new VM based on your previous one. You will need to enable the GUI in the VagrantFile just like you did for the Fedora box to see the Windows desktop, but you can check it's running from *Virtualbox* itself. If you want to avoid the SSH timeout you can install OpenSSH in your base box, but while it looks untidy it is perfectly safe to ignore.

There are still some steps you will need to do manually in the Virtualbox GUI, like setting up folder shares or changing hardware resource allocations. Also, using the "halt" vagrant command is the equivalent of powering it off without shutting down. However, this barebones box is still capable of running any Windows software you simply can not do without. LXF

Quick tip

Vagrant can be extended with plugins. These are useful when you're trying out new virtualization technologies or seeking to speed up package installs and reduce the amount of required configuration. You can find more information on what plugins can do and how to install them from **http://bit.ly/2tfl4Kp**.

Where's Wine?

For those who are unfamiliar, Wine is an emulation layer that sits between a Windows program and Linux, and attempts to translate the instructions between them in real time. You can install supporting libraries and customise compatibility settings on an application by application basis using separate wine "prefixes" that behave like their own install directories.

Unfortunately, because the community doesn't have access to the source code for Windows, this is far from a perfect system. While there has been progress made on compatibility for common applications such as *Microsoft Office, Internet Explorer* and *Adobe Creative Suite*, your mileage will vary. You can find out how well an application is likely to run from **https://appdb.winehq.org**.

Another downside is it's not isolated from the host system, so you could find your home setup is exposed to malware designed for Windows. The benefit of running Windows in a VM is you avoid compatibility problems and run applications in a fully isolated environment. However, this will need more memory and processing power, because you're running a full operating system in addition to your application.