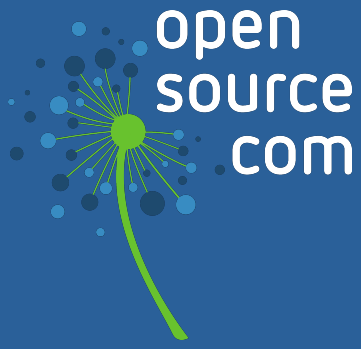


Everyday Virtualization

We are Opensource.com

Opensource.com is a community website publishing stories about creating, adopting, and sharing open source solutions. Visit Opensource.com to learn more about how the open source way is improving technologies, education, business, government, health, law, entertainment, humanitarian efforts, and more.

Do you have an open source story to tell? Submit a story idea at opensource.com/story

Email us at open@opensource.com

Supported by
Red Hat

Table of Contents

Getting started with GNOME Boxes virtualization

By Alan Formy-Duval

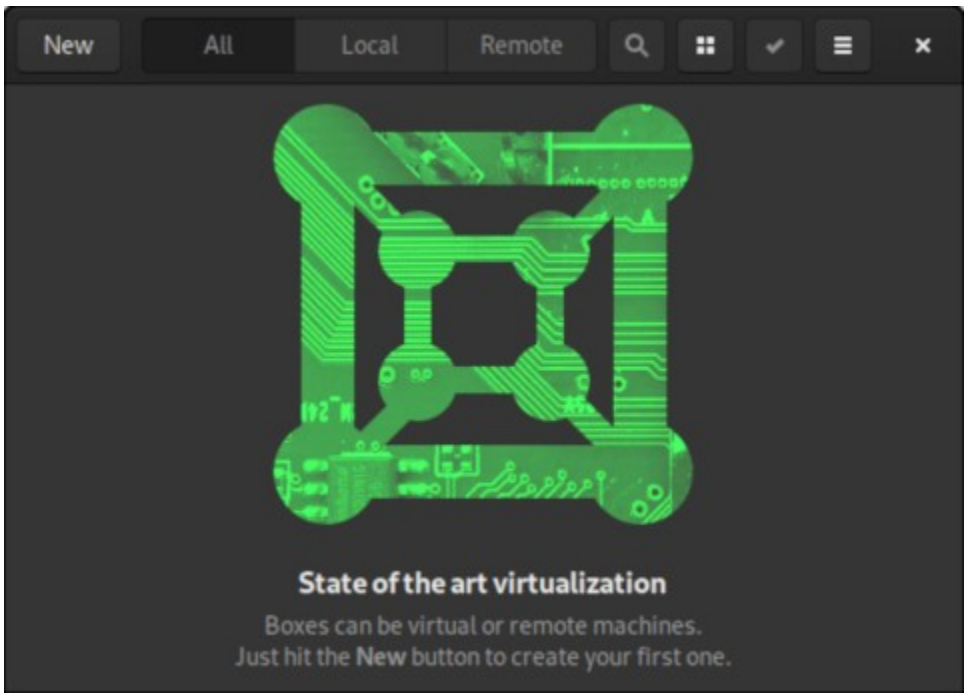
I've been a fan of virtualization technology for many years, using many different products along the way. Virtualization has advantages for both the data center and the desktop: data centers use it to increase server hardware utilization, while desktop users use it for modeling, testing, and development work. One operating system running on top of a different one on the same hardware, all thanks to the concept of a virtual machine (VM).

I recently upgraded my laptop to the latest Fedora Workstation Edition. I noticed GNOME Boxes, simply titled **Boxes**, in my application menu. The GNOME Project—whose members are the creators and maintainers of the GNOME Desktop Environment—describes GNOME Boxes as: "A simple GNOME application to view, access, and manage remote and virtual systems." Of course, I had to check this tool out.

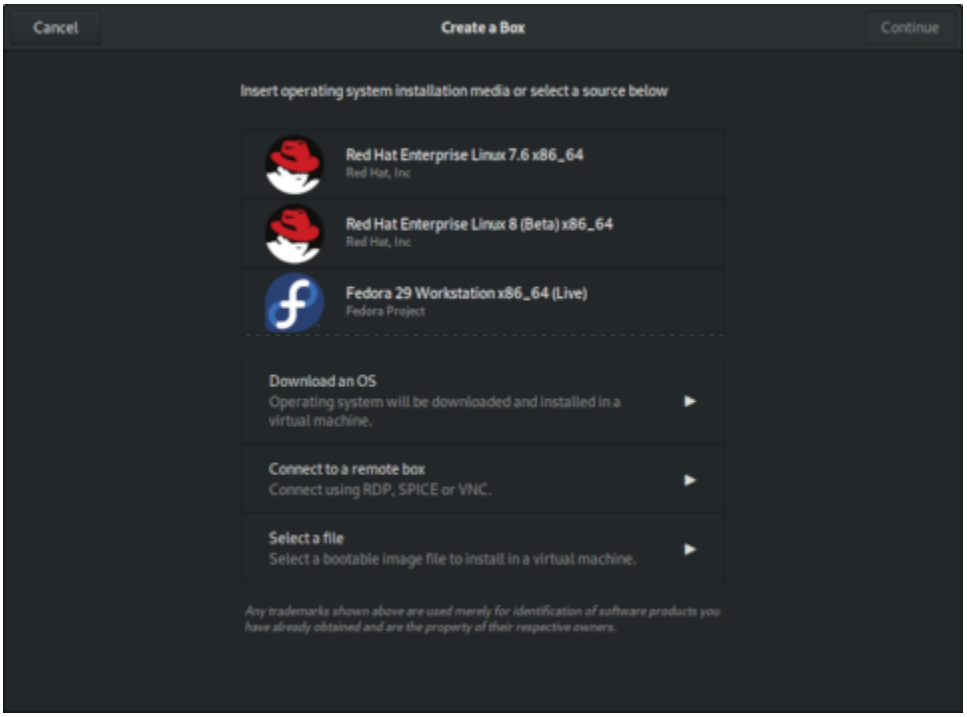
This two part series article will cover two of the main features of Boxes. Because the GNOME Boxes project refers to a VM as a "box," I'll use that terminology.

Create a box

When you launch Boxes, it opens to its main window:

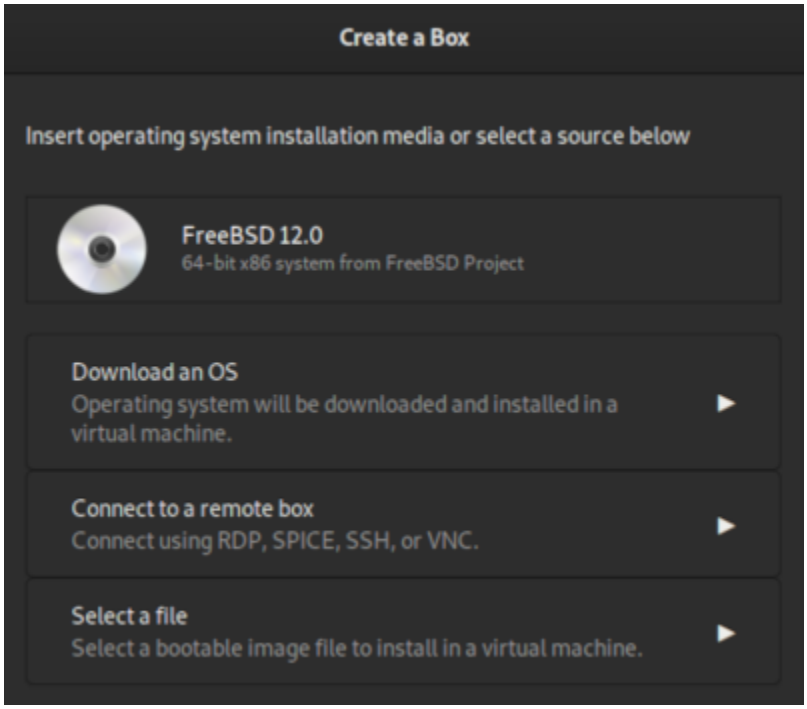


Start using Boxes by clicking the **New** button at the top-left corner of the application. This opens a dialog titled **Create a Box**. The first time you run Boxes, you'll see the following screen. Click **Continue**.



You see several options. You can download an operating system (OS), connect to a remote box, or select a file.

The OS list at the top of the screen above is the default. The list could be different if you have any OS ISO files in your Downloads directory. This is because Boxes detects ISO files and creates the OS list accordingly. For example, if I have the ISO file for FreeBSD 12 (FreeBSD-12.0-RELEASE-amd64-disc1.iso) in Downloads, it's included on the list, as shown below.



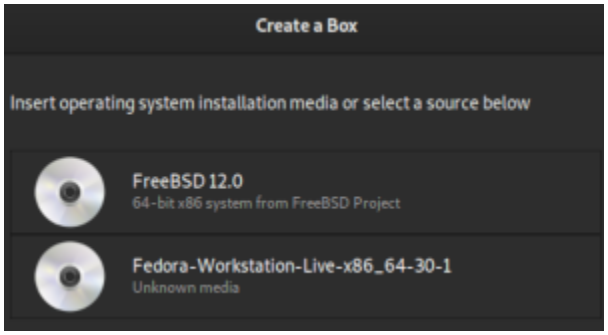
If you click **Download an OS**, you can choose an OS from a larger list with many options. The ISO file for the OS that you select will be saved to your Downloads directory.

Create a box with an external ISO

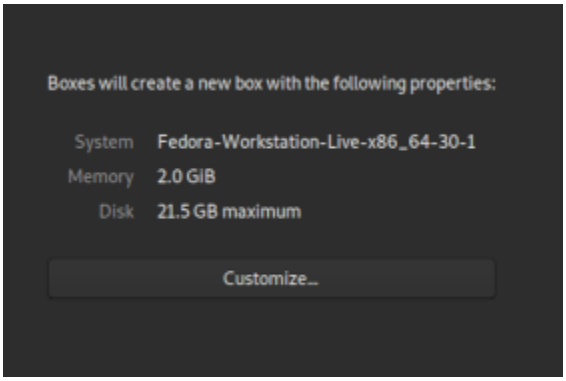
I chose to create my first box with a version of Fedora that, at the time, wasn't available in Boxes' default OS list. The version I wanted, at the time, was Fedora 30. I saved the Fedora 30 Workstation Live CD ISO file to my Downloads directory.

```
[alan@workstation Downloads]$ ls -l
-rw-r--r-- 1 alan 193.. May  2 20:08 Fedora-Workstation-Live-x86_64-30-1.2.iso
-rw-r--r-- 1 alan 892.. May  7 17:00 FreeBSD-12.0-RELEASE-amd64-disc1.iso
```

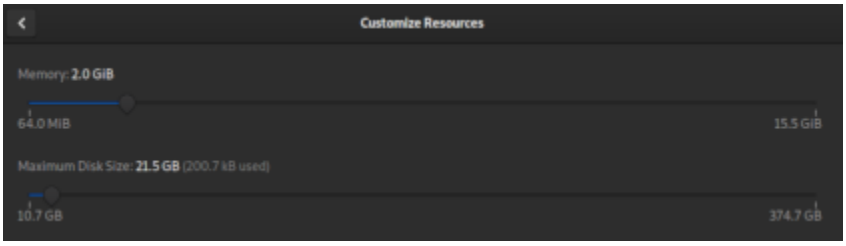
Now when Boxes launches, the OS list includes Fedora 30. Click **Fedora-Workstation** to begin.



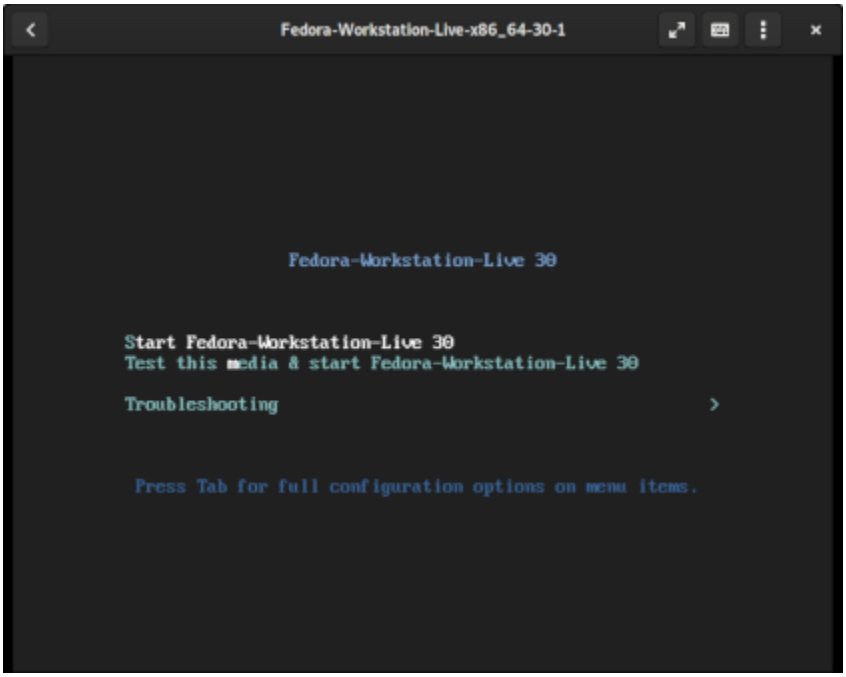
The next screen, called **Review**, shows the Memory and Disk properties for the new box you created.



Clicking **Customize** will allow you to adjust the Memory and Disk sizes.



To move on, click **Create** on the top-right of the window. The new box will be created and booted.



Once the Fedora Live CD boots, you can complete the installation as you usually would on bare metal.

The final thing I did was to rename the box in the properties of the box.

My opinion

I'm impressed. GNOME Boxes is great for users who want to quickly deploy various operating systems and software for development, experimenting, and learning tasks with minimal effort. This is as GNOME intended.

GNOME Boxes is comparatively simple. A more advanced configuration of hardware, network devices, and CPU features would require a tool like virt-manager. Your usage needs will determine whether you might need that level of customization.

In Part 2, I'll cover the remote access capabilities of Boxes.

Download Fedora 30 Workstation

To download Fedora 30 Workstation, browse to the Fedora website. On this page, put the mouse cursor over **WORKSTATION** on the left-hand side, and click **Download Now**.

The next screen provides the link to download the Fedora 30 Workstation 64-bit ISO file, which is 1.9GB. I saved the file *Fedora-Workstation-Live-x86_64-30-1.2.iso* to my Downloads directory.

An introduction to Virtual Machine Manager

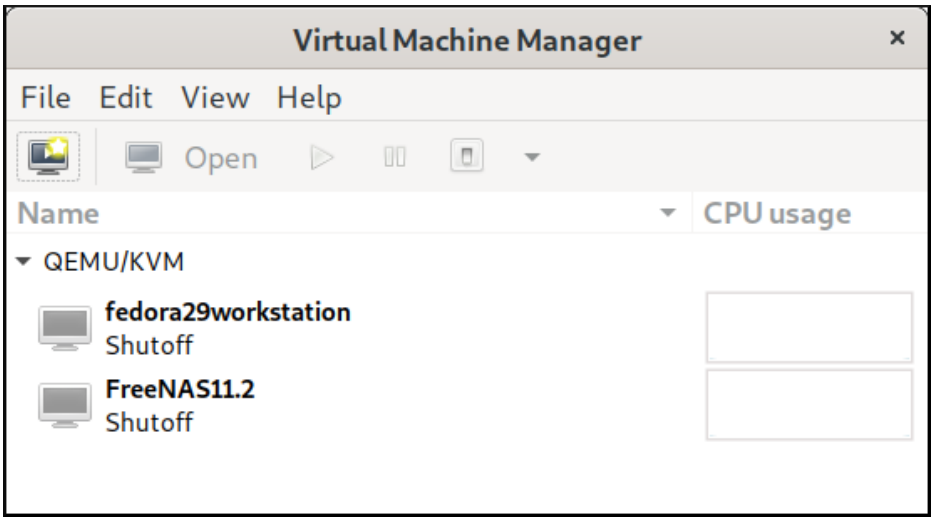
By Alan Formy-Duval

In my series about GNOME Boxes, I explained how Linux users can quickly spin up virtual machines on their desktop without much fuss. Boxes is ideal for creating virtual machines in a pinch when a simple configuration is all you need.

But if you need to configure more detail in your virtual machine, you need a tool that provides a full range of options for disks, network interface cards (NICs), and other hardware. This is where Virtual Machine Manager (virt-manager) comes in. If you don't see it in your applications menu, you can install it from your package manager or via the command line:

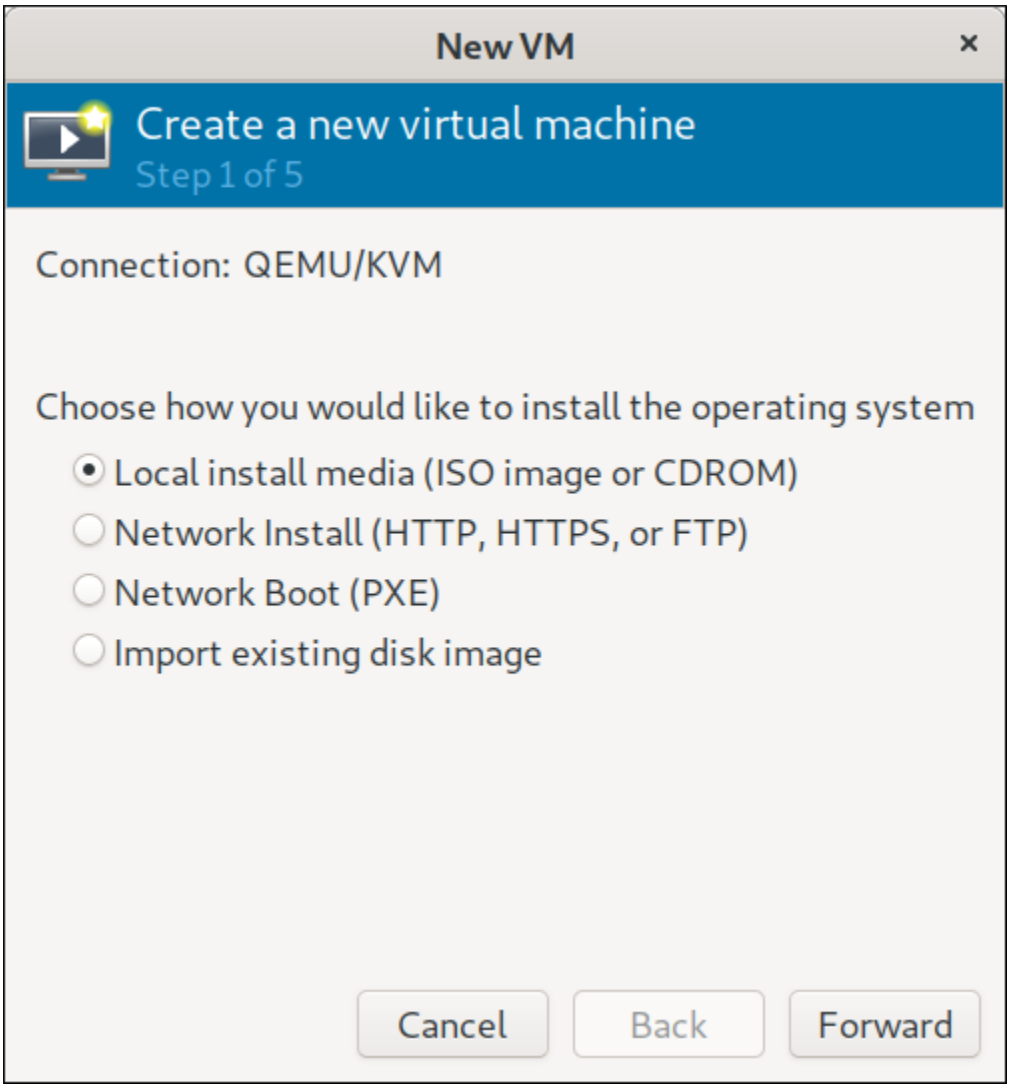
- On Fedora: **sudo dnf install virt-manager**
- On Ubuntu: **sudo apt install virt-manager**

Once it's installed, you can launch it from its application menu icon or from the command line by entering **virt-manager**.

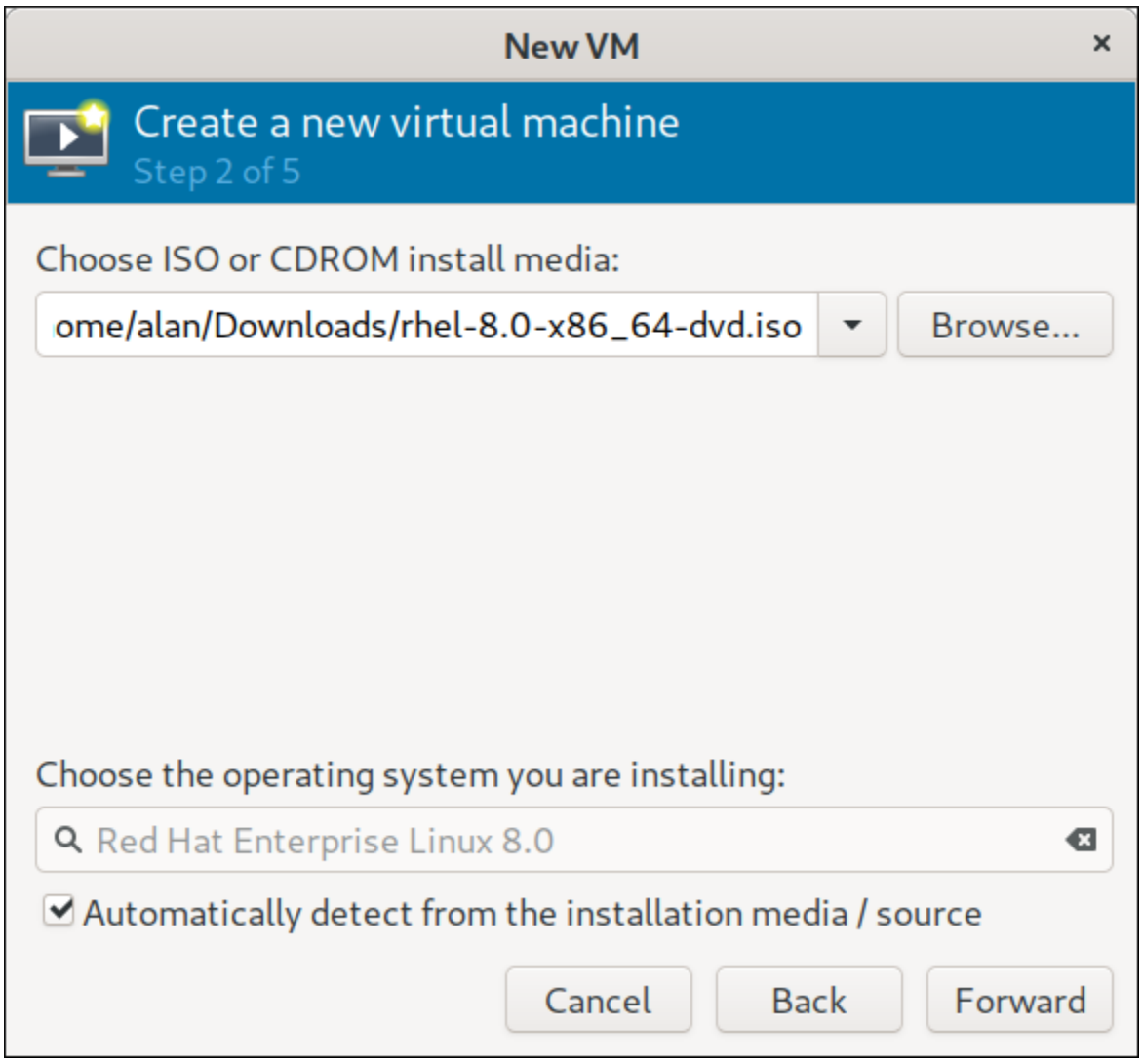


To demonstrate how to create a virtual machine using virt-manager, I'll go through the steps to set one up for Red Hat Enterprise Linux 8.

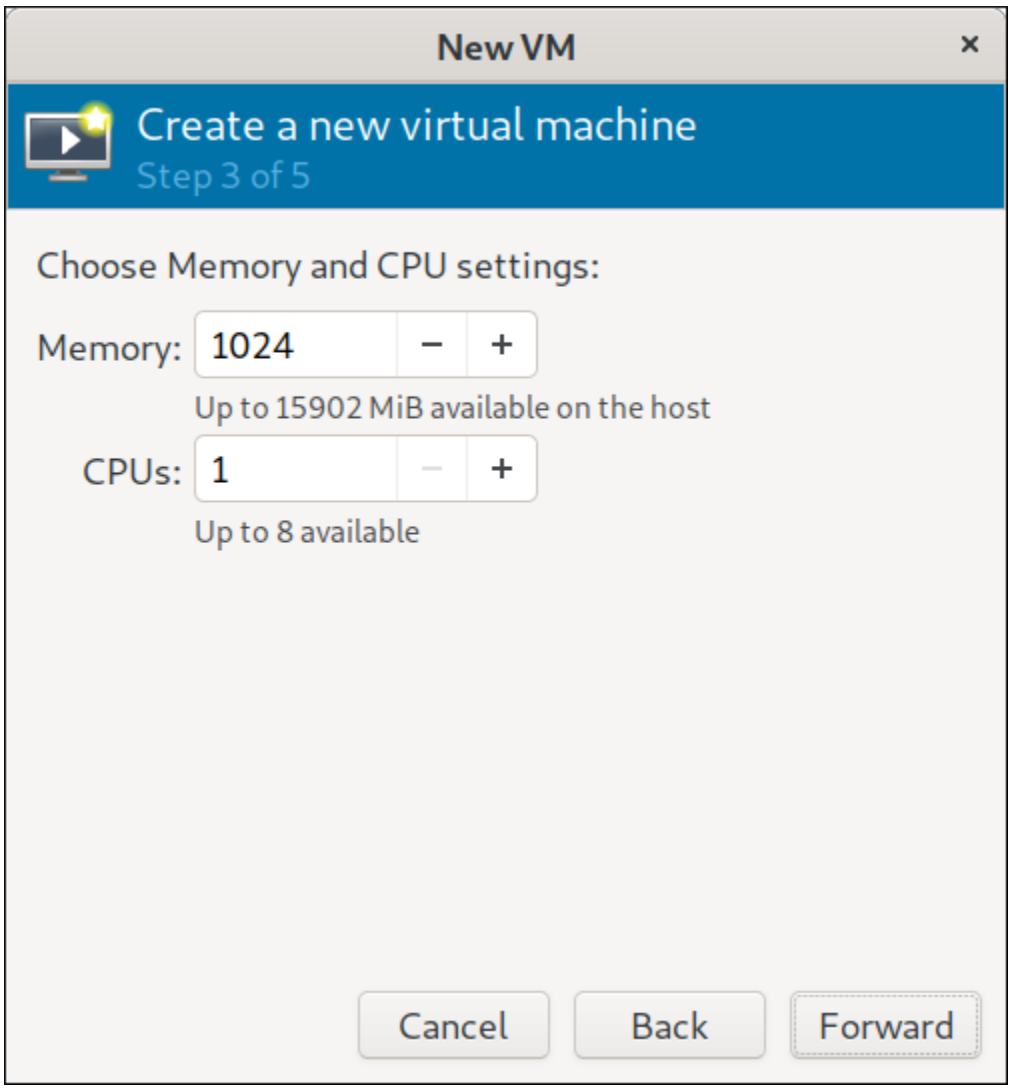
To start, click **File** then **New Virtual Machine**. Virt-manager's developers have thoughtfully titled each step of the process (for instance, "Step 1 of 5") to make it easy. Click **Local install media** and **Forward**.



On the next screen, browse to select the ISO file for the operating system you want to install. (My RHEL 8 image is located in my Downloads directory.) Virt-manager automatically detects the operating system.

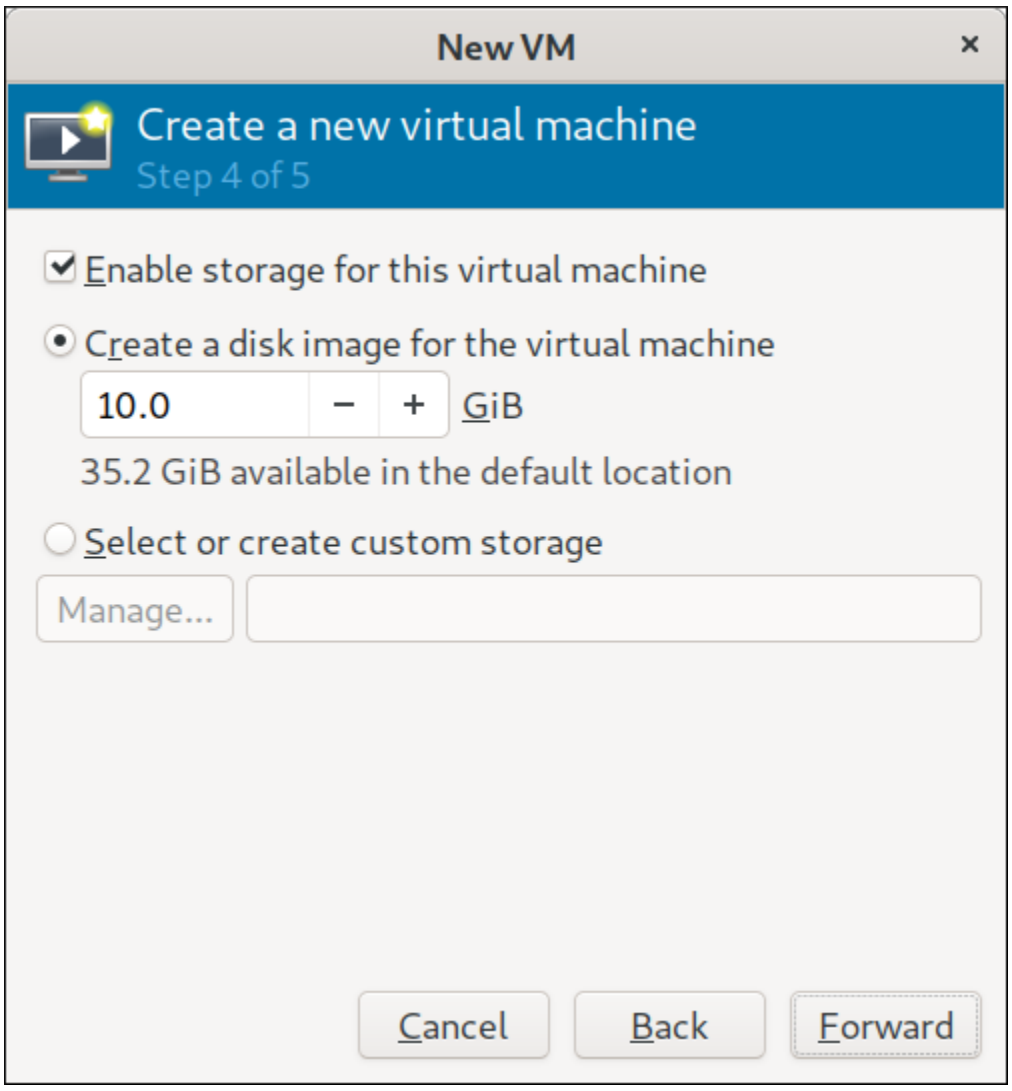


In Step 3, you can specify the virtual machine's memory and CPU. The defaults are 1,024MB memory and one CPU.



I want to give RHEL ample room to run—and the hardware I'm using can accommodate it—so I'll increase them (respectively) to 4,096MB and two CPUs.

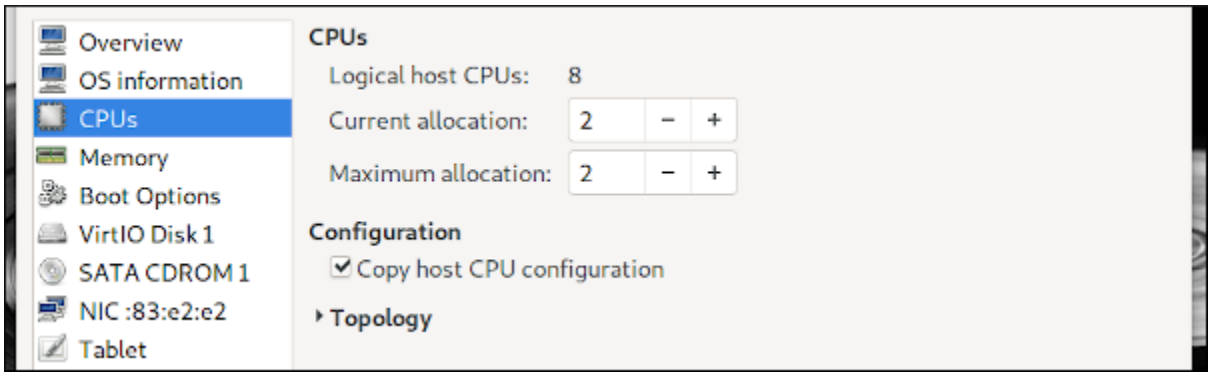
The next step configures storage for the virtual machine; the default setting is a 10GB disk image. (I'll keep this setting, but you can adjust it for your needs.) You can also choose an existing disk image or create one in a custom location.



Step 5 is the place to name your virtual machine and click Finish. This is equivalent to creating a virtual machine or a Box in GNOME Boxes. While it's technically the last step, you have several options (as you can see in the screenshot below). Since the advantage of virt-manager is the ability to customize a virtual machine, I'll check the box labeled **Customize configuration before install** before I click **Finish**.

Because I chose to customize the configuration, virt-manager opens a screen displaying a bunch of devices and settings. This is the fun part!

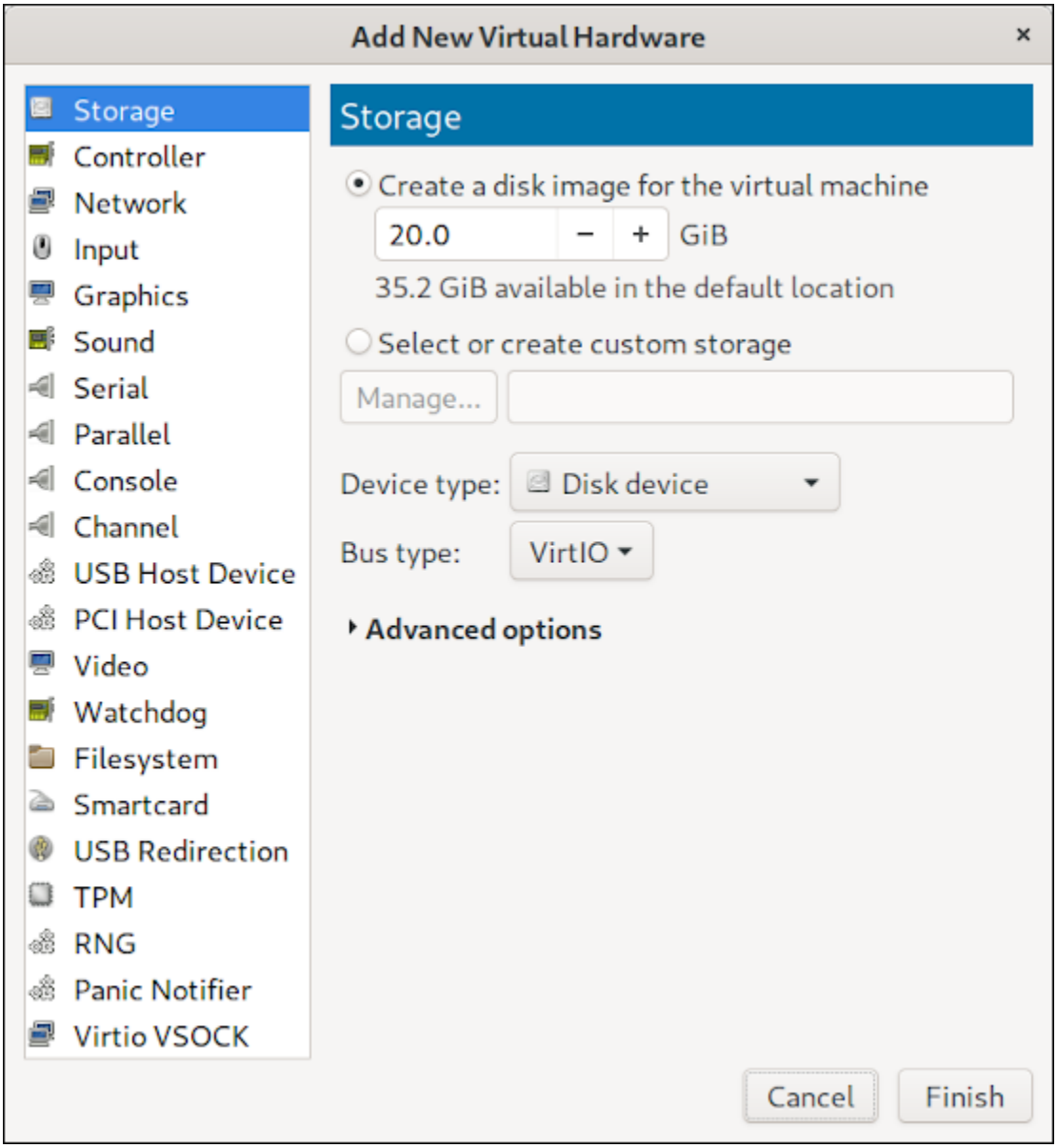
Here you have another chance to name the virtual machine. In the list on the left, you can view details on various aspects, such as CPU, memory, disks, controllers, and many other items. For example, I can click on **CPUs** to verify the change I made in Step 3.



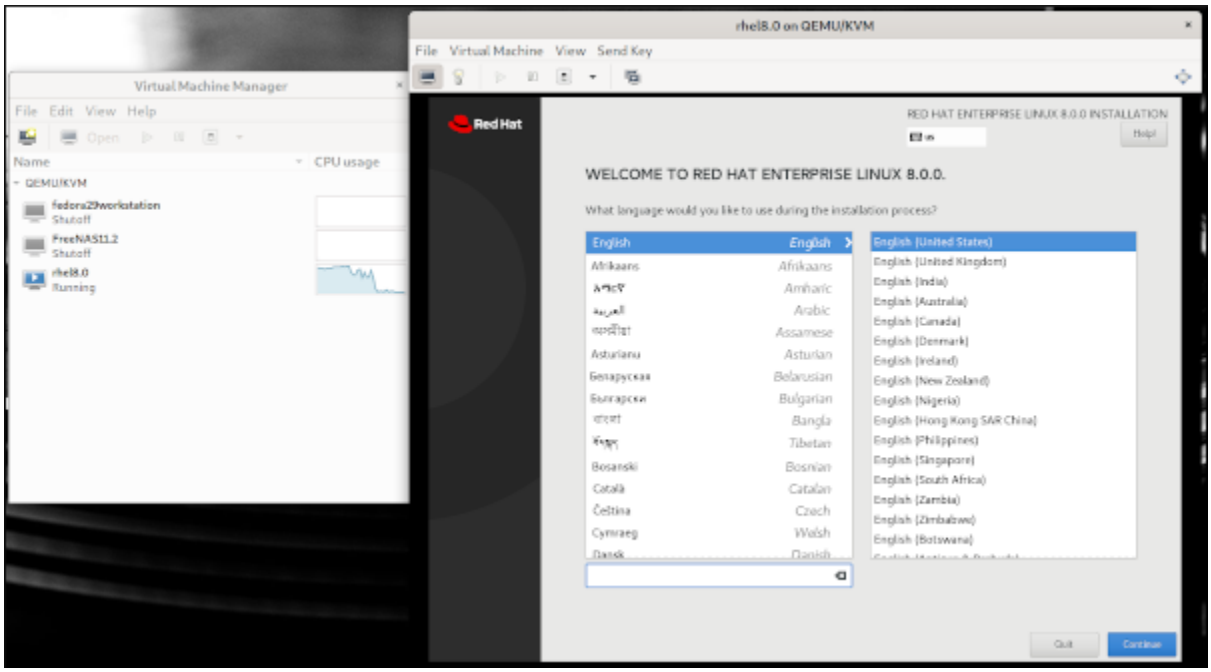
I can also confirm the amount of memory I set.

When installing a VM to run as a server, I usually disable or remove its sound capability. To do so, select **Sound** and click **Remove** or right-click on **Sound** and choose **Remove Hardware**.

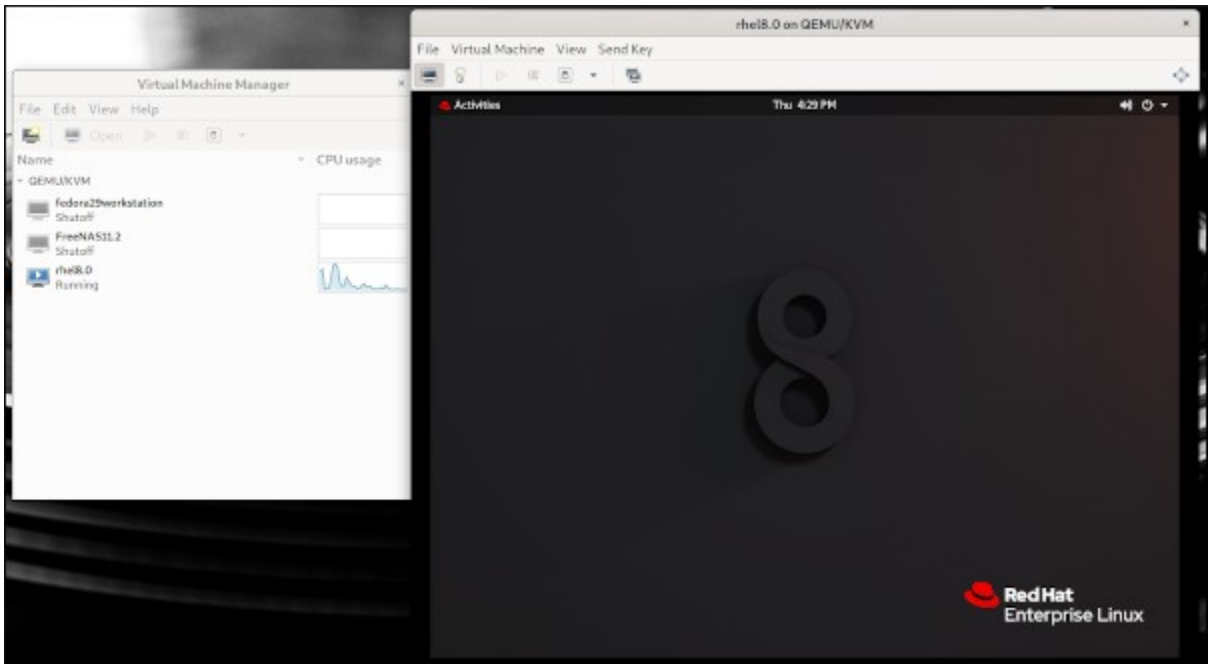
You can also add hardware with the **Add Hardware** button at the bottom. This brings up the **Add New Virtual Hardware** screen where you can add additional storage devices, memory, sound, etc. It's like having access to a very well-stocked (if virtual) computer hardware warehouse.


Add New Virtual Hardware
Storage
Controller
Network
Input
Graphics
Sound
Serial
Parallel
Console
Channel
USB Host Device
PCI Host Device
Video
Watchdog
Filesystem
Smartcard
USB Redirection
TPM
RNG
Panic Notifier
Virtio VSOCK
Storage
Create a disk image for the virtual machine
20.0
GiB
35.2 GiB available in the default location
Select or create custom storage
Manage...
Device type:
Disk device
Bus type:
VirtIO
Advanced options
Cancel
Finish

Once you are happy with your VM configuration, click **Begin Installation**, and the system will boot and begin installing your specified operating system from the ISO.



Once it completes, it reboots, and your new VM is ready for use.



Virtual Machine Manager is a powerful tool for desktop Linux users. It is open source and an excellent alternative to proprietary and closed virtualization products.

How to use GNOME Boxes' snapshot capability

By Alan Formy-Duval

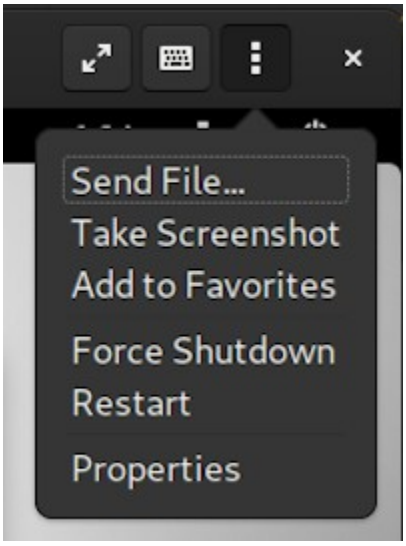
In the first article in this series about GNOME Boxes, I explained how to get started with the virtualization application, and in the second article, I described GNOME Boxes' remote access capabilities. Here in the third installment, I will cover GNOME Boxes' snapshot functionality, which is a useful way to preserve data quickly.

Snapshot technology, which has been implemented in databases, filesystems, and operating systems, is extremely useful with virtual machines (VMs). Taking a snapshot of a VM preserves its state at a specific point in time. Restoring or reverting the snapshot returns the VM to that state, regardless of any changes made after the snapshot was taken. This capability can be useful for conducting tests of new software or patches and also when something has gone horribly wrong. As a virtualization tool, GNOME Boxes has this feature.

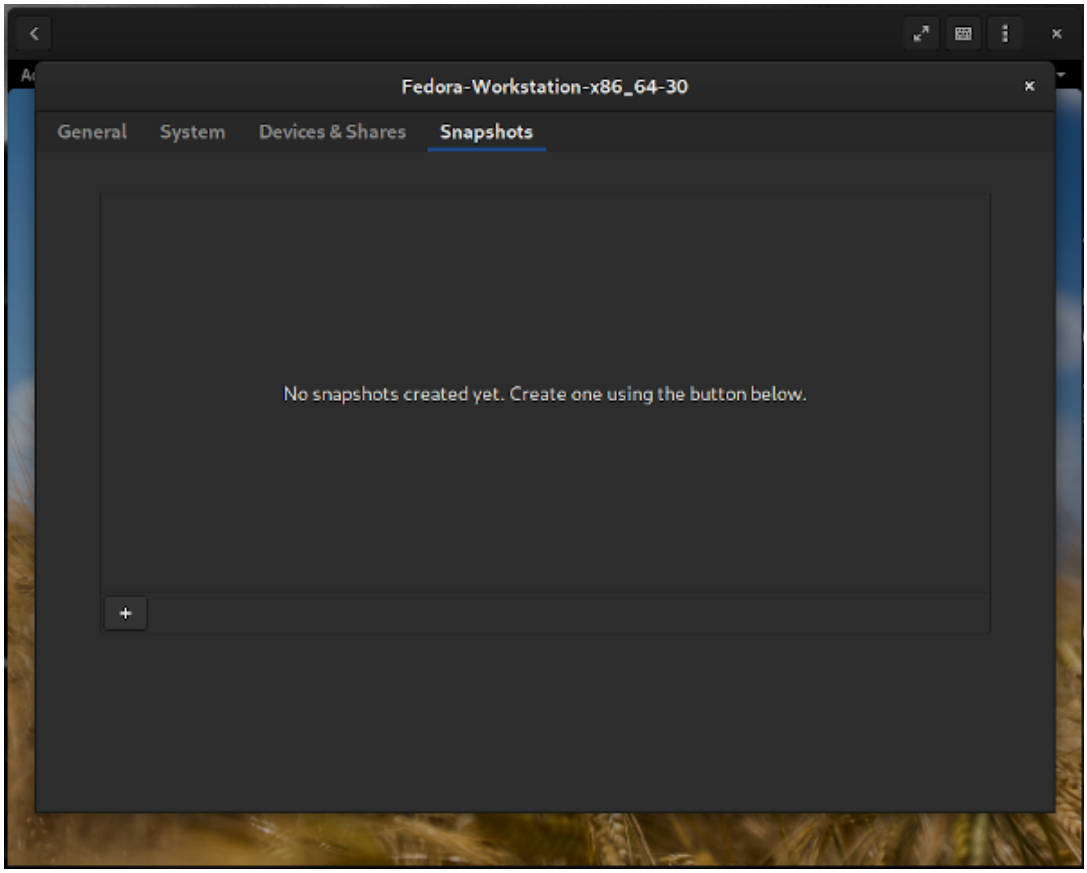
Time travel

Snapshots allow you to move forward and backward in time. You can save the Box at various points and revert to those points whenever we want to revisit one. There is one important thing to remember, though. The current, running state of a Box can be considered the active state. When you revert to any snapshot, you will lose the active state. So, if you intend to (or think you might) return to the current state, you must take a new snapshot prior to reverting to another one. Note that this feature is not applicable to remote Boxes.

To access the snapshot feature, click the **Properties** menu on a local Box. If you are on the main screen, right-click on the Box. If you are viewing a Box, you can access the menu from the button in the top-right corner. Then click **Snapshots** at the top of the Properties screen.



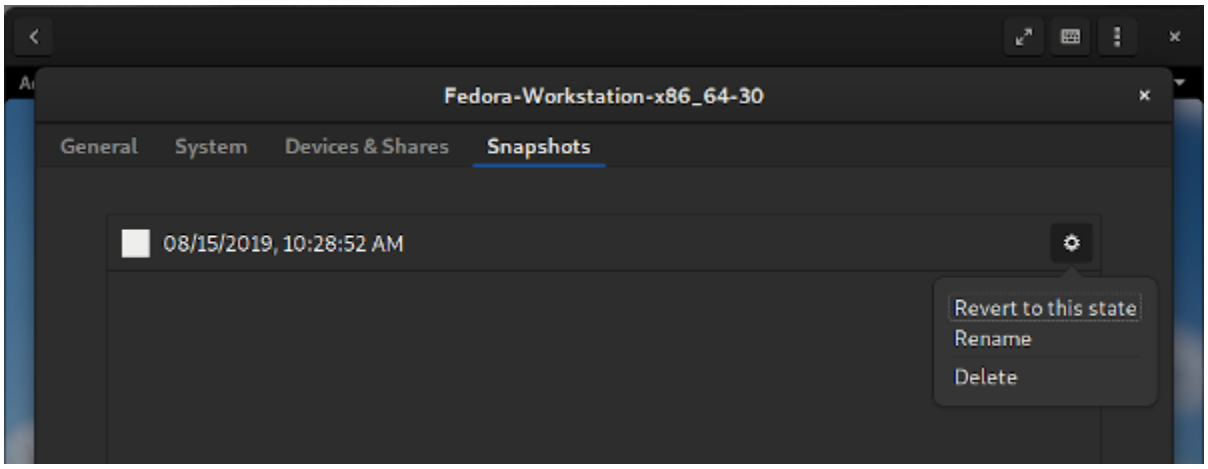
The Properties menu of a local box



Snapshots screen for a local box

Creating a snapshot

The Snapshots screen shows all existing snapshots and a small plus (**+**) button at the bottom for creating new ones. Go ahead and click the plus button. The first thing you see is "Creating new snapshot..." and a progress indicator. Be patient, as creating a snapshot can take a few minutes. When it's complete, the new snapshot appears with the date and time it was created and a small gear icon. This icon brings up the **Actions** menu for reverting, renaming, or deleting the snapshot.



A snapshot and its action menu

That's it! You now have a snapshot of your Box.

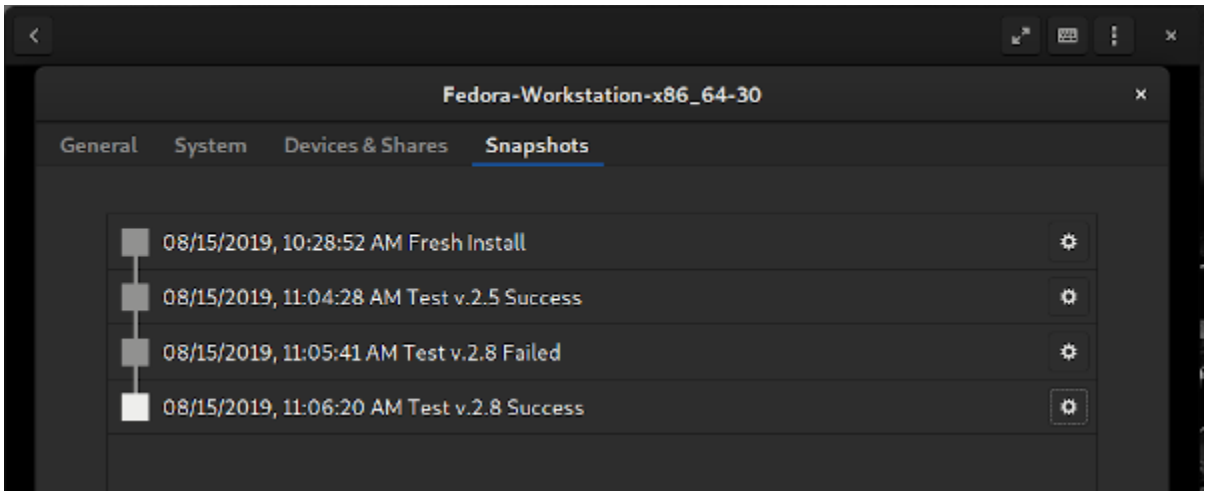
Reverting a snapshot

Suppose you are a tester. Each time a new version of your product is built, you need to conduct user acceptance testing. You could create a new VM every time or replace the old version with the new one. That can be a time-consuming process. Removing an old version may not completely remove all remnant files, which could taint future testing. This is where a snapshot comes in incredibly handy. Simply revert to a previous, "known-clean" state before each new test to ensure the integrity of the test.

Or suppose you made some changes to your VM and ran a few scripts, then you realize some important files in your home directory were mistakenly deleted. You don't need to worry, though. You can go back to the Snapshots screen, select the **Actions** menu for the snapshot named **08/15/2019, 10:28:52 AM**, and click **Revert to this state**. The Box will be returned to the exact point when this snapshot was taken.

Renaming a snapshot

As you create more snapshots, it can get confusing to keep track of them based only on the creation date and time, which is Boxes' default naming convention. For this reason, it is good to rename your snapshots to make them more descriptive. Just open the **Actions** menu for the snapshot and select **Rename** from the drop-down.



Snapshots renamed to be more descriptive

Deleting a snapshot

If you decide you no longer need to keep a snapshot, it's easy to delete it. Just go back to the snapshot's **Properties** and select **Delete** from the menu. Deleting a snapshot will not interfere with any other snapshot or the active state of the Box.

As I have shown in my previous Boxes articles, you can have one or more VMs—or as GNOME Boxes calls them, Boxes.

How to use GNOME Boxes' remote access capabilities

By Alan Formy-Duval

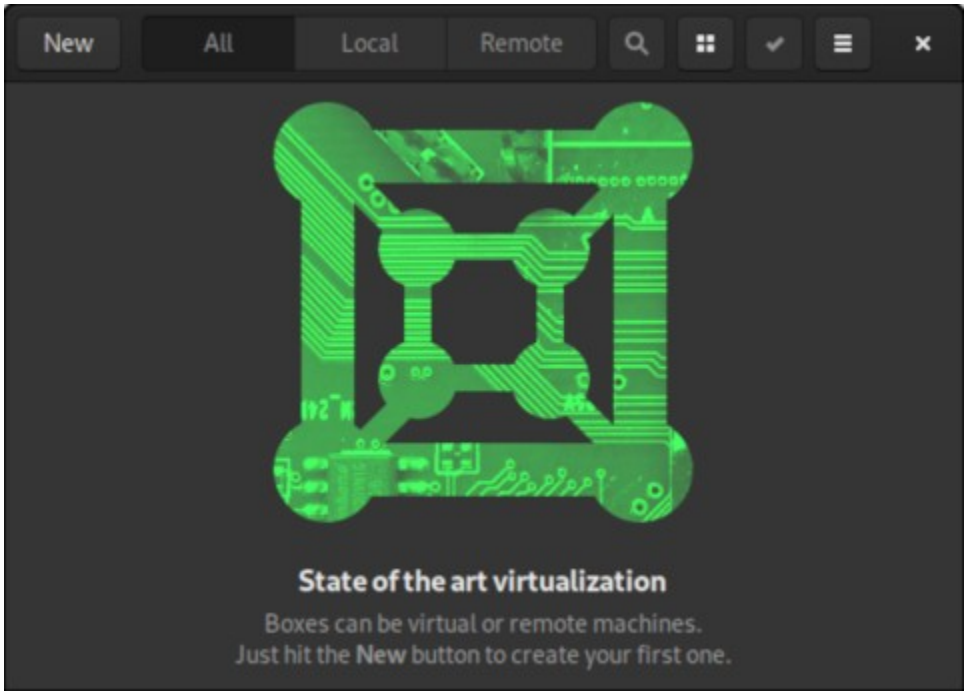
In the first chapter of this book, I introduced GNOME Boxes, an open source virtualization tool maintained by the GNOME Project as part of its GNOME Desktop Environment. The GNOME Project describes Boxes as: "A simple GNOME application to view, access, and manage remote and virtual systems."

Boxes is not a one-trick pony. In addition to quickly creating a box locally, you can also connect to remote systems, both physical and virtual, using various protocols. Boxes' main screen then displays both local and remote boxes in a way that brings them together for easier access.

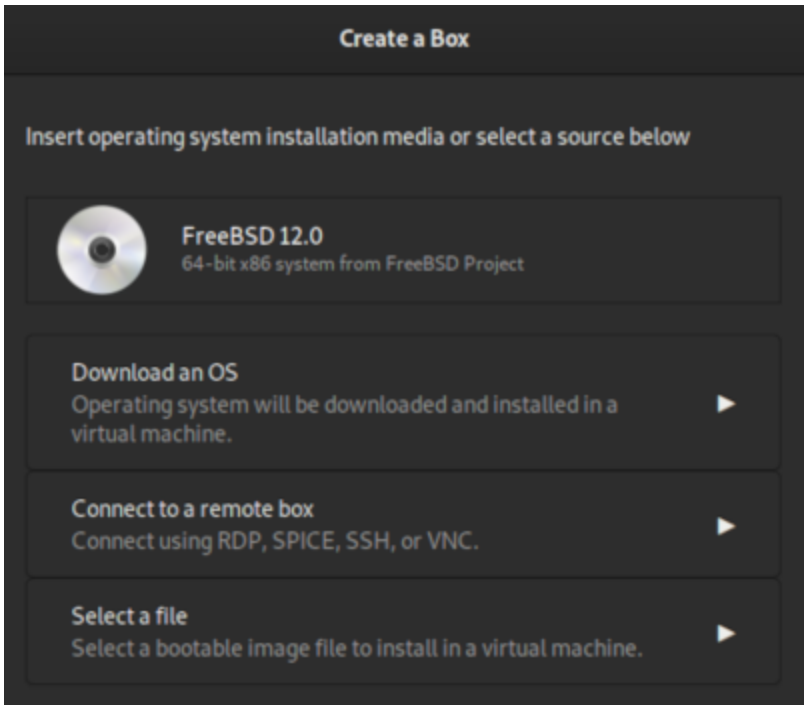
In this chapter, I cover the remote access capabilities of Boxes. As in the previous chapters, I'm using "box" to refer to a virtual machine.

Connect to a remote box

When you open Boxes, it displays the main window.

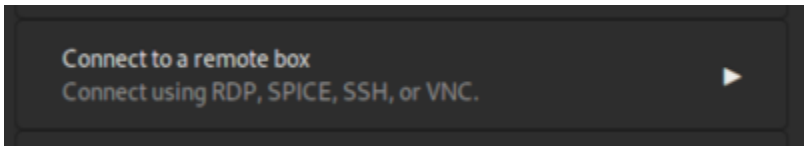


When you click the **New** button in the top-left corner, the **Create a Box** dialog will open.

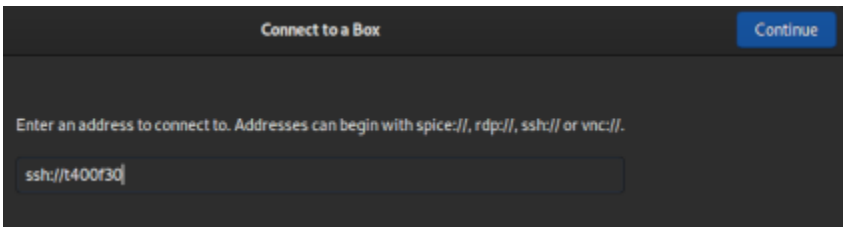


Connect with SSH

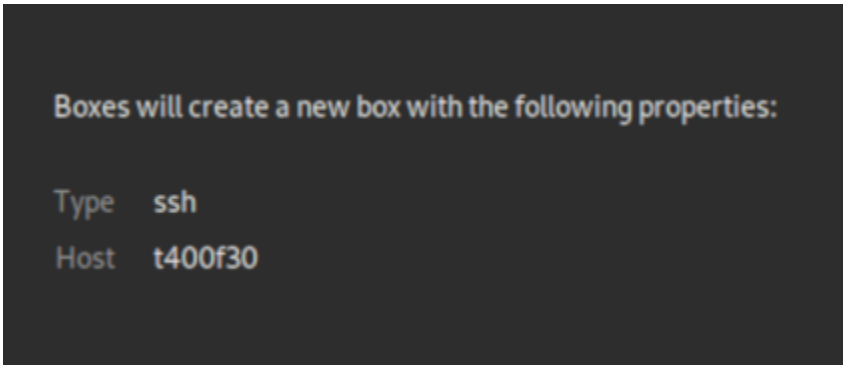
To connect to a remote system using the Secure Shell (SSH) protocol, click **Connect to a remote box**.



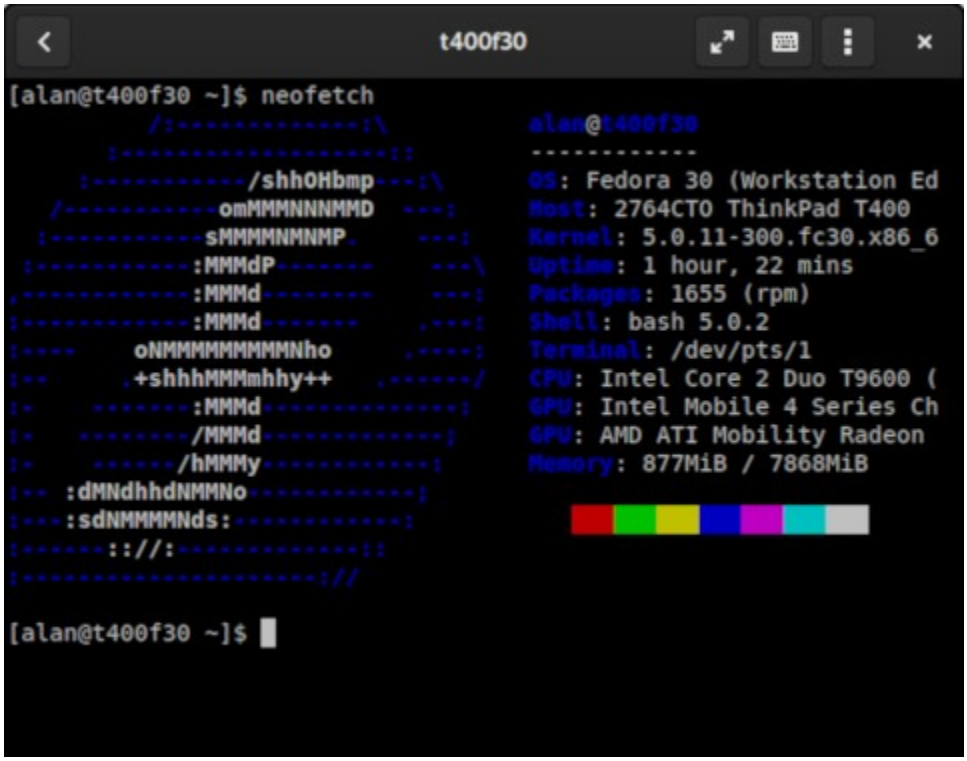
Enter the SSH address, for example, **ssh://t400f30**, and then click **Continue**.



Click **Create** on the **Review** screen.

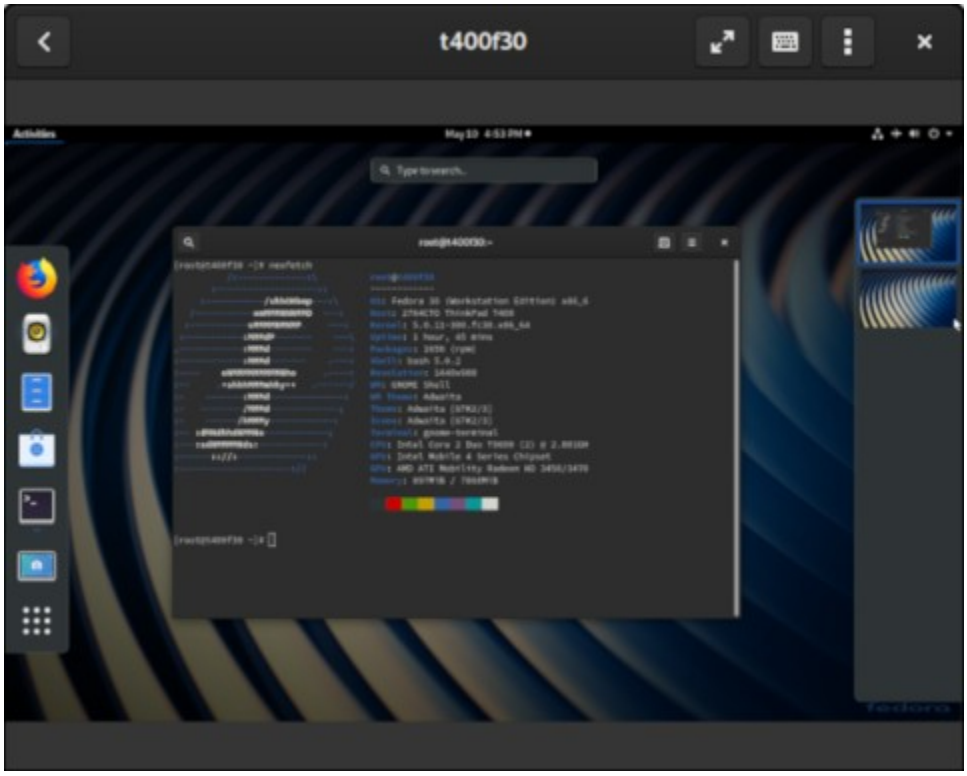


Boxes connects to the remote system via SSH, and you can log in as usual. The name of the box derives from the hostname. I recommend changing it in the Properties to something like **SSH to t400f30** to be more clear in case you have multiple connections to the same remote system.



Connect with VNC

To connect with Virtual Network Computing (VNC), enter the address, such as **vnc://t400f30**. VNC provides a graphical view of a remote system's desktop.

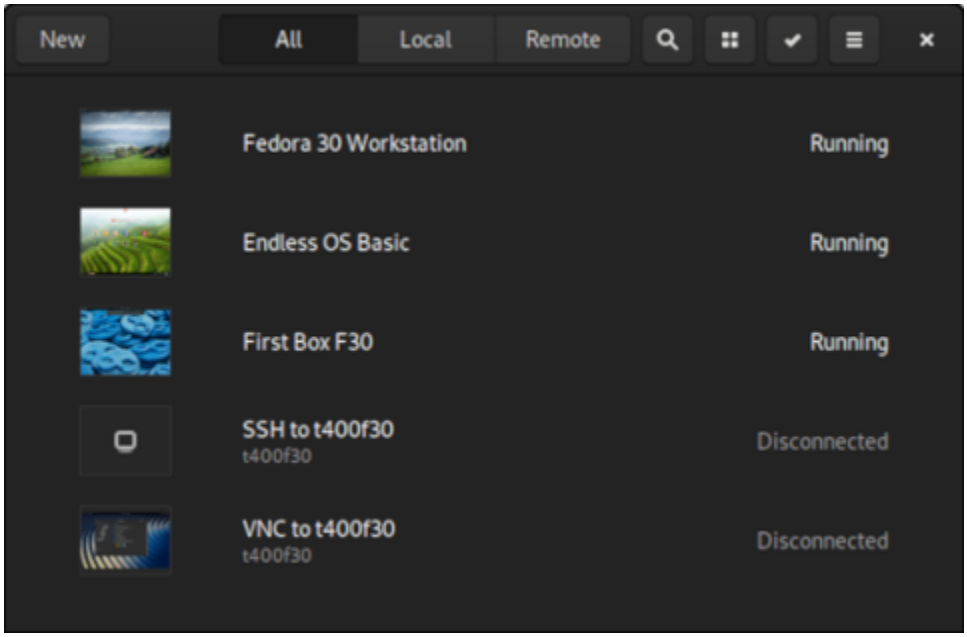


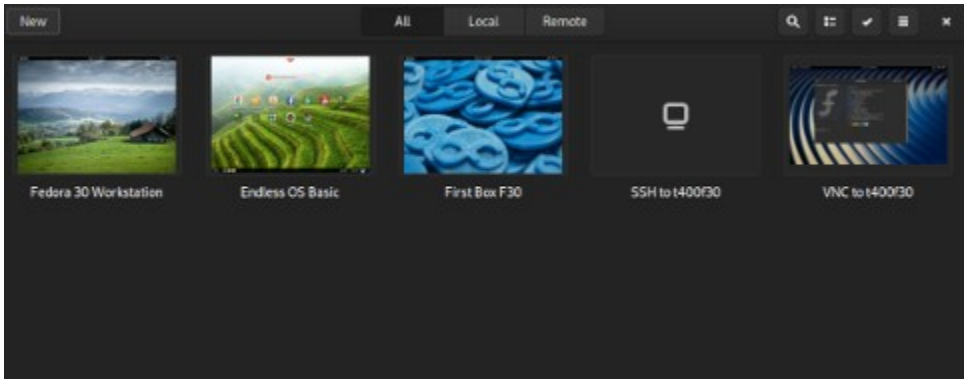
Connect with RDP and SPICE

Boxes also supports the Remote Desktop Protocol (RDP) and SPICE protocol; they are used in the same way as VNC and SSH, in that you provide the address of the system to which you want to connect.

Your boxes

As you create boxes and connect to remote systems, the main window will begin to fill and provide a centralized store for all of your boxes. You can also choose to view only your local boxes or only remote boxes. This screen can be configured to view them in two ways: either in a list or as large icons.





GNOME Boxes can expedite the deployment of virtual machines, or boxes, on your local Linux system. It's nice to have a combined view of local and remote boxes.

Boxes is possibly the simplest virtualization tool and it's great for people who don't have a lot of time or need for complex configurations.

How I use Vagrant with libvirt

By Seth Kenlon

I'm a fan of Linux not only in the professional sense, but also just for fun. While I've used Slackware on workstations and Red Hat Enterprise Linux (RHEL) on servers for years, I love seeing how other distributions do things. What's more, I really like to test applications and scripts I write on other distributions to ensure portability. In fact, that's one of the great advantages of Linux, as I see it: You can download a distro and test your software on it for free. You can't do that with a closed OS, at least not without either breaking an EULA or paying to play, and even then, you're often signing up to download several gigabytes just to test an application that's no more than a few hundred megabytes. But Linux is open source, so there's rarely an excuse to ignore at least the three or four main distros, except that setting up a virtual machine can take a lot of clicks and sometimes complex virtual networking. At least, that used to be the excuse until Vagrant changed the virtual machine workflow for developers.

What is Vagrant

Vagrant is a simple virtual machine manager for your terminal. It allows you to easily pull a minimal and pre-built virtual machine from the Internet, run it locally, and SSH into it in just a few steps. It's the quickest you'll ever set up a virtual machine. It's ideal for web developers needing a test web server, programmers who need to test an application across distributions, and hobbyists who enjoy seeing how different distributions work.

Vagrant itself is relatively minimal, too. It's not a virtualization framework itself. It only manages your virtual machines ("boxes" in Vagrant terminology). It can use VirtualBox or, through a plug-in, the lightweight libvirt project as a backend.

What is libvirt

The libvirt project is a toolkit designed to manage virtualization, with support for KVM, QEMU, LXC, and more. You might think of it as a sort of virtual machine API, allowing developers to write friendly applications that make it easy for users to orchestrate virtualization through libvirt. I use libvirt as the backend for Vagrant because it's useful across several applications, including virt-manager and GNOME Boxes.

Installing Vagrant

You can install Vagrant from vagrantup.com/downloads. There are builds available for Debian-based systems, CentOS-based systems, macOS, Windows, and more.

For CentOS, Fedora, or similar, you get an RPM package, which you can install with `dnf`:

```
$ sudo dnf install ./vagrant_X.Y.ZZ_x86_64.rpm
```

On Debian, Linux Mint, Elementary, and similar, you get a DEB package, which you can install with `apt`:

```
$ sudo apt install ./vagrant_X.Y.ZZ_x86_64.deb
```

Installing libvirt and support packages

On Linux, your distribution may already have libvirt installed, but to enable integration with Vagrant you need a few other packages, too. Install these with your package manager.

On Fedora, CentOS, and similar:

```
$ sudo dnf install gcc libvirt \
libvirt-devel libxml2-devel \
make ruby-devel libguestfs-tools
```

On Debian, Linux Mint, and similar:

```
$ sudo apt install build-dep vagrant ruby-libvirt \
qemu libvirt-daemon-system libvirt-clients ebtables \
dnsmasq-base libxslt-dev libxml2-dev libvirt-dev \
zlib1g-dev ruby-dev libguestfs-tools
```

Depending on your distribution, you may have to start the `libvirt` daemon (some distributions start it automatically after install, but it doesn't hurt to try to start it if you're not sure):

```
$ sudo systemctl start libvirtd
```

Installing the Vagrant-libvirt plugin

In Vagrant, libvirt is enabled through a plug-in. Vagrant makes it easy to install a plug-in, so your first Vagrant command is one you'll rarely run again:

```
$ vagrant plugin install vagrant-libvirt
```

Now that the libvirt plug-in is installed, you can start using virtual machines.

Setting up your Vagrant environment

To start with Vagrant, create a directory called `~/Vagrant`. This is where your `Vagrantfiles` are stored.

```
$ mkdir ~/Vagrant
```

In this directory, create a subdirectory to represent a distro you want to download. For instance, assume you need a CentOS test box.

Create a CentOS directory, and then change to it:

```
$ mkdir ~/Vagrant/centos
$ cd ~/Vagrant/centos
```

Now you need to find a virtual machine so you can convert the directory you've just made into a Vagrant environment.

Finding a Vagrant virtual machine

Broadly speaking, Vagrant boxes come from three different places: Hashicorp (the maintainers of Vagrant), maintainers of distributions, and people like you and me. Some images are minimal, intended to serve as a base for customization. In contrast, others try to solve a specific need (for instance, you might find a LAMP stack image ready for web

development). You can find images by browsing or searching the main hub for boxes app.vagrantup.com/boxes/search.

For this example, search for "centos" and find the entry named `generic/centos8`. Click on the image for instructions on how to use the virtual machine. The instructions come in two varieties:

- The code you need for a Vagrantfile
- The command you need to use the box from a terminal

The latter is the more straightforward method:

```
$ vagrant init generic/centos8
```

The `init` subcommand creates a configuration file, called a Vagrantfile, in your current directory, which transforms that directory into a Vagrant environment. At any time, you can view a list of known Vagrant environments using the `global-status` subcommand:

```
$ vagrant global-status
id       name    provider state   directory
-----------------------------------------
49c797f  default libvirt running /home/tux/Vagrant/centos8
```

Starting a virtual machine with Vagrant

Once you've run the `init` command, you can start your virtual machine with `vagrant up`:

```
$ vagrant up
```

This causes Vagrant to download the virtual machine image if it doesn't already exist locally, set up a virtual network, and configure your box.

Entering a Vagrant virtual machine

Once your virtual machine is up and running, you can log in to it with `vagrant ssh`:

```
$ vagrant ssh
box$
```

You connect to the box running in your current Vagrant environment. Once logged in, you can run all the commands native to that host. It's a virtual machine running its own kernel, with emulated hardware and common Linux software.

Leaving a Vagrant virtual machine

To leave your Vagrant virtual machine, log out of the host as you normally exit a Linux computer:

```
box$ exit
```

Alternately, you can power the virtual machine down:

```
box$ sudo poweroff
```

You can also stop the machine from running using the `vagrant` command:

```
box$ vagrant halt
```

Destroying a Vagrant virtual machine

When finished with a Vagrant virtual machine, you can destroy it:

```
$ vagrant destroy
```

Alternately, you can remove a virtual machine using the global `box` subcommand:

```
$ vagrant box remove generic/centos8
```

Vagrant is easy

Vagrant makes virtual machines trivial, disposable, and fast. When you need a test environment or a fake server to ping or develop on, or a clean lab computer for experimentation or monitoring, you can get one with Vagrant. Some people think virtual machines aren't relevant now that containers have taken over servers, but virtual machines have unique traits that make them useful. They run their own kernel, have a full and unique stack separate from the host machine, and use emulated hardware. When a virtual machine is what you need, Vagrant may be just the best way to get it.

A beginner's guide to using Vagrant

By Jessica Cherry

Vagrant describes itself as "a tool for building and managing virtual machine environments in a single workflow. With an easy-to-use workflow and focus on automation, Vagrant lowers development environment setup time, increases production parity, and makes the 'works on my machine' excuse a relic of the past."

Vagrant works with a standard format for documenting an environment, called a Vagrantfile. According to Vagrant's website:

> The primary function of the Vagrantfile is to describe the type of machine required for a project, and how to configure and provision these machines. Vagrantfiles are called Vagrantfiles because the actual literal filename for the file is **Vagrantfile** (casing does not matter unless your file system is running in a strict case sensitive mode).

Vagrant is essentially a wrapper to allow for repeatable virtual machine management, but it does not run VMs itself. This tutorial will use VirtualBox as that environment manager, though Hyper-V and Docker also work by default. Check out Vagrant's documentation to learn how to use a different provider for this tutorial.

Build a Vagrantfile

This tutorial works through an example application for a simple Hello World page inside a Ruby on Rails (Rails for short) web app. Before you begin, install the following (if you haven't already):

- Vagrant
- VirtualBox

- Ruby on Rails
- An editing environment, like Atom or Notepad++

If you're on Fedora and prefer using the command line, there is an excellent Fedora tutorial, and there's a similarly helpful tutorial for Windows using Chocolatey. After everything is installed, open your terminal and create a new directory to work in. You can put your directory wherever you like; I prefer to use a folder under my user account:

```
$ mkdir -p ~/Development/Rails_app
$ cd ~/Development/Rails_app
$ vagrant init

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.
```

This creates a Vagrantfile with the default configuration information written in Ruby syntax. Look at line 15:

```
config.vm.box = "base"
```

This indicates that Vagrant will use a default operating system image it hosts called **base**, which you don't have yet. Confirm that by running **box list**:

```
$ vagrant box list
There are no installed boxes! Use `vagrant box add` to add some.
```

Should you try to start your environment using the **up** command, it fails because Vagrant expects an OS called **base** to exist locally. Switch to the most commonly used environment, **bento/ubuntu-16.04**, then try to spin up your environment. Change the **config.vm.box** line in your Vagrantfile to:

```
config.vm.box = "centos/7"
```

And now you can run the most satisfying command in virtual machine history:

```
$ vagrant up
Bringing machine 'default' up with 'libvirt' provider...
==> default: Box 'centos/7' could not be found. Attempting to find and install...
    default: Box Provider: libvirt
    default: Box Version: >= 0
```

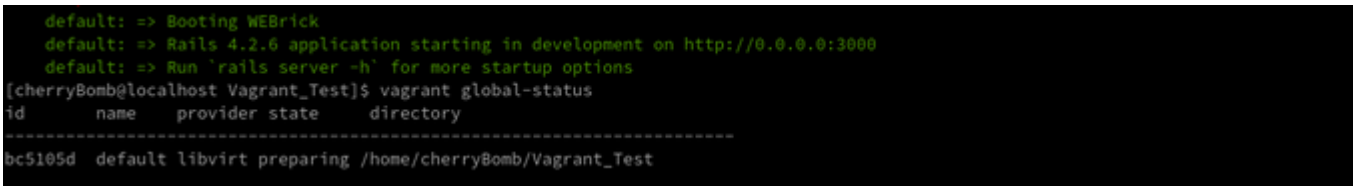
```
==> default: Loading metadata for box 'centos/7'
    default: URL: https://vagrantcloud.com/centos/7
==> default: Adding box 'centos/7' (v1905.1) for provider: libvirt
    default: Downloading:
https://vagrantcloud.com/centos/boxes/7/versions/1905.1/providers/libvirt.box
    default: Download redirected to host: cloud.centos.org
...
```

Here is why this is so nice. This tutorial sets up a small website, but if you had a larger website and needed to check whether the frontend looks right, your playbook file and copy-over files would allow you to see your changes. If you have small applications you want to test quickly—without doing an entire Docker image build or logging into a server—this local testing is good for quick checks and repairs. If you're working within hardware, this will make it easy to see if the application will work within your operating system, and it allows you to know which dependencies you need. In the end, it makes for easier deployment and faster testing than doing a from-scratch continuous integration and deployment (CI/CD) to a test server, and it provides quicker access and more control.

The reason this is so cool can be explained in one simple sentence: You now have local automation. It also allows you to gather a larger breadth of knowledge behind Ansible and headless server deployments.

Verify Vagrant worked correctly

One way to determine whether this finished properly is seeing a bunch of green text and the words **rails server -h** for startup options. This means the web app has started and is running.



But you want to use **vagrant global-status** as well as **vagrant status**.

```
[cherryBomb@localhost Vagrant_Test]$ vagrant global-status
id       name    provider state   directory
--------------------------------------------------------------------
bc5105d  default libvirt running /home/cherryBomb/Vagrant_Test

The above shows information about all known Vagrant environments
on this machine. This data is cached and may not be completely
up-to-date (use "vagrant global-status --prune" to prune invalid
entries). To interact with any of the machines, you can go to that
directory and run Vagrant, or you can use the ID directly with
Vagrant commands from any directory. For example:
"vagrant destroy 1a2b3c4d"
[cherryBomb@localhost Vagrant_Test]$ vagrant status
Current machine states:

default                   running (libvirt)

The Libvirt domain is running. To stop this machine, you can run
`vagrant halt`. To destroy the machine, you can run `vagrant destroy`.
[cherryBomb@localhost Vagrant_Test]$
```

The **vagrant status** command checks the machine states that originate in the current directory. So, if you have a VM up and running, it will show as up and running. If it is broken in any way, it will display a message with an error and some logs when you run **vagrant up**. If some machines are down, they will also show as not running or shut down.

However, the **vagrant global-status** command can give the status of multiple environments created in Vagrant. So, if you split the environments for different VM types or storage types, this command gives you an option to see everything in all the environments you've created.

Customize the Vagrant configuration

The machine settings have multiple config.vm options. This tutorial will use the networking option to allow port forwarding. Port forwarding allows you to access a network port in our virtual environment as if it was a local port via a special local network. This means traffic is allowed to see the one thing you allow on this server; in this case, it's a tiny frontend webpage.

The main reason this matters is for security. Limiting traffic can prevent bad actors and traffic overflow. The way this is built, you can't log into this server unless you configure it as such. This also means no one else can SSH in or see anything except the one little frontend webpage.

Before moving on, remove the VM so you can start over by running **vagrant destroy**:

```
$ vagrant destroy
    default: Are you sure you want to destroy the 'default' VM? [y/N] y
==> default: Removing domain...
```

To include port forwarding, add this in the next config line:

```
 Vagrant.configure("2") do |config|
  config.vm.box = "bento/ubuntu-16.04"
  config.vm.network "forwarded_port", guest: 3000, host: 9090
  end
```

Save the file and run:

```
vagrant up
```

You now have a VM that forwards port **3000** out to the open world as **9090**. You should now be able to go to **127.0.0.1:9090** on your web browser and see nothing but a plain white page.

Run **vagrant destroy** again to remove the VM so you can start over.

Provision Vagrant with Ansible and scripts

While base boxes offer a good starting point, it's common to customize a VM during the provisioning process, and you can use multiple provisioning tactics. To follow along, download the playbook and script.

This example uses Ansible to set up a basic install of the Ruby on Rails web framework. Then, it adds an extra shell script to configure the web app's welcome page to say: *Hello World, Sorry for the Delay*. (The purpose of this message is because this build takes a long time and people may become frustrated by the delay.)

The following Vagrantfile reflects Ansible and a playbook running locally on my machine, so it will differ from yours. You can read about using Ansible with Vagrant in Vagrant's docs.

```
Vagrant.configure("2") do |config|
  config.vm.box = "bento/ubuntu-16.04"
  config.vm.network "forwarded_port", guest: 3000, host: 9090
  ####### Provision #######
  config.vm.provision "ansible_local" do |ansible|
    ansible.playbook = "prov/playbook.yml"
    ansible.verbose = true
```

```
  config.vm.provision "shell", path: "script.sh"
  end
end
```

After saving the file, run my favorite command:

```
vagrant up
```

You now have a VM up and running with Rails, and when you enter **127.0.0.1:9090** in your web browser, you see a webpage that says: *Hello World, Sorry for the Delay*.

Now that you have all this background, you can try to build your own script.

Final notes

Vagrant is fairly easy to work with and has abundant documentation to help you along the way. It's is a great tool if you're looking to work with code in a small staging or development environment; any destruction is a non-issue because the environment itself is disposable.

Want to give it a try? Take a look at my repo.

Use Vagrant to test your scripts on different operating systems

By Ayush Sharma

I've been happy using Vagrant for quite a while now. I work with several DevOps tools, and installing them all on one system can get complicated. Vagrant lets you do cool things without breaking your system because you don't have to experiment on your production system at all.

If you're familiar with VirtualBox or GNOME Boxes, then learning Vagrant is easy. Vagrant is a simple and clean interface for working with virtual machines. A single config file, called `Vagrantfile`, allows you to customize your virtual machines (called *Vagrant boxes*). A simple command-line interface lets you start, stop, suspend, or destroy your boxes.

Consider this simple example.

Let's say you want to write Ansible or shell scripts to install Nginx on a new server. You can't do it on your own system because you might not be running the operating system you want to test, or you may not have all of the dependencies for what you want to do. Launching new cloud servers for testing can be time-consuming and expensive. This is where Vagrant comes in. You can use it to bring up a virtual machine, provision it using your scripts, and prove that everything works as expected. You can then delete the box, re-provision it, and re-run your scripts to verify it. You can repeat this process as many times as you want until you're confident your scripts work under all conditions. And you can commit your Vagrantfile to Git to ensure your team is testing the exact same environment (because they'll be using the exact same test box). No more "...but it works fine on my machine!"

Getting started

First, install Vagrant on your system and then create a new folder to experiment in. In this new folder, create a new file named `Vagrantfile` with these contents:

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/hirsute64"
  end
```

You can also run `vagrant init ubuntu/hirsute64`, and it will generate a new Vagrantfile for you. Now run `vagrant up`. This command will download the `ubuntu/hirsuite64` image from the Vagrant registry.

```
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'ubuntu/hirsute64'...
==> default: Matching MAC address for NAT networking...
==> default: Checking if box 'ubuntu/hirsute64' version '20210820.0.0' is up to
date...
==> default: Setting the name of the VM: a_default_1630204214778_76885
==> default: Clearing any previously set network interfaces...
==> default: Preparing network interfaces based on configuration...
    default: Adapter 1: nat
    default: Adapter 2: hostonly
==> default: Forwarding ports...
    default: 22 (guest) => 2222 (host) (adapter 1)
==> default: Running 'pre-boot' VM customizations...
==> default: Booting VM...
==> default: Waiting for machine to boot. This may take a few minutes...
    default: SSH address: 127.0.0.1:2222
    default: SSH username: vagrant
    default: SSH auth method: private key
    default: Warning: Remote connection disconnect. Retrying...
    default: Warning: Connection reset. Retrying...
    default:
    default: Vagrant insecure key detected. Vagrant will automatically replace
    default: this with a newly generated keypair for better security.
    default:
    default: Inserting generated public key within guest...
    default: Removing insecure key from the guest if it's present...
    default: Key inserted! Disconnecting and reconnecting using new SSH key...
==> default: Machine booted and ready!
```

At this point, if you open your Vagrant backend (such as VirtualBox or virt-manager), you'll see your box there. Next, run `vagrant ssh` to log in to the box. If you can see the Vagrant prompt, then you're in!

```
~ vagrant ssh
Welcome to Ubuntu 21.04 (GNU/Linux 5.11.0-31-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage
```

```
  System information as of Sun Aug 29 02:33:51 UTC 2021

  System load:  0.01              Processes:               110
  Usage of /:   4.1% of 38.71GB   Users logged in:         0
  Memory usage: 17%               IPv4 address for enp0s3: 10.0.2.15
  Swap usage:   0%                IPv4 address for enp0s8: 192.168.1.20


0 updates can be applied immediately.


vagrant@ubuntu-hirsute:~$
```

Vagrant uses "base boxes" to bring up your local machines. In our case, Vagrant downloads the `ubuntu/hirsuite64` image from Hashicorp's Vagrant catalogue and plugs into VirtualBox to create the actual box.

Shared folders

Vagrant maps your current folder as `/vagrant` within the Vagrant box. This allows you to keep your files in sync on your system and within the box. This is great for testing a Nginx website by pointing your document root to `/vagrant`. You can use an IDE to make changes and Nginx within the box will serve them.

Vagrant commands

There are several Vagrant commands which you can use to control your box.

Some of the important ones are:

- `vagrant up`: Bring a box online.
- `vagrant status`: Show current box status.
- `vagrant suspend`: Pause the current box.
- `vagrant resume`: Resume the current box.
- `vagrant halt`: Shutdown the current box.
- `vagrant destroy`: Destroy the current box. By running this command, you will lose any data stored on the box.
- `vagrant snapshot`: Take a snapshot of the current box.

Try Vagrant

Vagrant is a time-tested tool for virtual machine management using DevOps principles. Configure your test machines, share the configs with your team, and test your projects in a predictable and reproducible environment. If you're developing software, then you'll do your users a great service by using Vagrant for testing. If you're not developing software but you love to try out new versions of an OS, then there's no easier way. Try Vagrant today!

Managing virtual environments with Vagrant

By Alex Juarez

Vagrant is a tool that offers a simple and easy to use command-line client for managing virtual environments. I started using it because it made it easier for me to develop websites, test solutions, and learn new things.

In this getting-started guide, I demonstrate how to use Vagrant to:

1. Create and configure a VirtualBox virtual machine (VM)
2. Run post-deployment configuration shell scripts and applications

Sounds simple, and it is. Vagrant's power comes from having a consistent workflow for deploying and configuring machines regardless of platform or operating system.

We'll start by using VirtualBox as a **provider**, setting up an Ubuntu 16.04 **box**, and applying a few shell commands as the **provisioner**. I'll refer to the physical machine (e.g., a laptop or desktop) as the host machine and the Vagrant VM as the guest.

In this tutorial, you'll put together a Vagrantfile and offer periodic checkpoints to make sure our files look the same. The introductory topics include:

- Installing Vagrant
- Choosing a Vagrant box
- Understanding the Vagrantfile
- Getting the VM running
- Using provisioners

Advanced topics:

- Networking
- Syncing folders

- Deploying multiple machines
- Making sure everything works

It looks like a lot, but it will all fit together nicely once we are finished.

Installing Vagrant

First, download Vagrant and VirtualBox and install the latest versions of each.

We can enter the following commands to ensure the latest versions of the applications are installed and ready to use.

Vagrant:

```
$ vagrant --version
Vagrant 2.0.3
```

VirtualBox:

```
$ VBoxManage --version
5.2.8r121009
```

Choosing a Vagrant box

Picking a Vagrant box is similar to picking an image for a server. At the base level, we choose which operating system (OS) we want to use. Some boxes go further and will have additional software (such as the Puppet or Chef client) already installed.

The go-to online repository for boxes is Vagrant Cloud; it offers a cornucopia of Vagrant boxes for multiple providers. In this tutorial, we'll be using Ubuntu Xenial Xerus 16.04 LTS daily build.

Understanding the Vagrantfile

Think of the Vagrantfile as the configuration file for an environment. It describes the Vagrant environment with regard to how to build and configure the VirtualBox VMs.

We need to create an empty project directory to work from, then initialize a Vagrant environment from that directory with this command:

```
$ vagrant init ubuntu/xenial64
```

This only creates the Vagrantfile; it doesn't bring up the Vagrant box.

The Vagrantfile is well-documented with a lot of guidance on how to use it. We can generate a minimized Vagrantfile with the `--minimal` flag.

```
$ vagrant init --minimal ubuntu/xenial64
```

The resulting file will look like this:

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

We will talk more about the Vagrantfile later, but for now, let's get this box up and running.

Getting the VM running

Let's issue the following command from our project directory:

```
$ vagrant up
```

It takes a bit of time to execute `vagrant up` the first time because it downloads the box to your machine. It is much faster on subsequent runs because it reuses the same downloaded box.

Once the VM is up and running, we can `ssh` into our single machine by issuing the following command in our project directory:

```
$ vagrant ssh
```

That's it! From here we should be able to log onto our VM and start working with it.

Using provisioners

Before we move on, let's review a bit. So far, we've picked an image and gotten the server running. For the most part, the server is unconfigured and doesn't have any of the software we might want.

Provisioners provide a way to use tools such as Ansible, Puppet, Chef, and even shell scripts to configure a server after deployment.

An example of using the shell provisioner can be found in a default Vagrantfile. In this example, we'll run the commands to update apt and install Apache2 to the server.

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

If we want to use an Ansible playbook, the configuration section would look like this:

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

A neat thing is we can run only the provisioning part of the Vagrantfile by issuing the `provision` subcommand. This is great for testing out scripts or configuration management plays without having to re-build the VM each time.

Vagrantfile checkpoint

Our minimal Vagrantfile should look like this:

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/xenial64"
   config.vm.provision "shell", inline: <<-SHELL
     apt-get update
     apt-get install -y apache2
   SHELL
end
```

After adding the provisioning section, we need to run this provisioning subcommand:

```
$ vagrant provision
```

Next, we'll continue to build on our Vagrantfile, touching on some more advanced topics to build a foundation for anyone who wants to dig in further.

Networking

In this section, we'll add an additional IP address on VirtualBox's `vboxnet0` network. This will allow us to access the machine via the `192.168.33.0/24` network.

Adding the following line to the Vagrantfile will configure the machine to have an additional IP on the `192.168.33.0/24` network. This line is also used as an example in the default Vagrantfile.

```
config.vm.network "private_network", ip: "192.168.33.10
```

Vagrantfile checkpoint

For those following along, here where our working Vagrantfile stands:

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/xenial64"
   config.vm.network "private_network", ip: "192.168.33.10"
   config.vm.provision "shell", inline: <<-SHELL
     apt-get update
     apt-get install -y apache2
   SHELL
end
```

Next, we need to reload our configuration to reconfigure our machine with this new interface and IP. This command will shut down the VM, reconfigure the Virtual Box VM with the new IP address, and bring the VM back up.

```
$ vagrant reload
```

When it comes back up, our machine should have two IP addresses.

Syncing folders

Synced folders are what got me into using Vagrant. They allowed me to work on my host machine, using my tools, and at the same time have the files available to the web server or application. It made my workflow much easier.

By default, the project directory on the host machine is mounted to the guest machine as `/home/vagrant`. This worked for me in the beginning, but eventually, I wanted to customize where this directory was mounted.

In our example, we are defining that the HTML directory within our project directory should be mounted as `/var/www/html` with user/group ownership of `root`.

```
config.vm.synced_folder "./html", "/var/www/html",
 owner: "root", group: "root"
```

One thing to note: If you are using a synced folder as a web server document root, you will need to disable `sendfile`, or you might run into an issue where it looks like the files are not updating.

Updating your web server's configuration is out of scope for this article, but here are the directives you will want to update.

In Apache:

```
EnableSendFile Off
```

In Nginx:

```
sendfile off;
```

Vagrantfile checkpoint

After adding our synced folder configuration, our Vagrantfile will look like this:

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/xenial64"
   config.vm.network "private_network", ip: "192.168.33.10"
   config.vm.synced_folder "./html, "/var/www/html",
      owner: "root", group: "root"
   config.vm.provision "shell", inline: <<-SHELL
     apt-get update
     apt-get install -y apache2
   SHELL
end
```

We need to reload our machine to make the new configuration active.

```
$ vagrant reload
```

Deploying multiple machines

We sometimes refer to the project directory as an "environment," and one machine is not much of an environment. This last section extends our Vagrantfile to deploy two machines.

To create two machines, we need to enclose the definition of a single machine inside a `vm.define` block. The rest of the configuration is exactly the same.

Here is an example of a server definition within a `define` block.

```
Vagrant.configure("2") do |config|

config.vm.define "web" do |web|
    web.vm.box = "web"
    web.vm.box = "ubuntu/xenial64"
     web.vm.network "private_network", ip: "192.168.33.10"
     web.vm.synced_folder "./html", "/var/www/html",
        owner: "root", group: "root"
     web.vm.provision "shell", inline: <<-SHELL
       apt-get update
       apt-get install -y apache2
     SHELL
  end

end
```

Notice in the `define` block, our variable is called `"web"` and it is carried through the block to reference each configuration method. We'll use the same name to access it later.

In this next example, we'll add a second machine called `"db"` to our configuration. Where we used `"web"` in our second block before, we'll use `"db"` to reference the second machine. We'll also update our IP address on the `private_network` so we can communicate between the machines.

```
Vagrant.configure("2") do |config|

config.vm.define "web" do |web|
    web.vm.box = "web"
    web.vm.box = "ubuntu/xenial64"
     web.vm.network "private_network", ip: "192.168.33.10"
     web.vm.synced_folder "./html", "/var/www/html",
        owner: "root", group: "root"
     web.vm.provision "shell", inline: <<-SHELL
       apt-get update
       apt-get install -y apache2
     SHELL
  end

  config.vm.define "db" do |db|
      db.vm.box = "db"
      db.vm.box = "ubuntu/xenial64"
       db.vm.network "private_network", ip: "192.168.33.20"
       db.vm.synced_folder "./html", "/var/www/html",
          owner: "root", group: "root"
       db.vm.provision "shell", inline: <<-SHELL
         apt-get update
```

```
        apt-get install -y apache2
      SHELL
    end

end
```

Completed Vagrantfile checkpoint

In our final Vagrantfile, we'll install the MySQL server, update the IP address, and remove the configuration for the synced folder from the second machine.

```
Vagrant.configure("2") do |config|

config.vm.define "web" do |web|
    web.vm.box = "web"
    web.vm.box = "ubuntu/xenial64"
     web.vm.network "private_network", ip: "192.168.33.10"
     web.vm.synced_folder "./html", "/var/www/html",
        owner: "root", group: "root"
     web.vm.provision "shell", inline: <<-SHELL
       apt-get update
       apt-get install -y apache2
     SHELL
  end

  config.vm.define "db" do |db|
      db.vm.box = "db"
      db.vm.box = "ubuntu/xenial64"
       db.vm.network "private_network", ip: "192.168.33.20"
       db.vm.provision "shell", inline: <<-SHELL
         export DEBIAN_FRONTEND="noninteractive"
         apt-get update
         apt-get install -y mysql-server
       SHELL
    end

end
```

Making sure everything works

Now we have a completed Vagrantfile. Let's introduce one more Vagrant command to make sure everything works.

Let's destroy our machine and build it brand new.

The following command will remove our previous Vagrant image but keep the box we downloaded earlier.

```
$ vagrant destroy --force
```

Now we need to bring the environment back up.

```
$ vagrant up
```

We can ssh into the machines using the `vagrant ssh` command:

```
$ vagrant ssh web
```

or

```
$ vagrant ssh db
```

You should have a working Vagrantfile you can expand upon and serve as a base for learning more. Vagrant is a powerful tool for testing, developing and learning new things. I encourage you to keep adding to it and exploring the options it offers.