Vagrant Virtual Development Environment Cookbook

Vagrant allows you to use virtualization and cloud technologies to power faster, efficient, and sharable development environments. It duplicates the development environment to allow users to easily share and combine data on different machines and also takes care of security concerns.

Each recipe of Vagrant Virtual Development Environment Cookbook provides practical information on using Vagrant to solve specific problems and additional resources to help you learn more about the techniques demonstrated.

With recipes ranging from getting new users acquainted with Vagrant, to setting up multimachine environments, you will be able to develop common project types and solutions with the help of this practical guide.

What this book will do for you...

- Define single and multiple virtual machine Vagrant environments
- Provision Vagrant environments in a consistent and repeatable manner with various configuration management tools
- Control powerful cloud resources from a desktop development environment
- Use Vagrant to publish and share development environments
- Start and expand your Vagrant environment with community resources
- Share resources on a development machine with a virtual Vagrant environment

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions for solving the problem efficiently
- Clear explanations of what you did
- Apply the solution to other situations

Quick answers to common problems

Vagrant Virtual Development Environment Cookbook

Over 35 hands-on recipes to help you master Vagrant, and create and manage virtual computational environments

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 5 "Networked Vagrant Environments"
- A synopsis of the book’s content
- More information on Vagrant Virtual Development Environment Cookbook

About the Author

Chad Thompson is a software developer, architect, and DevOps specialist in Central Iowa. He has 15 years of experience in creating and deploying applications for the Web. Chad began using Vagrant 3 years ago when he was trying to solve a tough problem in legacy application development. Since then, he has made use of Vagrant and configuration management tools to support the development and deployment of several web applications in data centers and cloud platforms. He holds certifications in Puppet and Oracle technologies and has enjoyed the pleasure of speaking before several technical conferences and camps. Chad holds two degrees in physics and can be found playing low brass instruments in ensembles around the state of Iowa.

Chad has written articles for O’Reilly web publications and the IOUG SELECT Journal (where he briefly worked as an executive editor). Recently, he reviewed the book Creating Development Environments with Vagrant for Packt Publishing, and recorded a set of video presentations titled Learning Git by Infinite Skills.

I owe a great measure of gratitude to many people for helping me with the production of this book. I would like to thank my colleagues at Dice Holdings Inc. for their support and feedback during the development of the book. I would like to thank Zach Arlen of FullContact in Denver, CO, for introducing me to Vagrant as a solution to a problem years ago. Mostly, I would like to thank my family for their continued love and support.

With the publication of this book, I would also like to offer my gratitude to Dr. Robert Merlino and the late Dr. Nicola D'Angelo of the University of Iowa. They both taught me a great deal about formulating ideas and teaching others, which I hope serves the readers of this book.
Vagrant Virtual Development Environment Cookbook

If you have written software on a desktop computer and attempted to deploy your code to another computer (a server), you have already encountered the challenges presented when deploying software. Developers and administrators frequently struggle with errors and defects, when development environments are different from the eventual production machines. There can be a number of differences introduced when the environments are different at the operating system level. Development with desktop operating systems (such as Windows or OS X) can introduce many issues when deploying to production environments that run a Unix (or Linux) environment.

The introduction of desktop hypervisor software allowed developers to develop and test software using virtual machines. A virtual machine is essentially a system within a system, wherein developers working on a desktop operating system can develop and deploy with a copy of the operating system and environment that closely mimics the eventual production environment. When desktop hypervisors became available, development teams found that they could share development environments by sharing the files used by the hypervisors to store the state of virtual machines. In many cases, sharing a virtual machine involved passing around copies of files on a portable hard drive or a shared network folder.

A few years ago, I encountered this specific example when working on a project that involved adding new features to software that ran on an environment, which we could not support with our modern desktop hardware. As many projects reveal, technical debt was introduced to the application by using some very specific features of the Java Development Kit (version 1.5), an environment that was impossible to work on with a 64-bit OS X machine. This machine had dual problems of being a 64-bit machine and it also lacked native support for Java 1.5 XML libraries. The solution to this problem was the creation of a single virtual machine that was shared between developers, passing around a copy of the machine created by a team lead and using it locally to compile and test our modifications.

As time passed by, changes to the environment became an issue, as we began struggling with the differences between not only the development and production environments, but also between our individual development environments as changes were made, making sure that each developer was working on the latest version of the virtual machine on that portable hard drive, which soon had a few different versions itself.
Eventually, the problem of maintaining development environments was large enough to begin looking for new solutions. Configuration management approaches helped us to start defining our environment in code, but we still had issues with sharing and maintaining our base environment. We found immediate use of an open source project called Vagrant, which was gaining some traction.

Vagrant (http://vagrantup.com) is a tool that allows you to define a virtual environment with code. A single file allows you to define a basic environment for a virtual machine as well as a series of provisioning actions that prepare the environment for use. Vagrant works by running code (Vagrantfiles) on top of packaged operating system images called boxes. The Vagrant code and box files can be versioned and distributed using automated tooling. This allows you to share virtual machines, which is not much different than the process of software development that uses source control.

Using Vagrant boxes and provisioning controlled by Vagrantfiles not only simplified the process of distributing virtual machines (and updates to virtual machines), but it also made the virtual machines we were working with inexpensive in terms of effort to rebuild. The amazing thing that we found was that Vagrant not only made it simple to distribute virtual machines, but also gave developers more freedom to experiment and make deeper modifications to the code without losing time due to changes in the development environment that could not be rolled back. This flexibility and a simplified on-boarding process for new developers made it much simpler for the team to spend more time doing software development (and tackling that technical debt!), rather than attempting to fix and find problems due to environments.

I've found Vagrant to be an invaluable tool in my work. I hope that this book can be a valuable resource for you in getting started with Vagrant, or perhaps, using Vagrant in new and different ways.

What This Book Covers

Chapter 1, Setting Up Your Environment, covers a few basics about hypervisor technology, the installation of Vagrant and VirtualBox, and some simple recipes to get started with Vagrant machines.

Chapter 2, Single Machine Environments, contains recipes to get started with writing single machine Vagrantfiles, including booting machines, forwarding ports, and customizing the virtual machine environment.

Chapter 3, Provisioning a Vagrant Environment, introduces the concept of provisioning Vagrant machines, installing software, and customizing the environment to develop and deploy software. This chapter focuses on using shell (bash) scripting to modify the Vagrant environment.
Chapter 4, *Provisioning With Configuration Management Tools*, contains simple recipes to provision Vagrant machines with four common configuration management tools: Puppet, Chef, Ansible, and Salt. These tools allow easier configuration of machines that have more complex environments. They also allow Vagrant machines to share the same provisioning instructions as other environments.

Chapter 5, *Networked Vagrant Environments*, contains recipes focused on networking Vagrant machines with external hosts and with each other. We cover a few topics from the basics of assigning host entries to networking a cluster of Vagrant machines with Consul.

Chapter 6, *Vagrant in the Cloud*, contains recipes to use Vagrant with cloud providers (specifically, Amazon Web Services and DigitalOcean). It also contains the use of Hashicorp's Atlas tool to share Vagrant environments with remote users.

Chapter 7, *Packaging Vagrant Boxes*, introduces methods to package Vagrant boxes for others to use. Recipes include the packaging of boxes using manual and automated tools and tips to share your box with others on Atlas.

Appendix A, *Vagrant Plugins*, gives a short introduction on how to extend the capabilities of Vagrant by developing plugins.

Appendix B, *A Puppet Development Environment*, expands on the introduction in Chapter 4, *Provisioning With Configuration Management Tools*, to set up a more robust configuration environment to develop Puppet scripts. While the focus is on using Puppet to provision, similar environments can be created to support the configuration management environment of your choice.

Appendix C, *Using Docker With Vagrant*, is an introduction to use Vagrant to create, deploy, and test Docker (http://docker.io) containers. This appendix introduces techniques to launch Docker containers with Vagrant as well as build and test a complete Docker environment.
In this chapter, we will cover the following topics:

- Creating a local network
- Defining a multimachine environment
- Specifying the order of machine provisioners
- Creating clusters of Vagrant machines

**Introduction**

Standalone Vagrant environments can meet the needs of a variety of use cases. A common case would be using Vagrant to facilitate web and application development. In this case, forwarding the Vagrant guest web server port (usually port 80) to a port on the localhost would allow applications hosted on the web server to be accessed through a localhost address. (For example, opening http://localhost:8080 in a browser.)

The port forwarding model might not work well for a few use cases. For example:

- Situations where a machine must be addressed using a real hostname, either in cases where a web application requires it or when a machine is using SSL certificates.
- Modeling deployment environments where different services are installed on dedicated machines. A common example would be developing a web application where a web application is installed and configured on a machine that connects to a database running on a separate virtual machine.
Modeling clustered environments where virtual machines might register themselves for discovery. As an example, Vagrant can be a useful tool to model and develop systems with Consul (https://consul.io) or CoreOS (http://coreos.com).

Vagrant can be used to assign IP addresses or set up service discovery that allows virtual machines to have fixed (or discoverable) IP addresses to be used by other services. This chapter contains recipes with basic Vagrant networking and use cases where a network of virtual machines is required.

While Vagrant networking makes setting up networks rather simple, keep in mind that virtual machines will still use the local system's RAM and CPU. The number of virtual machines that can be used in a Vagrant network are limited by the resources of the host machine. If you have the need to create larger networks of machines, Vagrant can facilitate the use of cloud providers to create virtual machines using the compute resources of cloud services. This effectively allows you to rent computing space for a development environment. These use cases will be covered in the next chapter.

Creating a local network

Creating a local network is the process of assigning an IP to a Vagrant machine.

Getting ready

Before setting up a network, you might want to consider the type of network you wish to create. Vagrant essentially offers two options:

- A **local** network that limits access to Virtual Machines (VMs) running on the host computer. The hypervisor software typically specifies an address range.

- A **bridged** network that will obtain an IP address from outside the local range. This means that the Vagrant machine can be accessed as any other machine on the host computer network. You can, for example, specify bridged networking if you want your Vagrant machine to be a shared resource among many different people in your office. The downside is that the Vagrant machine will obtain an IP that cannot be controlled by the Vagrant environment and will rely on the larger network environment. This will make it difficult to create machine networks and we will not cover bridged networking in any depth here.

In this recipe, we'll create a simple Vagrant machine running Ubuntu 14.04 LTS and assign an IP to the machine. We'll also discuss how we can use these machines on our host environment.
A quick note regarding static IP addresses

When using a static IP address on a local machine, we'll want to ensure that we are using IP ranges reserved for private networks to avoid any possible collisions with our outside environment. The IP ranges for private networks are established by the Internet Engineering Task Force and are reserved for use by private networks. The three ranges are defined in RFC1918 (http://tools.ietf.org/html/rfc1918) as:

- 10.0.0.0-10.255.255.255 (10/8 prefix)
- 172.16.0.0-172.31.255.255 (172.16/12 prefix)
- 192.168.0.0-192.168.255.255 (192.168/16 prefix)

When assigning static IPs in a Vagrantfile, choose one of these ranges to assign IPs in. More specifically, you'll likely want to assign ranges in either the 172 or 192 ranges, many corporate (or even home) networks use the 10 range for resources located within the wider network by default. Your hypervisor software will typically alert you if you are running into an IP address conflict.

How to do it...

1. Start with a simple Vagrantfile. In this case, we'll start with a basic machine definition:

   ```ruby
   # -*- mode: ruby -*-
   # vi: set ft=ruby :

   VAGRANTFILE_API_VERSION = "2"

   Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
     config.vm.box = "puppetlabs/ubuntu-14.04-64-nocm"
   end
   ```

2. To this configuration, assign an IP to the Vagrant machine using the `config.vm.network` parameter. Add this parameter after the `box` definition:

   ```ruby
   config.vm.network "private_network", ip: "192.168.99.100"
   ```

   This will assign the "192.168.99.100" IP to the Vagrant machine.

3. Start the machine with the `vagrant up` command.
4. Once the machine starts, verify that the IP address has been set by using `vagrant ssh` to access the machine. Once at a command prompt for the Vagrant machine, verify the IP ranges of the machine by using the `ifconfig` command. This will display information about the machine's network environment. For this example, the `inet addr` sections are the most important.

```
2. vagrant@localhost:~$ ifconfig
eth0  Link encap:Ethernet  HWaddr 00:0c:29:e1:d4:2d
       inet6 addr: fe80::2c9:ff:fe1:d42d/64 Scope:Link
       UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
       RX packets:1154 errors:0 dropped:0 overruns:0 frame:0
       TX packets:890 errors:0 dropped:0 overruns:0 carrier:0
       collisions:0 txqueuelen:1000
       RX bytes:194072  (194.0 KB)  TX bytes:99865  (99.8 KB)

eth1  Link encap:Ethernet  HWaddr 00:0c:29:e1:d4:37
       inet6 addr: fe80::2c9:ff:fe1:d437/64 Scope:Link
       UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
       RX packets:102 errors:0 dropped:0 overruns:0 frame:0
       TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
       collisions:0 txqueuelen:1000
       RX bytes:21318  (21.3 KB)  TX bytes:648  (648.0 B)

lo    Link encap:Local Loopback
       inet addr:127.0.0.1  Mask:255.0.0.0
       inet6 addr: ::1/128 Scope:Host
       UP LOOPBACK RUNNING  MTU:65536  Metric:1
       RX packets:0 errors:0 dropped:0 overruns:0 frame:0
       TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
       collisions:0 txqueuelen:0
       RX bytes:0  (0.0 B)  TX bytes:0  (0.0 B)
```

Note that the Vagrant machine has two separate IPs on different interfaces defined here as `eth0` and `eth1`. The machine can respond to either of the IPs (one is assigned by the hypervisor, while the other is defined in the Vagrant configuration).

It is entirely possible that there is only one (the assigned) IP address for our Vagrant machine. For this example, the hypervisor added our IP address as a second interface, while keeping the other address for internal consistency.
Using a static IP address with a hosts file

Now that we have a machine with an assigned static IP address, there are a few ways that can be used to access the machine. In many cases, with static IPs, we will want to refer to the machine with a real hostname (that is, referring to this as `web.local` rather than `192.168.99.100`). Computers are usually assigned these addresses through the **Domain Name System (DNS)** where you register an address and hostname with a DNS entry, but for local development, DNS can be overridden with a local hosts file. On Unix (Linux and OS X included), the hosts file is `/etc/hosts`. Windows machines also have hosts files typically in the `\Windows\system32\drivers\etc\hosts` file, although this has been different for some versions; consult your system documentation for the proper path to the hosts file.

**Warning!**

You will require administrator privileges on your machine to modify your `/etc/hosts` file. Modifying this file can have some adverse effects on your system and even leave your computer open to attack, should an override be added to the hosts file. If you modify this file, make sure that the `localhost` entry is left untouched (with IP address `127.0.0.1`).

You know about every entry added to this file (some system attacks attempt to add entries to the file in order to override DNS entries to sensitive sites in an attempt to trick a user into handing over sensitive data). By default, the only definition in the file is `localhost`, make sure that the only items in here are entries that are added by you or with your explicit permission.

To use our Vagrant machine as a real IP address (say, for instance, `web.local`), we can add a new entry with the IP address assigned in the Vagrantfile. A complete `/etc/hosts` file with only the addition of `web.local` assigned to our static IP of `192.168.99.100` would look like this:

```
127.0.0.1 localhost cthompson
192.168.99.100 web.local
255.255.255.255 broadcasthost
::1 localhost
fe80::1%lo0 localhost

"/etc/hosts" [readonly] 13L, 275C
```
The Vagrant machine can then be accessed using the `web.local` name address. (For example, opening a default web server on the Vagrant machine would be `http://web.local` rather than a forwarded port address of something like `http://localhost:8080`.)

**How it works...**

In this recipe, we assigned a static IP address to a Vagrant address and assigned a URL to this IP. Assigning a static IP address requires a Vagrantfile parameter with an unused IP address on a local network. (Be sure to use addresses in the `local` range as specified in the Getting ready section of this recipe.)

Vagrant itself will use an internal network defined by the hypervisor software. IP routing is managed by a virtualized infrastructure embedded in the hypervisor. You might have noticed this when Vagrant outputs messages about `vmnet` (if using VMware – other messages for different hypervisors) during the bootup cycle.

One issue that you might encounter when starting or stopping many Vagrant hosts (or virtual machines in general) is that an occasional error can be thrown when the virtual networking infrastructure runs into collisions assigning IPs. In these cases, it might be okay to restart the affected virtual machine, but in many cases, a clean reboot of the host system might be required to reset the hypervisor network.

**There's more...**

There are a few different ways that we could manage static IPs and real URLs without manually editing the `/etc/hosts` file, with some simple methods using Vagrant plugins. There are many plugins to choose from and we will not be able to cover all the options. See [https://github.com/mitchellh/vagrant/wiki/Available-Vagrant-Plugins](https://github.com/mitchellh/vagrant/wiki/Available-Vagrant-Plugins) for an up-to-date list of available plugins. Plugins dealing with assigning real addresses fall into two categories:

- **Using `/etc/hosts` files**: There are a number of plugins available to manage host machine's `/etc/hosts` files. One of the most commonly used plugins is the `vagrant-hosts` plugin that can be installed with the command:

  ```bash
  vagrant plugin install vagrant-hosts
  ```

  The `vagrant-hosts` plugin will supply another option, available in the Vagrantfile, that allows assigned IPs to be added to the host machine's `/etc/hosts` files with an additional attribute added along with an IP assignment:

  ```text
  web.vm.provision :hosts
  ```
When starting a Vagrant machine with a plugin that edits the /etc/hosts file, Vagrant will prompt for a password; editing the hosts file will always require administrator privileges. Using plugins to manage this file might be simpler for frequent use, but be sure that all users of the created Vagrant file have the plugin installed.

- **Using /etc/resolver for local DNS**: There are also Vagrant plugins that create local DNS servers and modify the resolver files on the guest and host operating systems. Some of them (such as landrush) are quite fully featured and can cover many complex scenarios for local development. Again, these plugins might require administrator privileges as DNS configuration can also have some adverse effects. You might wish to consider the type of network that you are establishing (whether or not it is a host only DNS setting or a setting shared between guests and hosts) and the operating systems supported by the plugins before choosing an appropriate one.

### See also
- Currently available (and listed) Vagrant plugins are at [https://github.com/mitchellh/vagrant/wiki/Available-Vagrant-Plugins](https://github.com/mitchellh/vagrant/wiki/Available-Vagrant-Plugins)

### Defining a multimachine environment

The primary reason we wish to create networks of Vagrant machines is often because we wish to model an environment of more than one machine. A common example might be the desire to model a web application with dedicated web server machines and database machines, or even an environment that creates a cluster of identical virtual machines.

In this recipe, we will create a simple multimachine environment as well as look at techniques to create clusters of Vagrant machines.

### Getting ready

Before we start with creating an environment of many machines, let's review the technique of defining machine names. When creating a multimachine environment, we'll want to ensure that each machine has a unique name. A unique name can be assigned by defining a new Vagrant machine:

```ruby
Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.define "definedmachine" do |definedmachine|
```
The `config.vm.define` method is how we define machines and specify actions that will be performed on a specific host.

### How to do it...

In this example, we will create a small network of two virtual machines that defines a simple two-tier web application with a web server and a database server. These two servers will be defined in a single Vagrantfile, and we will manage our networks using `/etc/hosts` methods rather than using DNS.

1. Start with a simple Vagrantfile without a machine or box definition:
   ```
   # -*- mode: ruby -*-
   # vi: set ft=ruby :

   VAGRANTFILE_API_VERSION = "2"
   Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
   end
   ```

2. In the `|config|` section, define a database server using a Vagrant machine definition. We'll add some detail about this machine, namely, the box that it will use to boot the machine, and a unique hostname. In a multimachine environment, we'll usually want to define a machine IP (particularly in the case where one Vagrant machine (a web server) will need to connect to another (a database server):
   ```ruby
   config.vm.define "database" do |db|
     db.vm.box = "puppetlabs/ubuntu-14.04-64-nocm"
     db.vm.hostname = "db01"
     db.vm.network "private_network", ip: "192.168.55.100"
   end
   ```

3. Create a second defined machine in a block after the `|db|` code block. This will be the web machine:
   ```ruby
   config.vm.define "web" do |web|
     web.vm.box = "puppetlabs/ubuntu-14.04-64-nocm"
     web.vm.hostname = "web01"
     web.vm.network "private_network", ip: "192.168.55.101"
   end
   ```
4. Verify that both these machines are defined using the `vagrant status` command. This command will provide a list of all defined Vagrant machines in the file:

```
$ vagrant status
Current machine states:
database                 not created (vmware_fusion)
web                      not created (vmware_fusion)
```

This environment represents multiple VMs. The VMs are all listed above with their current state. For more information about a specific VM, run `vagrant status NAME`.

```
vagrantbook-examples/Chapter5/1.2 Defining a Multi-Machine Environment$ vagrant status
```

The `vagrant status` command will provide a list of defined machines, and their status, as well as the provider that will be used.

5. To complete the example, we'll use the shell provisioner to define the `/etc/hosts` file in the web server. This allows the web server to refer to the database server with the `db01` hostname. The complete web server definition will include this provisioning command. (In this case, we will overwrite the `/etc/hosts` file, which will allow our provisioning to be idempotent, although we will need to take care to define the `localhost` entry):

```
config.vm.define "web" do |web|
  web.vm.box = "puppetlabs/ubuntu-14.04-64-nocm"
  web.vm.hostname = "web01"
  web.vm.network "private_network", ip:"192.168.55.101"
  web.vm.provision "shell",
    inline: "echo '127.0.0.1 localhost web01\n192.168.55.100 db01' > /etc/hosts"
end
```
6. Start both machines by executing the `vagrant up` command. This command will return the startup commands of both machines. (In the case of using local hypervisors such as VMware Fusion, the machines will also boot in the order that is specified in the Vagrantfile.)

```
$ vagrant up
Bringing machine 'database' up with 'vmware_fusion' provider...
Bringing machine 'web' up with 'vmware_fusion' provider...
=> database: Cloning VMware VM 'puppetlabs/ubuntu-14.04-64-nocm'. This can take some time...
=> database: Checking if box 'puppetlabs/ubuntu-14.04-64-nocm' is up to date...
=> database: Verifying vmnet devices are healthy...
=> database: Preparing network adapters...
=> database: Starting the VMware VM...
=> database: Waiting for machine to boot. This may take a few minutes...
```

There might be cases where we wish to start a single machine in the Vagrantfile. This can be accomplished by defining the machine that will be booted or provisioned. For example, to only boot the database server, we would execute the `vagrant up database` command.

7. Once the machines have booted, the status can be verified once again using the `vagrant status` command:

```
$ vagrant status
Current machine states:
  database running (vmware_fusion)
  web running (vmware_fusion)

This environment represents multiple VMs. The VMs are all listed above with their current state. For more information about a specific VM, run 'vagrant status NAME'.
```

This environment includes the following VMs:
- `database` running on `vmware_fusion`
- `web` running on `vmware_fusion`
8. Access the web machine by using the `vagrant ssh` command, specifying that we wish to connect to the web machine:

```
$ vagrant ssh web
```

Welcome to Ubuntu 14.04 LTS (GNU/Linux 3.13.0-24-generic x86_64)

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

```
9. Verify that the web server can contact the database with the `db01` hostname:

```
$ ping db01
PING db01 (192.168.55.100) 56(84) bytes of data.
64 bytes from db01 (192.168.55.100): icmp_seq=1 ttl=64 time=0.465 ms
64 bytes from db01 (192.168.55.100): icmp_seq=2 ttl=64 time=0.487 ms
64 bytes from db01 (192.168.55.100): icmp_seq=3 ttl=64 time=0.497 ms
64 bytes from db01 (192.168.55.100): icmp_seq=4 ttl=64 time=0.761 ms
64 bytes from db01 (192.168.55.100): icmp_seq=5 ttl=64 time=0.739 ms
^C
```

```
--- db01 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 399ms
rtt min/avg/max/mdev = 0.465/0.595/0.791/0.142 ms
```

With this network in place, we can proceed to the task of setting up our web application by using Vagrant provisioners (such as shell scripts, Puppet, Chef, and so on) to install and configure database and web servers with the appropriate software and configurations for your application.

**How it works...**

Setting up multimachine environments in this simple context works by:

- **Defining specific hosts and hostnames:** In the example, we defined a specific web server and a specific database server.

- **Defining the network settings required to make our environment work:** Specifically, we gave our web server the ability to locate the database server by modifying the web server's `/etc/hosts` file. This allows the web server to find the database server, but it will not allow the database server to contact the web server.
As such, this environment is relatively static, but will not require additional infrastructure to manage network and/or DNS; we have an environment where a web (or application server) can connect to a database server using a hostname defined in an /etc/hosts file. For many scenarios, this is sufficient to allow for local development.

Defining different machines locally will also allow for separation of concerns. There might be cases where a developer is actively doing web server development using local provisioners, but the details of how the database is created are not particularly important. In this case, it might be desirable to allow the web server to use local provisioning scripts, allowing the database server to be provisioned using a centralized provisioning tool such as a Puppet master. Using separate machines allows developers to model entire systems while working, hopefully mimicking a production deployment early in the development process.

### Specifying the order of machine provisioners

When setting up multimachine environments, it is often important to specify how machines will provision and the order in which they will provision.

#### Getting ready

Before we start with an example, there are a few important things to keep in mind about the ordering of Vagrant resources:

- Ordering and dependencies in Vagrant environments are often dependent on the type of resource being provisioned. In the case of desktop hypervisors, a Vagrant boot cycle will proceed in the order in which resources are defined as the Vagrantfile will wait for the process to exit. In the case of provisioning cloud environments, the return to the calling Vagrant process will be nearly immediate (as the call itself is to an asynchronous RESTful API), so the boot order can be difficult to enforce without modifying the Vagrantfile to use cloud service APIs in order to check for boot health.

- Vagrant will also evaluate code blocks from the outside in order with the code in the inner blocks either overriding (should the property be the same) or in an outside in order, which is especially important for provisioners.

In this recipe, we will demonstrate overriding and ordering in a simple Vagrantfile.

#### How to do it...

1. Start with a simple Vagrantfile. In this case, simply a Vagrantfile with a default box name (something that could be generated with a vagrant init command):

```ruby
# -*- mode: ruby -*-
# vi: set ft=ruby :
VAGRANTFILE_API_VERSION = "2"
```
Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "puppetlabs/ubuntu-14.04-64-nocm"
end

2. Add a default hostname and provisioner to the Vagrantfile below the box definition:
   
   ```ruby
   config.vm.hostname = "override_me"
   config.vm.provision "shell", inline: "echo 'First Command to Execute'"
   ```

   With this file, a `vagrant up` command would boot with a hostname of `override_me` and text from the first provisioner would be output to the console.

3. Add a machine definition block with an override for the hostname and box type. In this case, we will name the machine `second`, override the box type (to a box with Puppet installed), and execute a second provisioner. Our complete Vagrantfile looks like this:

   ```ruby
   VAGRANTFILE_API_VERSION = "2"
   Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
     config.vm.define "second" do |second|
       second.vm.box = "puppetlabs/ubuntu-14.04-64-puppet"
       second.vm.hostname = "second"
       second.vm.provision "shell", inline: "echo 'Second Command to Execute'"
     end
     config.vm.box = "puppetlabs/ubuntu-14.04-64-nocm"
     config.vm.hostname = "first"
     config.vm.provision "shell", inline: "echo 'First Command to Execute'"
   end
   ```
4. Execute this Vagrantfile with the `vagrant up` command. The output will show us the results of our hostname and the order of provisioning:

```
$ vagrant up
Bringing machine 'second' up with 'vmware_fusion' provider...
  => second: Cloning VMware VM: 'puppetlabs/ubuntu-14.04-64-puppet'. This can take some time...
  => second: Checking if box 'puppetlabs/ubuntu-14.04-64-puppet' is up to date...
  => second: Verifying vmmet devices are healthy...
  => second: Preparing network adapters...
  => second: Starting the VMware VM...
  => second: Waiting for machine to boot. This may take a few minutes...
    second: SSH address: 192.168.38.129:22
    second: SSH username: vagrant
    second: SSH auth method: private key
  => second: Machine booted and ready!
  => second: Forwarding ports...
    second: -- 22 => 2222
  => second: Setting hostname...
  => second: Configuring network adapters within the VM...
  => second: Waiting for HGFS kernel module to load...
  => second: Enabling and configuring shared folders...
    second: -- /Volumes/WD HDD/vagrantbook/vagrantbook-examples/Chapter5/1.3 Ordering Provisioning: /vagrant
  => second: Running provisioner: shell...
    second: Running: inline script
      => second: stdin: is not a tty
    => second: First Command to Execute
  => second: Running provisioner: shell...
    second: Running: inline script
      => second: stdin: is not a tty
    => second: Second Command to Execute
```

Note a few results from this Vagrantfile:

- Despite an initial name defined outside a block, the booted machine (and in this case, there is only one) is referred to by the `second` hostname.
- The second host booted with a Vagrant box that has Puppet installed.
- Two separate provisioners executed on our box (one defined globally and one defined within our `second` code block).
The ordering of execution and overriding is important in multimachine environments, as we can define provisioners that can run globally on all machines (such as an `apt-get update` command to be executed prior to other provisioning on a network of Ubuntu machines) or to define global rules with a few exceptions, such as the type of Vagrant box that will be available in the network.

**How it works...**

Vagrant defines variables and executions using ID fields for each parameter. Some parameters can only be defined once, such as the rule that *each Vagrant machine can only be started from a single box*, which causes box definitions in code blocks to override global parameters. In this case, we have a box override defined in our Vagrantfile:

```ruby
config.vm.box = "puppetlabs/ubuntu-14.04-64-nocm"
config.vm.define "second" do |second|
  second.vm.box = "puppetlabs/ubuntu-14.04-64-puppet"
end
```

The override specifies that the `second` box will use the `puppetlabs/ubuntu-14.04-64-puppet` box file.

Provisioners, on the other hand, are not overwritten as they are executed in an *outside in* order. Provisioners defined in a code block are executed after other provisioners outside the block are executed in a top-down manner. In this case, the order specified in the Vagrantfile caused the output of First Command to Execute, although it was listed below the code block:

```ruby
config.vm.define "second" do |second|
  second.vm.provision "shell", inline: "echo 'Second Command to Execute'"
end
config.vm.provision "shell", inline: "echo 'First Command to Execute'"
```

By default, provisioners are assigned different IDs, so overriding a provisioner requires specification of an ID in the Vagrantfile. Specifying an ID parameter will cause provisioners of identical IDs to perform an override. In this example, we can modify our provisioners to include the `shell_provisioner` ID definition:

```ruby
config.vm.define "second" do |second|
  second.vm.provision "shell", inline: "echo 'Second Command to Execute'", id:"shell_provisioner"
end
config.vm.provision "shell", inline: "echo 'First Command to Execute'", id:"shell_provisioner"
```
Networked Vagrant Environments

With identical ID tags, executing a Vagrant provision operation only echoes output from the provisioner in the `second` code block:

```ruby
===> second: Second Command to Execute
```

The ordering and overriding of provisioners and variables is especially important in multimachine Vagrant environments. A multimachine Vagrantfile can specify global parameters (such as boxes or common provisioning tasks) that allow for individual machines to override the global parameters.

## Creating clusters of Vagrant machines

While the scenario of mimicking defined application architectures (for example, the two-tier or three-tier web application) can be accomplished using simple hosts files and hosts file modifications, creating clusters of Vagrant machines will require a bit of additional complexity, namely, the ability for machines to discover one another using either DNS servers or through service discovery.

In this example, we will create a cluster of Vagrant machines that can communicate with DNS connections using two additional tools:

- **Consul** ([https://consul.io](https://consul.io)): This is a tool that allows services and machines to discover one another over a distributed network. In our case, we will use Consul very simply and set up a single Consul server that will serve multiple agents. We will, for this example, also limit our use of Consul to `node` discovery. This will essentially define a local DNS infrastructure.

- **Dnsmasq** ([http://www.thekelleys.org.uk/dnsmasq/doc.html](http://www.thekelleys.org.uk/dnsmasq/doc.html)): This is a utility that allows local services (such as Consul) to serve DNS requests from local processes. In this case, Dnsmasq allows our system to use the DNS interface of a local Consul agent in order to serve DNS requests.

### Getting ready

This recipe will install a number of different services using the combination of shell provisioners and Puppet. We will highlight some of the important aspects of the approach (the full source code is available in examples provided in the book).

### How to do it...

1. Start our example with a simple Vagrantfile. This Vagrantfile will define a global box file that will be used to start all our machines. We'll choose a box that has the Puppet provisioner installed, as this is how we will provision the Consul server and agents:

```ruby
# -*- mode: ruby -*-
# vi: set ft=ruby :
```
VAGRANTFILE_API_VERSION = "2"
Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  # Define a global box file to be used by all machines.
  config.vm.box = "puppetlabs/ubuntu-14.04-64-puppet"
end

2. Define our Consul server. In this case, we're also going to use a variable to define a static network IP that we can point our cluster members to in order to join the cluster. On the server, we will also add provisioners that will execute an apt-update command (and install an unzip program) prior to executing the puppet run command. The Puppet scripts will install and initialize the Consul server (see the code example for full details):

VAGRANTFILE_API_VERSION = "2"
$consul_server_ip = "192.168.30.130"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  # Define a global box file to be used by all machines.
  config.vm.box = "puppetlabs/ubuntu-14.04-64-puppet"
  # Create and provision a Consul server machine.
  config.vm.define "consul" do |consul|
    consul.vm.hostname = "consul"
    consul.vm.network "private_network", ip: $consul_server_ip
    consul.vm.provision "shell", inline: "apt-get update && apt-get install -y unzip"

    consul.vm.provision "puppet" do |puppet|
      puppet.manifests_path = "puppet/manifests"
      puppet.module_path = "puppet/modules"
      puppet.manifest_file = "site.pp"
    end
  end
end

3. Define a variable that allows us to create an arbitrary number of cluster members in our system. Add this variable before the definition of the consul_server_ip variable in the previous step. These variables are global throughout the Vagrantfile and can be used by each Vagrant machine defined. In fact, we will use the consul_server_ip variable when we instruct our cluster members to join the cluster in this example:

# Define a variable - the number of web nodes.
$cluster_nodes = 3
$consul_server_ip = "192.168.30.130"
4. Define the Consul server and provisioning steps. This will include two provisioning steps: a shell script that updates the apt-repositories command (this is a step that is only necessary on Ubuntu or Debian Linux distributions) and installs the unzip package. The second runs the puppet apply provisioner against the Vagrant machine. The Puppet scripts will install and start the Consul server:

```ruby
config.vm.define "consul" do |consul|
  consul.vm.hostname = "consul"
  consul.vm.network "private_network", ip: $consul_server_ip
  consul.vm.provision "shell", inline: "apt-get update && apt-get install unzip"

  consul.vm.provision "puppet" do |puppet|
    puppet.manifests_path = "puppet/manifests"
    puppet.module_path    = "puppet/modules"
    puppet.manifest_file  = "site.pp"
  end
end
```

5. With the Consul server in place, we'll now define the cluster members. Recall that we defined a parameter named $cluster_nodes (we'll use this to create a number of Vagrant machines). We'll do this by using a Ruby iterator. Create a new code block that contains this iterator:

```ruby
(1..$cluster_nodes).each do |i|
  << Code To Define Nodes >>
end
```

This will create an execution loop that will define the number of desired machines.

6. In the loop, define a virtual machine by using the i iterator to define a unique name to the cluster. We can define the vm_name constant and assign this constant as the hostname of our machine:

```ruby
config.vm.define vm_name = "cluster%02d" % i do |cluster|
  cluster.vm.hostname = vm_name
end
```
With the definition of the cluster machines in place, the Vagrant file can be verified by executing the `vagrant status` command. This command should return a list of all the defined machines, including those defined in our looping construct:

```
# vagrant status
Current machine states:
  consul      not created (vmware_fusion)
  cluster01   not created (vmware_fusion)
  cluster02   not created (vmware_fusion)
  cluster03   not created (vmware_fusion)
```

This environment represents multiple VMs. The VMs are all listed above with their current state. For more information about a specific VM, run `vagrant status NAME`.

7. Now, define the provisioning steps required to join the cluster virtual machines to the Consul cluster. We’ll do this in three steps for a client: the update for our Ubuntu/Debian machines, a `puppet run` command to install and configure the clients, and finally a step to execute a `join` command using the defined server IP:

```
(1..$cluster_nodes).each do |i|
  config.vm.define vm_name = "cluster%02d" % i do |cluster|
    cluster.vm.hostname = vm_name
    cluster.vm.provision "shell", inline: "apt-get update && apt-get install -y unzip"
    cluster.vm.provision "puppet" do |puppet|
      puppet.manifests_path = "puppet/manifests"
      puppet.module_path    = "puppet/modules"
      puppet.manifest_file  = "site.pp"
    end
    cluster.vm.provision "shell", inline: "consul join #{$consul_server_ip}" end
end
```
8. With all the provisioners (including our Puppet modules) in place, start the environment with the `vagrant up` command. This command should note that four machines will start:

```
$ vagrant up
Bringing machine 'consul' up with 'vmware_fusion' provider...
Bringing machine 'cluster01' up with 'vmware_fusion' provider...
Bringing machine 'cluster02' up with 'vmware_fusion' provider...
Bringing machine 'cluster03' up with 'vmware_fusion' provider...
$ consul: Cloning VMware VM: 'puppetlabs/ubuntu-14.04-64-puppet'. This can take some time...
```

Starting these separate machines could take a while to boot, but after a few minutes, the Vagrant startup should complete.

9. Access the first cluster machine with the `vagrant ssh cluster01` command.

10. Verify that a Consul cluster is active by executing the `consul members` command. This should return the list of three servers:

```
vagrant@cluster01:~/~$ consul members
  Node   Address  Status  Type   Build Protocol
  consul 192.168.30.130:8301 alive server 0.4.0 2
  cluster01 192.168.30.129:8301 alive client 0.4.0 2
  cluster02 192.168.30.131:8301 alive client 0.4.0 2
vagrant@cluster01:~/~$ 
```

11. The individual cluster members can also be pinged using the Consul DNS interface. From the `cluster01` machine, the `cluster02` machine can be pinged with the `cluster02.node.vagrant.consul` hostname:

```
vagrant@cluster01:~/~$ ping cluster02.node.vagrant.consul
PING cluster02.node.vagrant.consul (192.168.30.131) 56(84) bytes of data.
64 bytes from 192.168.30.131: icmp_seq=1 ttl=64 time=0.335 ms
64 bytes from 192.168.30.131: icmp_seq=2 ttl=64 time=0.608 ms
64 bytes from 192.168.30.131: icmp_seq=3 ttl=64 time=0.370 ms
--- cluster02.node.vagrant.consul ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2001ms
rtt min/avg/max/mdev = 0.335/0.464/0.608/0.154 ms
vagrant@cluster01:~/~$ 
```
12. Verify that other nodes can be pinged by using the `<hostname>.node.vagrant.consul` pattern. In this case, we should be able to ping `consul.node.vagrant.consul` and `cluster02.node.vagrant.consul` with them responding as a normal host. In this case, however, rather than using a centralized DNS server, the hosts in the `consul` domain are identified by the DNS interface of a local Consul agent.

With the machines running and responding to DNS, the machines consist of an effective cluster. We can expand our provisioning to install software that we wish to run on clustered machines (such as load-balanced web servers or shared database instances).

### How it works...

In this example, the cluster is effectively bound together with Consul. In our simple environment (only a handful of machines), we created a dedicated Consul server and connected agents through this server IP. It should be noted that a larger Consul deployment would use a number of servers possibly spread across geographic distributions and data centers. Consul is designed to allow for service discovery and failover for large clusters.

In our recipe, we are using Consul to provide flexible DNS services to a number of cluster members. There is no requirement (other than host system resources) on the number of servers that can join the cluster. In each case, the DNS entry follows the pattern:

```
<hostname>.node.vagrant.consul
```

The hostname can also be discovered without prior knowledge for the rest of the cluster by using the `cluster members` command to retrieve a list of hostnames that are present in the cluster. The reason for the longer DNS name is that the DNS interface to Consul will define machines by hostname (defined by the machine), the type of service being accessed (in this case, a node), the definition of a data center (vagrant), and the standard top-level domain defined by the Consul interface of `consul`. The datacenter parameter is defined in Consul configurations present on the agent and server. We simplify the deployment by specifying a single data center (or cluster of machines) named `vagrant`.

The Consul environment is defined by using the Puppet scripts executed by the `puppet apply` provisioner. The Puppet scripts are shared by all environments with each environment being given a catalog by definitions in the `site.pp` file. The `site.pp` file defines two types of hosts: servers and cluster members:

```puppet
node /^consul/{
  class{"consul::server": }
}
node /^cluster/
  include consul
```
The Consul module referenced in this recipe does four basic things:

- Installs the Consul software from a released ZIP package
- Configures Consul to run as a service
- Configures Consul to run in server or agent mode through configuration files
- Installs and configures Dnsmasq to forward DNS requests for the consul domain to the Consul agent

With a simple Consul server/agent configuration, we can create clusters of Vagrant machines without configuring DNS servers or relying on machine-specific configurations. We can also use this infrastructure to expand our Vagrant environment to different hypervisor or cloud computing environments.

There's more...

There are many different ways we could implement similar solutions, Consul is only one of the choices. This list is not an exhaustive list of possibilities, but there are some more popular methods to manage clusters of machines, including our clustered Vagrant environment.

### Configuring DNS with plugins

There are a number of Vagrant plugins that create lightweight DNS servers to serve hostnames and IP addresses to servers configured to use the lightweight DNS server. This would be sufficient for simple local clusters, although plugins might not be accessible in all deployment environments.

### Configuring a cluster with etcd

Another solution to cluster machines is provided by the CoreOS project ([http://coreos.com](http://coreos.com)). CoreOS aims to create massive clusters of CoreOS machines that operate nearly entirely by managing Docker container deployments with etcd acting as a service discovery layer for the containers as well as the fleet orchestration tool. The CoreOS project hosts a project demonstrating this type of clustering with CoreOS at [https://github.com/coreos/coreos-vagrant](https://github.com/coreos/coreos-vagrant).

### Clustering with Apache Mesos

Another popular method for clustering and cluster management is Apache Mesos. The Mesos project also provides a Vagrant-based project to learn how to manage servers using Mesos at [https://mesosphere.com/docs/getting-started/](https://mesosphere.com/docs/getting-started/).
See also

- **Consul** (https://consul.io): a tool for service discovery. Consul is also sponsored by HashiCorp, the same company that supports Vagrant itself.
- **Running CoreOS on Vagrant** (https://coreos.com/docs/running-coreos/platforms/vagrant/): a project provided by the CoreOS project to start clusters of CoreOS machines.
- **Apache Mesos** (http://mesos.apache.org) and **Mesosphere** (https://mesosphere.com).
Where to buy this book

You can buy Vagrant Virtual Development Environment Cookbook from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.