Thomas Uphill, John Arundel
Neependra Khare, Hideto Saito,
Hui-Chuan Chloe Lee, Ke-Jou Carol Hsu

DevOps: Puppet, Docker, and Kubernetes

Learning Path

Get hands-on recipes to automate and manage Linux containers with the Docker 1.6 environment and jump-start your Puppet development

Packt

DevOps: Puppet, Docker, and Kubernetes

Get hands-on recipes to automate and manage Linux containers with the Docker 1.6 environment and jump-start your Puppet development

A course in three modules

BIRMINGHAM - MUMBAI

DevOps: Puppet, Docker, and Kubernetes

Published on: March 2017

Production reference: 1150317

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN: 978-1-78829-761-5

`www.packtpub.com`

Credits

Authors

Thomas Uphill

John Arundel

Neependra Khare

Hideto Saito

Hui-Chuan Chloe Lee

Ke-Jou Carol Hsu

Reviewers

Dhruv Ahuja

James Fryman

Jeroen Hooyberghs

Pedro Morgado

Scott Collier

Julien Duponchelle

Allan Espinosa

Vishnu Gopal

Matt Ma

Content Development Editor

Juliana Nair

Graphics

Kirk D'Penha

Production Coordinator

Shantanu N. Zagade

Preface

With so many IT management and DevOps tools on the market, both open source and commercial, it's difficult to know where to start. DevOps is incredibly powerful when implemented correctly, here's how to get it done.

What this learning path covers

Module 1, Puppet Cookbook (third edition), this module covers all aspects of your puppet infrastructure using simple easy to follow recipes that are independent and can be used to solve real world problems quickly.

Puppet Cookbook takes the reader from a basic knowledge of Puppet to a complete and expert understanding of Puppet's latest and most advanced features.
With emphasis on real-world implementation, this book delves into various aspects of writing good Puppet code, including using Puppet community style, checking your manifests with puppet-lint and community best practices. It then shows the readers how to set up Puppet for the first time, including instructions on installing Puppet, creating your first manifests, using version control with Puppet and so on. You will also learn to write better manifests, manage resources, files and applications. You'll then be introduced to powerful tools that have grown up around Puppet, including Hiera, Facter, and rspec-puppet. Finally, you will also learn to master common Monitoring, Reporting, and Troubleshooting techniques.

Updated with the latest advancements and best practices, this book gives you a clear view on how to "connect the dots" and expands your understanding to successfully use and extend Puppet.

Module 2, Docker Cookbook, this module aims to help you get working with Docker by providing you with step-by-step recipes that enable you to effectively deploy Docker in your development, test, and production environments.

You will start with verifying and installing Docker on different environments and look into understanding and working with containers and images. Next, you will move on to study the operations related to images. You then proceed to learn about network and data management for containers and how to build an environment for Continuous Integration with the help of services from companies like Shippable and Drone. The book then explores the RESTful APIs provided by Docker to perform different operations like image/container operations before taking a look at the Docker Remote API client. The book ends with a look at logs and troubleshooting Docker to solve issues and bottlenecks.

Module 3, Kubernetes Cookbook, Kubernetes is Google's solution to managing a cluster of containers. Kubernetes provides a declarative API to manage clusters while giving us a lot of flexibility. This book will provide you with recipes to better manage containers in different scenarios in production using Kubernetes.

We will start by giving you a quick brush up on how Kubernetes works with containers along with an overview of the main Kubernetes features such as Pods, Replication Controllers, and more. Next, we will teach you how to create Kubernetes cluster and how to run programs on Kubernetes. We'll explain features such as High Availability Kubernetes master setup, using Kubernetes with Docker, and orchestration with Kubernetes using AWS. Later, will show you how to use Kubernetes-UI, and how to set up and manage Kubernetes clusters on the cloud and bare metal.

Upon completion of this book, you will be able use Kubernetes in production and will have a better understanding of how to manage your containers using Kubernetes.

What you need for this learning path

The primary softwares required are as follows:

- Puppet 3.7.3
- Kubernetes 1.1.3Java
- Etcd 2.1.1
- Flanneld 0.5.2
- Docker 1.7.1
- Kubernetes 1.2.2
- Etcd 2.3.1
- Amazon Web Services

- entOS
    - 7.1/ubuntu
    - 14.04/Amazeon
    - Linux 2015.09
- Debian and Enterprise Linux-based distributions

Who this learning path is for

This Learning Path is for developers, system administrators, and DevOps engineers who want to use Puppet, Docker, and Kubernetes in their development, QA, or production environments. This Learning Path assumes experience with Linux administration and requires some experience with command-line usage and basic text file editing.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the course's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt course, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this course from your account at `http://www.packtpub.com`. If you purchased this course elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the course in the **Search** box.
5. Select the course for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this course from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the course is also hosted on GitHub at `https://github.com/PacktPublishing/DevOps-Puppet-Docker-and-Kubernetes`. We also have other code bundles from our rich catalog of books, videos, and courses available at `https://github.com/PacktPublishing/`. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our courses—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this course. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your course, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the course in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this course, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

Module 1: Puppet Cookbook

Module 1

Puppet Cookbook

Jump-start your puppet deployment using engaging and practical recipes

1
Puppet Language and Style

"Computer language design is just like a stroll in the park. Jurassic Park, that is."

— Larry Wall

In this chapter, we will cover the following recipes:

- Adding a resource to a node
- Using **Facter** to describe a node
- Installing a package before starting a service
- Installing, configuring, and starting a service
- Using community **Puppet** style
- Creating a manifest
- Checking your manifests with Puppet-lint
- Using modules
- Using standard naming conventions
- Using inline templates
- Iterating over multiple items
- Writing powerful conditional statements
- Using regular expressions in if statements
- Using selectors and case statements
- Using the in operator
- Using regular expression substitutions
- Using the future parser

Introduction

In this chapter, we'll start with the basics of Puppet syntax and show you how some of the syntactic sugar in Puppet is used. We'll then move on to how Puppet deals with dependencies and how to make Puppet do the work for you.

We'll look at how to organize and structure your code into modules following community conventions, so that other people will find it easy to read and maintain your code. I'll also show you some powerful features of Puppet language, which will let you write concise, yet expressive manifests.

Adding a resource to a node

This recipe will introduce the language and show you the basics of writing Puppet code. A beginner may wish to reference *Puppet 3: Beginner's Guide*, *John Arundel*, *Packt Publishing* in addition to this section. Puppet code files are called manifests; manifests declare resources. A resource in Puppet may be a type, class, or node. A type is something like a file or package or anything that has a type declared in the language. The current list of standard types is available on puppetlabs website at `https://docs.puppetlabs.com/references/latest/type.html`. I find myself referencing this site very often. You may define your own types, either using a mechanism, similar to a subroutine, named **defined types**, or you can extend the language using a custom type. Types are the heart of the language; they describe the things that make up a node (node is the word Puppet uses for client computers/devices). Puppet uses resources to describe the state of a node; for example, we will declare the following package resource for a node using a site manifest (`site.pp`).

How to do it...

Create a `site.pp` file and place the following code in it:

```
node default {
  package { 'httpd':
    ensure => 'installed'
  }
}
```

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

How it works...

This manifest will ensure that any node, on which this manifest is applied, will install a package called `'httpd'`. The `default` keyword is a wildcard to Puppet; it applies anything within the node default definition to any node. When Puppet applies the manifest to a node, it uses a **Resource Abstraction Layer** (**RAL**) to translate the package type into the package management system of the target node. What this means is that we can use the same manifest to install the `httpd` package on any system for which Puppet has a **Provider** for the package type. Providers are the pieces of code that do the real work of applying a manifest. When the previous code is applied to a node running on a YUM-based distribution, the YUM provider will be used to install the `httpd` RPM packages. When the same code is applied to a node running on an **APT**-based distribution, the APT provider will be used to install the `httpd` DEB package (which may not exist, most debian-based systems call this package `apache2`; we'll deal with this sort of naming problem later).

Using Facter to describe a node

Facter is a separate utility upon which Puppet depends. It is the system used by Puppet to gather information about the target system (node); `facter` calls the nuggets of information facts. You may run `facter` from the command line to obtain real-time information from the system.

How to do it...

1. Use `facter` to find the current uptime of the system, the uptime fact:

   ```
   t@cookbook ~$ facter uptime
   0:12 hours
   ```

2. Compare this with the output of the Linux uptime command:

   ```
   t@cookbook ~$ uptime
    01:18:52 up 12 min,  1 user,  load average: 0.00, 0.00, 0.00
   ```

How it works...

When `facter` is installed (as a dependency for puppet), several fact definitions are installed by default. You can reference each of these facts by name from the command line.

There's more...

Running `facter` without any arguments causes `facter` to print all the facts known about the system. We will see in later chapters that `facter` can be extended with your own custom facts. All facts are available for you to use as variables; variables are discussed in the next section.

Variables

Variables in Puppet are marked with a dollar sign ($) character. When using variables within a manifest, it is preferred to enclose the variable within braces `"${myvariable}"` instead of `"$myvariable"`. All of the facts from `facter` can be referenced as top scope variables (we will discuss scope in the next section). For example, the **fully qualified domain name** (**FQDN**) of the node may be referenced by `"${::fqdn}"`. Variables can only contain alphabetic characters, numerals, and the underscore character (_). As a matter of style, variables should start with an alphabetic character. Never use dashes in variable names.

Scope

In the variable example explained in the *There's more...* section, the fully qualified domain name was referred to as `${::fqdn}` rather than `${fqdn}`; the double colons are how Puppet differentiates scope. The highest level scope, top scope or global, is referred to by two colons (`::`) at the beginning of a variable identifier. To reduce namespace collisions, always use fully scoped variable identifiers in your manifests. For a Unix user, think of top scope variables as the / (root) level. You can refer to variables using the double colon syntax similar to how you would refer to a directory by its full path. For the developer, you can think of top scope variables as global variables; however, unlike global variables, you must always refer to them with the double colon notation to guarantee that a local variable isn't obscuring the top scope variable.

Installing a package before starting a service

To show how ordering works, we'll create a manifest that installs `httpd` and then ensures the `httpd` package service is running.

How to do it...

1. We start by creating a manifest that defines the service:

```
service {'httpd':
  ensure  => running,
  require => Package['httpd'],
}
```

2. The service definition references a package resource named `httpd`; we now need to define that resource:

```
package {'httpd':
  ensure => 'installed',
}
```

How it works...

In this example, the package will be installed before the service is started. Using `require` within the definition of the `httpd` service ensures that the package is installed first, regardless of the order within the manifest file.

Capitalization

Capitalization is important in Puppet. In our previous example, we created a package named `httpd`. If we wanted to refer to this package later, we would capitalize its type (`package`) as follows:

```
Package['httpd']
```

To refer to a class, for example, the `something::somewhere` class, which has already been included/defined in your manifest, you can reference it with the full path as follows:

```
Class['something::somewhere']
```

When you have a defined type, for example the following defined type:

```
example::thing {'one':}
```

The preceding resource may be referenced later as follows:

```
Example::Thing['one']
```

Knowing how to reference previously defined resources is necessary for the next section on metaparameters and ordering.

Learning metaparameters and ordering

All the manifests that will be used to define a node are compiled into a catalog. A catalog is the code that will be applied to configure a node. It is important to remember that manifests are not applied to nodes sequentially. There is no inherent order to the application of manifests. With this in mind, in the previous `httpd` example, what if we wanted to ensure that the `httpd` process started after the `httpd` package was installed?

We couldn't rely on the `httpd` service coming after the `httpd` package in the manifests. What we have to do is use metaparameters to tell Puppet the order in which we want resources applied to the node. Metaparameters are parameters that can be applied to any resource and are not specific to any one resource type. They are used for catalog compilation and as hints to Puppet but not to define anything about the resource to which they are attached. When dealing with ordering, there are four metaparameters used:

- `before`
- `require`
- `notify`
- `subscribe`

The `before` and `require` metaparameters specify a direct ordering; `notify` implies `before` and `subscribe` implies `require`. The `notify` metaparameter is only applicable to services; what notify does is tell a service to restart after the notifying resource has been applied to the node (this is most often a package or file resource). In the case of files, once the file is created on the node, a notify parameter will restart any services mentioned. The `subscribe` metaparameter has the same effect but is defined on the service; the service will subscribe to the file.

Trifecta

The relationship between package and service previously mentioned is an important and powerful paradigm of Puppet. Adding one more resource-type file into the fold, creates what puppeteers refer to as the **trifecta**. Almost all system administration tasks revolve around these three resource types. As a system administrator, you install a package, configure the package with files, and then start the service.

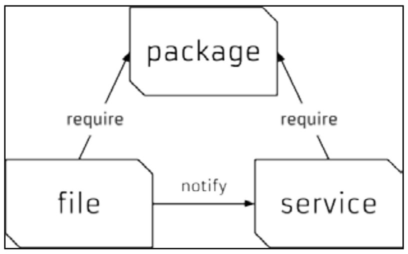


Diagram of Trifecta (Files require package for directory, service requires files and package)

Idempotency

A key concept of Puppet is that the state of the system when a catalog is applied to a node cannot affect the outcome of Puppet run. In other words, at the end of Puppet run (if the run was successful), the system will be in a known state and any further application of the catalog will result in a system that is in the same state. This property of Puppet is known as idempotency. **Idempotency** is the property that no matter how many times you do something, it remains in the same state as the first time you did it. For instance, if you had a light switch and you gave the instruction to turn it on, the light would turn on. If you gave the instruction again, the light would remain on.

Installing, configuring, and starting a service

There are many examples of this pattern online. In our simple example, we will create an Apache configuration file under `/etc/httpd/conf.d/cookbook.conf`. The `/etc/httpd/conf.d` directory will not exist until the `httpd` package is installed. After this file is created, we would want `httpd` to restart to notice the change; we can achieve this with a notify parameter.

How to do it...

We will need the same definitions as our last example; we need the package and service installed. We now need two more things. We need the configuration file and index page (`index.html`) created. For this, we follow these steps:

1. As in the previous example, we ensure the service is running and specify that the service requires the `httpd` package:

   ```
   service {'httpd':
     ensure => running,
     require => Package['httpd'],
   }
   ```

2. We then define the package as follows:

   ```
   package {'httpd':
     ensure => installed,
   }
   ```

3. Now, we create the `/etc/httpd/conf.d/cookbook.conf` configuration file; the `/etc/httpd/conf.d` directory will not exist until the `httpd` package is installed. The `require` metaparameter tells Puppet that this file requires the `httpd` package to be installed before it is created:

   ```
   file {'/etc/httpd/conf.d/cookbook.conf':
     content => "<VirtualHost *:80>\nServername
       cookbook\nDocumentRoot
       /var/www/cookbook\n</VirtualHost>\n",
     require => Package['httpd'],
     notify => Service['httpd'],
   }
   ```

4. We then go on to create an `index.html` file for our virtual host in `/var/www/cookbook`. This directory won't exist yet, so we need to create this as well, using the following code:

   ```
   file {'/var/www/cookbook':
     ensure => directory,
   }
   file {'/var/www/cookbook/index.html':
     content => "<html><h1>Hello World!</h1></html>\n",
     require => File['/var/www/cookbook'],
   }
   ```

How it works...

The `require` attribute to the file resources tell Puppet that we need the `/var/www/cookbook` directory created before we can create the `index.html` file. The important concept to remember is that we cannot assume anything about the target system (node). We need to define everything on which the target depends. Anytime you create a file in a manifest, you have to ensure that the directory containing that file exists. Anytime you specify that a service should be running, you have to ensure that the package providing that service is installed.

In this example, using metaparameters, we can be confident that no matter what state the node is in before running Puppet, after Puppet runs, the following will be true:

- `httpd` will be running
- The `VirtualHost` configuration file will exist
- `httpd` will restart and be aware of the `VirtualHost` file
- The `DocumentRoot` directory will exist
- An `index.html` file will exist in the `DocumentRoot` directory

Using community Puppet style

If other people need to read or maintain your manifests, or if you want to share code with the community, it's a good idea to follow the existing style conventions as closely as possible. These govern such aspects of your code as layout, spacing, quoting, alignment, and variable references, and the official puppetlabs recommendations on style are available at `http://docs.puppetlabs.com/guides/style_guide.html`.

How to do it...

In this section, I'll show you a few of the more important examples and how to make sure that your code is style compliant.

Indentation

Indent your manifests using two spaces (not tabs), as follows:

```
service {'httpd':
  ensure  => running,
}
```

Quoting

Always quote your resource names, as follows:

```
package { 'exim4':
```

We cannot do this as follows though:

```
package { exim4:
```

Use single quotes for all strings, except when:

- The string contains variable references such as `"${::fqdn}"`
- The string contains character escape sequences such as `"\n"`

Consider the following code:

```
file { '/etc/motd':
  content => "Welcome to ${::fqdn}\n"
}
```

Puppet doesn't process variable references or escape sequences unless they're inside double quotes.

Always quote parameter values that are not reserved words in Puppet. For example, the following values are not reserved words:

```
name => 'Nucky Thompson',
mode => '0700',
owner => 'deploy',
```

However, these values are reserved words and therefore not quoted:

```
ensure => installed,
enable => true,
ensure => running,
```

False

There is only one thing in Puppet that is false, that is, the word `false` without any quotes. The string `"false"` evaluates to `true` and the string `"true"` also evaluates to true. Actually, everything besides the literal false evaluates to true (when treated as a Boolean):

```
if "false" {
  notify { 'True': }
}
if 'false' {
  notify { 'Also true': }
}
```

```
if false {
  notify { 'Not true': }
}
```

When this code is run through `puppet apply`, the first two notifies are triggered. The final notify is not triggered; it is the only one that evaluates to `false`.

Variables

Always include curly braces (`{}`) around variable names when referring to them in strings, for example, as follows:

```
source => "puppet:///modules/webserver/${brand}.conf",
```

Otherwise, Puppet's parser has to guess which characters should be a part of the variable name and which belong to the surrounding string. Curly braces make it explicit.

Parameters

Always end lines that declare parameters with a comma, even if it is the last parameter:

```
service { 'memcached':
  ensure => running,
  enable => true,
}
```

This is allowed by Puppet, and makes it easier if you want to add parameters later, or reorder the existing parameters.

When declaring a resource with a single parameter, make the declaration all on one line and with no trailing comma, as shown in the following snippet:

```
package { 'puppet': ensure => installed }
```

Where there is more than one parameter, give each parameter its own line:

```
package { 'rake':
  ensure   => installed,
  provider => gem,
  require  => Package['rubygems'],
}
```

To make the code easier to read, line up the parameter arrows in line with the longest parameter, as follows:

```
file { "/var/www/${app}/shared/config/rvmrc":
  owner   => 'deploy',
  group   => 'deploy',
  content => template('rails/rvmrc.erb'),
  require => File["/var/www/${app}/shared/config"],
}
```

The arrows should be aligned per resource, but not across the whole file, otherwise it can make it difficult for you to cut and paste code from one file to another.

Symlinks

When declaring file resources which are symlinks, use `ensure => link` and set the target attribute, as follows:

```
    file { '/etc/php5/cli/php.ini':
      ensure => link,
      target => '/etc/php.ini',
    }
```

Creating a manifest

If you already have some Puppet code (known as a Puppet manifest), you can skip this section and go on to the next. If not, we'll see how to create and apply a simple manifest.

How to do it...

To create and apply a simple manifest, follow these steps:

1. First, install Puppet locally on your machine or create a virtual machine and install Puppet on that machine. For YUM-based systems, use `https://yum.puppetlabs.com/` and for APT-based systems, use `https://apt.puppetlabs.com/`. You may also use gem to install Puppet. For our examples, we'll install Puppet using gem on a Debian Wheezy system (hostname: `cookbook`). To use gem, we need the `rubygems` package as follows:

```
t@cookbook:~$ sudo apt-get install rubygems
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  rubygems
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
Need to get 0 B/597 kB of archives.
After this operation, 3,844 kB of additional disk space will be
used.
```

```
Selecting previously unselected package rubygems.
(Reading database ... 30390 files and directories currently
installed.)
Unpacking rubygems (from .../rubygems_1.8.24-1_all.deb) ...
Processing triggers for man-db ...
Setting up rubygems (1.8.24-1) ...
```

2. Now, use `gem` to install Puppet:

```
t@cookbook $ sudo gem install puppet
Successfully installed hiera-1.3.4
Fetching: facter-2.3.0.gem (100%)
Successfully installed facter-2.3.0
Fetching: puppet-3.7.3.gem (100%)
Successfully installed puppet-3.7.3
Installing ri documentation for hiera-1.3.4
Installing ri documentation for facter-2.3.0
Installing ri documentation for puppet-3.7.3
Done installing documentation for hiera, facter, puppet after 239
seconds
```

3. Three gems are installed. Now, with Puppet installed, we can create a directory to contain our Puppet code:

```
t@cookbook:~$ mkdir -p .puppet/manifests
t@cookbook:~$ cd .puppet/manifests
t@cookbook:~/.puppet/manifests$
```

4. Within your `manifests` directory, create the `site.pp` file with the following content:

```
  node default {
    file { '/tmp/hello':
      content => "Hello, world!\n",
    }
  }
```

5. Test your manifest with the `puppet apply` command. This will tell Puppet to read the manifest, compare it to the state of the machine, and make any necessary changes to that state:

```
t@cookbook:~/.puppet/manifests$ puppet apply site.pp
Notice: Compiled catalog for cookbook in environment production in
0.14 seconds
Notice: /Stage[main]/Main/Node[default]/File[/tmp/hello]/ensure:
defined content as '{md5}746308829575e17c3331bbcb00c0898b'
Notice: Finished catalog run in 0.04 seconds
```

6. To see if Puppet did what we expected (create the `/tmp/hello` file with the `Hello, world`! content), run the following command:

```
t@cookbook:~/puppet/manifests$ cat /tmp/hello
Hello, world!
 t@cookbook:~/puppet/manifests$
```

Note that creating the file in `/tmp` did not require special permissions. We did not run Puppet via `sudo`. Puppet need not be run through `sudo`; there are cases where running via an unprivileged user can be useful.

There's more...

When several people are working on a code base, it's easy for style inconsistencies to creep in. Fortunately, there's a tool available which can automatically check your code for compliance with the style guide: `puppet-lint`. We'll see how to use this in the next section.

Checking your manifests with Puppet-lint

The puppetlabs official style guide outlines a number of style conventions for Puppet code, some of which we've touched on in the preceding section. For example, according to the style guide, manifests:

- Must use two-space soft tabs
- Must not use literal tab characters
- Must not contain trailing white space
- Should not exceed an 80 character line width
- Should align parameter arrows (`=>`) within blocks

Following the style guide will make sure that your Puppet code is easy to read and maintain, and if you're planning to release your code to the public, style compliance is essential.

The `puppet-lint` tool will automatically check your code against the style guide. The next section explains how to use it.

Getting ready

Here's what you need to do to install Puppet-lint:

1. We'll install Puppet-lint using the gem provider because the gem version is much more up to date than the APT or RPM packages available. Create a `puppet-lint.pp` manifest as shown in the following code snippet:

   ```
   package {'puppet-lint':
     ensure => 'installed',
     provider => 'gem',
   }
   ```

2. Run `puppet apply` on the `puppet-lint.pp` manifest, as shown in the following command:

   ```
   t@cookbook ~$ puppet apply puppet-lint.pp Notice: Compiled catalog
   for node1.example.com in environment production in 0.42 seconds
   Notice: /Stage[main]/Main/Package[puppet-lint]/ensure: created
   Notice: Finished catalog run in 2.96 seconds
   t@cookbook ~$ gem list puppet-lint *** LOCAL GEMS *** puppet-lint
   (1.0.1)
   ```

How to do it...

Follow these steps to use Puppet-lint:

1. Choose a Puppet manifest file that you want to check with Puppet-lint, and run the following command:

   ```
   t@cookbook ~$ puppet-lint puppet-lint.pp
   WARNING: indentation of => is not properly aligned on line 2
   ERROR: trailing whitespace found on line 4
   ```

2. As you can see, Puppet-lint found a number of problems with the manifest file. Correct the errors, save the file, and rerun Puppet-lint to check that all is well. If successful, you'll see no output:

   ```
   t@cookbook ~$ puppet-lint puppet-lint.pp
   t@cookbook ~$
   ```

There's more...

You can find out more about Puppet-lint at `https://github.com/rodjek/puppet-lint`.

Should you follow Puppet style guide and, by extension, keep your code lint-clean? It's up to you, but here are a couple of things to think about:

- It makes sense to use some style conventions, especially when you're working collaboratively on code. Unless you and your colleagues can agree on standards for whitespace, tabs, quoting, alignment, and so on, your code will be messy and difficult to read or maintain.
- If you're choosing a set of style conventions to follow, the logical choice would be that issued by puppetlabs and adopted by the community for use in public modules.

Having said that, it's possible to tell Puppet-lint to ignore certain checks if you've chosen not to adopt them in your codebase. For example, if you don't want Puppet-lint to warn you about code lines exceeding 80 characters, you can run Puppet-lint with the following option:

```
t@cookbook ~$ puppet-lint --no-80chars-check
```

Run `puppet-lint --help` to see the complete list of check configuration commands.

See also

- The *Automatic syntax checking with Git hooks* recipe in *Chapter 2*, *Puppet Infrastructure*
- The *Testing your Puppet manifests with rspec-puppet* recipe in *Chapter 9*, *External Tools and the Puppet Ecosystem*

Using modules

One of the most important things you can do to make your Puppet manifests clearer and more maintainable is to organize them into modules.

Modules are self-contained bundles of Puppet code that include all the files necessary to implement a thing. Modules may contain flat files, templates, Puppet manifests, custom fact declarations, augeas lenses, and custom Puppet types and providers.

Separating things into modules makes it easier to reuse and share code; it's also the most logical way to organize your manifests. In this example, we'll create a module to manage memcached, a memory caching system commonly used with web applications.

How to do it...

Following are the steps to create an example module:

1. We will use Puppet's module subcommand to create the directory structure for our new module:

```
t@cookbook:~$ mkdir -p .puppet/modules
t@cookbook:~$ cd .puppet/modules
t@cookbook:~/.puppet/modules$ puppet module generate thomas-
memcached
We need to create a metadata.json file for this module.  Please
answer the following questions; if the question is not applicable
to this module, feel free to leave it blank. Puppet uses Semantic
Versioning (semver.org) to version modules.What version is this
module?  [0.1.0]
--> Who wrote this module?  [thomas]
--> What license does this module code fall under?  [Apache 2.0]
--> How would you describe this module in a single sentence?
--> A module to install memcached Where is this module's source
code repository?
--> Where can others go to learn more about this module?
--> Where can others go to file issues about this module?
-->
----------------------------------------
{
  "name": "thomas-memcached",
  "version": "0.1.0",
  "author": "thomas",
  "summary": "A module to install memcached",
  "license": "Apache 2.0",
  "source": "",
  "issues_url": null,
  "project_page": null,
  "dependencies": [
    {
      "version_range": ">= 1.0.0",
      "name": "puppetlabs-stdlib"
    }
  ]
}
```

```
----------------------------------------
About to generate this metadata; continue? [n/Y]
--> y
Notice: Generating module at /home/thomas/.puppet/modules/thomas-
memcached...
Notice: Populating ERB templates...
Finished; module generated in thomas-memcached.
thomas-memcached/manifests
thomas-memcached/manifests/init.pp
thomas-memcached/spec
thomas-memcached/spec/classes
thomas-memcached/spec/classes/init_spec.rb
thomas-memcached/spec/spec_helper.rb
thomas-memcached/README.md
thomas-memcached/metadata.json
thomas-memcached/Rakefile
thomas-memcached/tests
thomas-memcached/tests/init.pp
```

This command creates the module directory and creates some empty files as starting points. To use the module, we'll create a symlink to the module name (memcached).

```
t@cookbook:~/.puppet/modules$ ln -s thomas-memcached memcached
```

2. Now, edit `memcached/manifests/init.pp` and change the class definition at the end of the file to the following. Note that `puppet module generate` created many lines of comments; in a production module you would want to edit those default comments:

```
class memcached {
  package { 'memcached':
    ensure => installed,
  }

  file { '/etc/memcached.conf':
    source  => 'puppet:///modules/memcached/memcached.conf',
    owner   => 'root',
    group   => 'root',
    mode    => '0644',
    require => Package['memcached'],
  }
```

```
    service { 'memcached':
      ensure  => running,
      enable  => true,
      require => [Package['memcached'],
                  File['/etc/memcached.conf']],
    }
  }
```

3. Create the `modules/thomas-memcached/files` directory and then create a file named `memcached.conf` with the following contents:

   ```
   -m 64
   -p 11211
   -u nobody
   -l 127.0.0.1
   ```

4. Change your `site.pp` file to the following:

   ```
   node default {
     include memcached
   }
   ```

5. We would like this module to install memcached. We'll need to run Puppet with root privileges, and we'll use sudo for that. We'll need Puppet to be able to find the module in our home directory; we can specify this on the command line when we run Puppet as shown in the following code snippet:

   ```
   t@cookbook:~$ sudo puppet apply --modulepath=/home/thomas/.puppet/
   modules /home/thomas/.puppet/manifests/site.pp
   Notice: Compiled catalog for cookbook.example.com in environment
   production in 0.33 seconds
   Notice: /Stage[main]/Memcached/File[/etc/memcached.conf]/content:
   content changed '{md5}a977521922a151c959ac953712840803' to '{md5}9
   429eff3e3354c0be232a020bcf78f75'
   Notice: Finished catalog run in 0.11 seconds
   ```

6. Check whether the new service is running:

   ```
   t@cookbook:~$ sudo service memcached status
   [ ok ] memcached is running.
   ```

How it works...

When we created the module using Puppet's module generate command, we used the name `thomas-memcached`. The name before the hyphen is your username or your username on Puppet forge (an online repository of modules). Since we want Puppet to be able to find the module by the name `memcached`, we make a symbolic link between `thomas-memcached` and `memcached`.

Modules have a specific directory structure. Not all of these directories need to be present, but if they are, this is how they should be organized:

```
modules/
  └MODULE_NAME/               never use a dash (-) in a module name
     └examples/               example usage of the module
     └files/                  flat files used by the module
     └lib/
        └facter/              define new facts for facter
        └puppet/
           └parser/
              └functions/     define a new puppet function, like
sort()
           └provider/    define a provider for a new or existing type
           └util/        define helper functions (in ruby)
           └type/        define a new type in puppet
     └manifests/
        └init.pp         class MODULE_NAME { }
     └spec/ rSpec        tests
     └templates/         erb template files used by the module
```

All manifest files (those containing Puppet code) live in the manifests directory. In our example, the `memcached` class is defined in the `manifests/init.pp` file, which will be imported automatically.

Inside the `memcached` class, we refer to the `memcached.conf` file:

```
file { '/etc/memcached.conf':
  source => 'puppet:///modules/memcached/memcached.conf',
}
```

The preceding `source` parameter tells Puppet to look for the file in:

```
MODULEPATH/   (/home/thomas/.puppet/modules)
  └memcached/
    └files/
      └memcached.conf
```

There's more...

Learn to love modules because they'll make your Puppet life a lot easier. They're not complicated, however, practice and experience will help you judge when things should be grouped into modules, and how best to arrange your module structure. Modules can hold more than manifests and files as we'll see in the next two sections.

Templates

If you need to use a template as a part of the module, place it in the module's templates directory and refer to it as follows:

```
file { '/etc/memcached.conf':
  content => template('memcached/memcached.conf.erb'),
}
```

Puppet will look for the file in:

```
MODULEPATH/memcached/templates/memcached.conf.erb
```

Facts, functions, types, and providers

Modules can also contain custom facts, custom functions, custom types, and providers.

For more information about these, refer to *Chapter 9*, *External Tools and the Puppet Ecosystem*.

Third-party modules

You can download modules provided by other people and use them in your own manifests just like the modules you create. For more on this, see Using Public Modules recipe in *Chapter 7*, *Managing Applications*.

Module organization

For more details on how to organize your modules, see puppetlabs website:

`http://docs.puppetlabs.com/puppet/3/reference/modules_fundamentals.html`

See also

- The *Creating custom facts* recipe in *Chapter 9*, *External Tools and the Puppet Ecosystem*
- The *Using public modules* recipe in *Chapter 7*, *Managing Applications*
- The *Creating your own resource types* recipe in *Chapter 9*, *External Tools and the Puppet Ecosystem*
- The *Creating your own providers* recipe in *Chapter 9*, *External Tools and the Puppet Ecosystem*

Using standard naming conventions

Choosing appropriate and informative names for your modules and classes will be a big help when it comes to maintaining your code. This is even truer if other people need to read and work on your manifests.

How to do it...

Here are some tips on how to name things in your manifests:

1. Name modules after the software or service they manage, for example, `apache` or `haproxy`.
2. Name classes within modules (subclasses) after the function or service they provide to the module, for example, `apache::vhosts` or `rails::dependencies`.
3. If a class within a module disables the service provided by that module, name it `disabled`. For example, a class that disables Apache should be named `apache::disabled`.
4. Create a roles and profiles hierarchy of modules. Each node should have a single role consisting of one or more profiles. Each profile module should configure a single service.
5. The module that manages users should be named `user`.
6. Within the user module, declare your virtual users within the class `user::virtual` (for more on virtual users and other resources, see the *Using virtual resources* recipe in *Chapter 5*, *Users and Virtual Resources*).
7. Within the user module, subclasses for particular groups of users should be named after the group, for example, `user::sysadmins` or `user::contractors`.
8. When using Puppet to deploy the config files for different services, name the file after the service, but with a suffix indicating what kind of file it is, for example:
   - Apache init script: `apache.init`
   - Logrotate config snippet for Rails: `rails.logrotate`
   - Nginx vhost file for mywizzoapp: `mywizzoapp.vhost.nginx`
   - MySQL config for standalone server: `standalone.mysql`
9. If you need to deploy a different version of a file depending on the operating system release, for example, you can use a naming convention like the following:
   ```
   memcached.lucid.conf
   memcached.precise.conf
   ```
10. You can have Puppet automatically select the appropriate version as follows:
    ```
    source = > "puppet:///modules/memcached
      /memcached.${::lsbdistrelease}.conf",
    ```
11. If you need to manage, for example, different Ruby versions, name the class after the version it is responsible for, for example, `ruby192` or `ruby186`.

There's more...

Puppet community maintains a set of best practice guidelines for your Puppet infrastructure, which includes some hints on naming conventions:

`http://docs.puppetlabs.com/guides/best_practices.html`

Some people prefer to include multiple classes on a node by using a comma-separated list, rather than separate `include` statements, for example:

```
node 'server014' inherits 'server' {
  include mail::server, repo::gem, repo::apt, zabbix
}
```

This is a matter of style, but I prefer to use separate `include` statements, one on a line, because it makes it easier to copy and move around class inclusions between nodes without having to tidy up the commas and indentation every time.

I mentioned inheritance in a couple of the preceding examples; if you're not sure what this is, don't worry, I'll explain this in detail in the next chapter.

Using inline templates

Templates are a powerful way of using **Embedded Ruby** (**ERB**) to help build config files dynamically. You can also use ERB syntax directly without having to use a separate file by calling the `inline_template` function. ERB allows you to use conditional logic, iterate over arrays, and include variables.

How to do it...

Here's an example of how to use `inline_template`:

Pass your Ruby code to `inline_template` within Puppet manifest, as follows:

```
cron { 'chkrootkit':
  command => '/usr/sbin/chkrootkit >
    /var/log/chkrootkit.log 2>&1',
  hour    => inline_template('<%= @hostname.sum % 24 %>'),
  minute  => '00',
}
```

How it works...

Anything inside the string passed to `inline_template` is executed as if it were an ERB template. That is, anything inside the `<%=` and `%>` delimiters will be executed as Ruby code, and the rest will be treated as a string.

In this example, we use `inline_template` to compute a different hour for this cron resource (a scheduled job) for each machine, so that the same job does not run at the same time on all machines. For more on this technique, see the *Distributing cron jobs efficiently* recipe in *Chapter 6*, *Managing Resources and Files*.

There's more...

In ERB code, whether inside a template file or an `inline_template` string, you can access your Puppet variables directly by name using an `@` prefix, if they are in the current scope or the top scope (facts):

```
<%= @fqdn %>
```

To reference variables in another scope, use `scope.lookupvar`, as follows:

```
<%= "The value of something from otherclass is " +
  scope.lookupvar('otherclass::something') %>
```

You should use inline templates sparingly. If you really need to use some complicated logic in your manifest, consider using a custom function instead (see the *Creating custom functions* recipe in *Chapter 9*, *External Tools and the Puppet Ecosystem*).

See also

- The *Using ERB templates* recipe in *Chapter 4*, *Working with Files and Packages*
- The *Using array iteration in templates* recipe in *Chapter 4*, *Working with Files and Packages*

Iterating over multiple items

Arrays are a powerful feature in Puppet; wherever you want to perform the same operation on a list of things, an array may be able to help. You can create an array just by putting its content in square brackets:

```
$lunch = [ 'franks', 'beans', 'mustard' ]
```

How to do it...

Here's a common example of how arrays are used:

1. Add the following code to your manifest:

   ```
   $packages = [ 'ruby1.8-dev',
     'ruby1.8',
     'ri1.8',
     'rdoc1.8',
     'irb1.8',
     'libreadline-ruby1.8',
     'libruby1.8',
     'libopenssl-ruby' ]

   package { $packages: ensure => installed }
   ```

2. Run Puppet and note that each package should now be installed.

How it works...

Where Puppet encounters an array as the name of a resource, it creates a resource for each element in the array. In the example, a new package resource is created for each of the packages in the `$packages` array, with the same parameters (`ensure => installed`). This is a very compact way to instantiate many similar resources.

There's more...

Although arrays will take you a long way with Puppet, it's also useful to know about an even more flexible data structure: the hash.

Using hashes

A hash is like an array, but each of the elements can be stored and looked up by name (referred to as the key), for example (`hash.pp`):

```
$interface = {
  'name'  => 'eth0',
  'ip'    => '192.168.0.1',
  'mac'   => '52:54:00:4a:60:07'
}
notify { "(${interface['ip']}) at ${interface['mac']} on
  ${interface['name']}": }
```

When we run Puppet on this, we see the following notify in the output:

```
t@cookbook:~/.puppet/manifests$ puppet apply hash.pp
Notice: (192.168.0.1) at 52:54:00:4a:60:07 on etho
```

Hash values can be anything that you can assign to variables, strings, function calls, expressions, and even other hashes or arrays. Hashes are useful to store a bunch of information about a particular thing because by accessing each element of the hash using a key, we can quickly find the information for which we are looking.

Creating arrays with the split function

You can declare literal arrays using square brackets, as follows:

```
define lunchprint() {
  notify { "Lunch included ${name}":}": }
}

$lunch = ['egg', 'beans', 'chips']
lunchprint { $lunch: }
```

Now, when we run Puppet on the preceding code, we see the following notice messages in the output:

```
t@mylaptop ~ $ puppet apply lunchprint.pp
...
Notice: Lunch included chips
Notice: Lunch included beans
Notice: Lunch included egg
```

However, Puppet can also create arrays for you from strings, using the `split` function, as follows:

```
$menu = 'egg beans chips'
$items = split($menu, ' ')
lunchprint { $items: }
```

Running `puppet apply` against this new manifest, we see the same messages in the output:

```
t@mylaptop ~ $ puppet apply lunchprint2.pp
...
Notice: Lunch included chips
Notice: Lunch included beans
Notice: Lunch included egg.
```

Note that `split` takes two arguments: the first argument is the string to be split. The second argument is the character to split on; in this example, a single space. As Puppet works its way through the string, when it encounters a space, it will interpret it as the end of one item and the beginning of the next. So, given the string '`egg beans chips'`, this will be split into three items.

The character to split on can be any character or string:

```
$menu = 'egg and beans and chips'
$items = split($menu, ' and ')
```

The character can also be a regular expression, for example, a set of alternatives separated by a | (pipe) character:

```
$lunch = 'egg:beans,chips'
$items = split($lunch, ':|,')
```

Writing powerful conditional statements

Puppet's `if` statement allows you to change the manifest behavior based on the value of a variable or an expression. With it, you can apply different resources or parameter values depending on certain facts about the node, for example, the operating system, or the memory size.

You can also set variables within the manifest, which can change the behavior of included classes. For example, nodes in data center A might need to use different DNS servers than nodes in data center B, or you might need to include one set of classes for an Ubuntu system, and a different set for other systems.

How to do it...

Here's an example of a useful conditional statement. Add the following code to your manifest:

```
if $::timezone == 'UTC' {
  notify { 'Universal Time Coordinated':}
} else {
  notify { "$::timezone is not UTC": }
}
```

How it works...

Puppet treats whatever follows an `if` keyword as an expression and evaluates it. If the expression evaluates to true, Puppet will execute the code within the curly braces.

Optionally, you can add an else branch, which will be executed if the expression evaluates to false.

There's more...

Here are some more tips on using `if` statements.

Elsif branches

You can add further tests using the `elsif` keyword, as follows:

```
if $::timezone == 'UTC' {
  notify { 'Universal Time Coordinated': }
} elsif $::timezone == 'GMT' {
  notify { 'Greenwich Mean Time': }
} else {
  notify { "$::timezone is not UTC": }
}
```

Comparisons

You can check whether two values are equal using the `==` syntax, as in our example:

```
if $::timezone == 'UTC' {

}
```

Alternatively, you can check whether they are not equal using `!=`:

```
if $::timezone != 'UTC' {
  …
}
```

You can also compare numeric values using `<` and `>`:

```
if $::uptime_days > 365 {
  notify { 'Time to upgrade your kernel!': }
}
```

To test whether a value is greater (or less) than or equal to another value, use `<=` or `>=`:

```
if $::mtu_eth0 <= 1500 {
  notify {"Not Jumbo Frames": }
}
```

Combining expressions

You can put together the kind of simple expressions described previously into more complex logical expressions, using `and`, `or`, and `not`:

```
if ($::uptime_days > 365) and ($::kernel == 'Linux') {
  …
}

if ($role == 'webserver') and ( ($datacenter == 'A') or ($datacenter
== 'B') ) {
  …
}
```

See also

- The *Using the in operator* recipe in this chapter
- The *Using selectors and case statements* recipe in this chapter

Using regular expressions in if statements

Another kind of expression you can test in `if` statements and other conditionals is the regular expression. A regular expression is a powerful way to compare strings using pattern matching.

How to do it...

This is one example of using a regular expression in a conditional statement. Add the following to your manifest:

```
if $::architecture =~ /64/ {
  notify { '64Bit OS Installed': }
} else {
  notify { 'Upgrade to 64Bit': }
  fail('Not 64 Bit')
}
```

How it works...

Puppet treats the text supplied between the forward slashes as a regular expression, specifying the text to be matched. If the match succeeds, the `if` expression will be true and so the code between the first set of curly braces will be executed. In this example, we used a regular expression because different distributions have different ideas on what to call `64bit`; some use `amd64`, while others use `x86_64`. The only thing we can count on is the presence of the number 64 within the fact. Some facts that have version numbers in them are treated as strings to Puppet. For instance, `$::facterversion`. On my test system, this is `2.0.1`, but when I try to compare that with `2`, Puppet fails to make the comparison:

```
Error: comparison of String with 2 failed at /home/thomas/.puppet/
manifests/version.pp:1 on node cookbook.example.com
```

If you wanted instead to do something if the text does not match, use `!~` rather than `=~`:

```
if $::kernel !~ /Linux/ {
  notify { 'Not Linux, could be Windows, MacOS X, AIX, or ?': }
}
```

There's more...

Regular expressions are very powerful, but can be difficult to understand and debug. If you find yourself using a regular expression so complex that you can't see at a glance what it does, think about simplifying your design to make it easier. However, one particularly useful feature of regular expressions is the ability to capture patterns.

Capturing patterns

You can not only match text using a regular expression, but also capture the matched text and store it in a variable:

```
$input = 'Puppet is better than manual configuration'
if $input =~ /(.*) is better than (.*)/ {
  notify { "You said '${0}'. Looks like you're comparing ${1}
    to ${2}!": }
}
```

The preceding code produces this output:

You said 'Puppet is better than manual configuration'. Looks like you're comparing Puppet to manual configuration!

The variable `$0` stores the whole matched text (assuming the overall match succeeded). If you put brackets around any part of the regular expression, it creates a group, and any matched groups will also be stored in variables. The first matched group will be `$1`, the second `$2`, and so on, as shown in the preceding example.

Regular expression syntax

Puppet's regular expression syntax is the same as Ruby's, so resources that explain Ruby's regular expression syntax will also help you with Puppet. You can find a good introduction to Ruby's regular expression syntax at this website:

`http://www.tutorialspoint.com/ruby/ruby_regular_expressions.htm`.

See also

- Refer to the *Using regular expression substitutions* recipe in this chapter

Using selectors and case statements

Although you could write any conditional statement using `if`, Puppet provides a couple of extra forms to help you express conditionals more easily: the selector and the `case` statement.

How to do it...

Here are some examples of selector and `case` statements:

1. Add the following code to your manifest:

```
$systemtype = $::operatingsystem ? {
  'Ubuntu' => 'debianlike',
  'Debian' => 'debianlike',
  'RedHat' => 'redhatlike',
  'Fedora' => 'redhatlike',
  'CentOS' => 'redhatlike',
  default  => 'unknown',
}

notify { "You have a ${systemtype} system": }
```

2. Add the following code to your manifest:

```
class debianlike {
  notify { 'Special manifest for Debian-like systems': }
}

class redhatlike {
  notify { 'Special manifest for RedHat-like systems': }
}

case $::operatingsystem {
  'Ubuntu',
  'Debian': {
    include debianlike
  }
  'RedHat',
  'Fedora',
  'CentOS',
  'Springdale': {
    include redhatlike
  }
  default: {
    notify { "I don't know what kind of system you have!":
    }
  }
}
```

How it works...

Our example demonstrates both the selector and the `case` statement, so let's see in detail how each of them works.

Selector

In the first example, we used a selector (the `?` operator) to choose a value for the `$systemtype` variable depending on the value of `$::operatingsystem`. This is similar to the ternary operator in C or Ruby, but instead of choosing between two possible values, you can have as many values as you like.

Puppet will compare the value of `$::operatingsystem` to each of the possible values we have supplied in Ubuntu, Debian, and so on. These values could be regular expressions (for example, for a partial string match, or to use wildcards), but in our case, we have just used literal strings.

As soon as it finds a match, the selector expression returns whatever value is associated with the matching string. If the value of `$::operatingsystem` is Fedora, for example, the selector expression will return the `redhatlike` string and this will be assigned to the variable `$systemtype`.

Case statement

Unlike selectors, the `case` statement does not return a value. `case` statements come in handy when you want to execute different code depending on the value of some expression. In our second example, we used the `case` statement to include either the `debianlike` or `redhatlike` class, depending on the value of `$::operatingsystem`.

Again, Puppet compares the value of `$::operatingsystem` to a list of potential matches. These could be regular expressions or strings, or as in our example, comma-separated lists of strings. When it finds a match, the associated code between curly braces is executed. So, if the value of `$::operatingsystem` is `Ubuntu`, then the code including `debianlike` will be executed.

There's more...

Once you've got a grip of the basic use of selectors and `case` statements, you may find the following tips useful.

Regular expressions

As with `if` statements, you can use regular expressions with selectors and `case` statements, and you can also capture the values of the matched groups and refer to them using `$1`, `$2`, and so on:

```
case $::lsbdistdescription {
  /Ubuntu (.+)/: {
    notify { "You have Ubuntu version ${1}": }
  }
  /CentOS (.+)/: {
    notify { "You have CentOS version ${1}": }
  }
  default: {}
}
```

Defaults

Both selectors and `case` statements let you specify a default value, which is chosen if none of the other options match (the style guide suggests you always have a default clause defined):

```
$lunch = 'Filet mignon.'
$lunchtype =  $lunch ? {
  /fries/ => 'unhealthy',
  /salad/ => 'healthy',
  default => 'unknown',
}

notify { "Your lunch was ${lunchtype}": }
```

The output is as follows:

```
t@mylaptop ~ $ puppet apply lunchtype.pp
Notice: Your lunch was unknown
```

```
Notice: /Stage[main]/Main/Notify[Your lunch was unknown]/message: defined
'message' as 'Your lunch was unknown'
```

When the default action shouldn't normally occur, use the `fail()` function to halt the Puppet run.

Using the in operator

The `in` operator tests whether one string contains another string. Here's an example:

```
if 'spring' in 'springfield'
```

The preceding expression is true if the `spring` string is a substring of `springfield`, which it is. The `in` operator can also test for membership of arrays as follows:

```
if $crewmember in ['Frank', 'Dave', 'HAL' ]
```

When `in` is used with a hash, it tests whether the string is a key of the hash:

```
$ifaces = { 'lo'   => '127.0.0.1',
            'eth0' => '192.168.0.1' }
if 'eth0' in $ifaces {
  notify { "eth0 has address ${ifaces['eth0']}": }
}
```

How to do it...

The following steps will show you how to use the `in` operator:

1. Add the following code to your manifest:

```
if $::operatingsystem in [ 'Ubuntu', 'Debian' ] {
  notify { 'Debian-type operating system detected': }
} elsif $::operatingsystem in [ 'RedHat', 'Fedora', 'SuSE',
  'CentOS' ] {
  notify { 'RedHat-type operating system detected': }
} else {
  notify { 'Some other operating system detected': }
}
```

2. Run Puppet:

```
t@cookbook:~/.puppet/manifests$ puppet apply in.pp
Notice: Compiled catalog for cookbook.example.com in environment
production in 0.03 seconds
Notice: Debian-type operating system detected
```

```
Notice: /Stage[main]/Main/Notify[Debian-type operating system
detected]/message: defined 'message' as 'Debian-type operating
system detected'
Notice: Finished catalog run in 0.02 seconds
```

There's more...

The value of an `in` expression is Boolean (true or false) so you can assign it to a variable:

```
$debianlike = $::operatingsystem in [ 'Debian', 'Ubuntu' ]

if $debianlike {
  notify { 'You are in a maze of twisty little packages, all alike': }
}
```

Using regular expression substitutions

Puppet's `regsubst` function provides an easy way to manipulate text, search and replace expressions within strings, or extract patterns from strings. We often need to do this with data obtained from a fact, for example, or from external programs.

In this example, we'll see how to use `regsubst` to extract the first three octets of an IPv4 address (the network part, assuming it's a `/24` class C address).

How to do it...

Follow these steps to build the example:

1. Add the following code to your manifest:

   ```
   $class_c = regsubst($::ipaddress, '(.*)\..*', '\1.0')
   notify { "The network part of ${::ipaddress} is ${class_c}": }
   ```

2. Run Puppet:

   ```
   t@cookbook:~/.puppet/manifests$ puppet apply ipaddress.pp
   Notice: Compiled catalog for cookbook.example.com in environment
   production in 0.02 seconds
   Notice: The network part of 192.168.122.148 is
     192.168.122.0
   Notice: /Stage[main]/Main/Notify[The network part of
   192.168.122.148 is
     192.168.122.0]/message: defined 'message' as 'The network part
   of 192.168.122.148 is
   ```

```
    192.168.122.0'
Notice: Finished catalog run in 0.03 seconds
```

How it works...

The `regsubst` function takes at least three parameters: source, pattern, and replacement. In our example, we specified the source string as `$::ipaddress`, which, on this machine, is as follows:

```
192.168.122.148
```

We specify the `pattern` function as follows:

```
(.*)\..*
```

We specify the `replacement` function as follows:

```
\1.0
```

The pattern captures all of the string up to the last period (`\.`) in the `\1` variable. We then match on `.*`, which matches everything to the end of the string, so when we replace the string at the end with `\1.0`, we end up with only the network portion of the IP address, which evaluates to the following:

```
192.168.122.0
```

We could have got the same result in other ways, of course, including the following:

```
$class_c = regsubst($::ipaddress, '\.\d+$', '.0')
```

Here, we only match the last octet and replace it with `.0`, which achieves the same result without capturing.

There's more...

The `pattern` function can be any regular expression, using the same (Ruby) syntax as regular expressions in `if` statements.

See also

- The *Importing dynamic information* recipe in *Chapter 3*, *Writing Better Manifests*
- The *Getting information about the environment* recipe in *Chapter 3*, *Writing Better Manifests*
- The *Using regular expressions in if statements* recipe in this chapter

Using the future parser

Puppet language is evolving at the moment; many features that are expected to be included in the next major release (4) are available if you enable the future parser.

Getting ready

- Ensure that the `rgen` gem is installed.
- Set `parser = future` in the `[main]` section of your `puppet.conf(/etc/puppet/puppet.conf` for open source Puppet as `root,/etc/puppetlabs/puppet/puppet.conf` for Puppet `Enterprise, and~/.puppet/puppet.conf` for a non-root user running puppet).
- To temporarily test with the future parser, use `--parser=future` on the command line.

How to do it...

Many of the experimental features deal with how code is evaluated, for example, in an earlier example we compared the value of the `$::facterversion` fact with a number, but the value is treated as a string so the code fails to compile. Using the future parser, the value is converted and no error is reported as shown in the following command line output:

```
t@cookbook:~/.puppet/manifests$ puppet apply --parser=future version.pp
Notice: Compiled catalog for cookbook.example.com in environment
production in 0.36 seconds
Notice: Finished catalog run in 0.03 seconds
```

Appending to and concatenating arrays

You can concatenate arrays with the + operator or append them with the << operator. In the following example, we use the ternary operator to assign a specific package name to the `$apache` variable. We then append that value to an array using the << operator:

```
$apache = $::osfamily ? {
  'Debian' => 'apache2',
  'RedHat' => 'httpd'
}
$packages = ['memcached'] << $apache
package {$packages: ensure => installed}
```

If we have two arrays, we can use the + operator to concatenate the two arrays. In this example, we define an array of system administrators (`$sysadmins`) and another array of application owners (`$appowners`). We can then concatenate the array and use it as an argument to our allowed users:

```
$sysadmins = [ 'thomas','john','josko' ]
$appowners = [ 'mike', 'patty', 'erin' ]
$users = $sysadmins + $appowners
notice ($users)
```

When we apply this manifest, we see that the two arrays have been joined as shown in the following command line output:

```
t@cookbook:~/.puppet/manifests$ puppet apply --parser=future concat.pp
Notice: [thomas, john, josko, mike, patty, erin]
Notice: Compiled catalog for cookbook.example.com in environment
production in 0.36 seconds
Notice: Finished catalog run in 0.03 seconds
```

Merging Hashes

If we have two hashes, we can merge them using the same + operator we used for arrays. Consider our `$interfaces` hash from a previous example; we can add another interface to the hash:

```
$iface = {
  'name' => 'eth0',
  'ip'   => '192.168.0.1',
  'mac'  => '52:54:00:4a:60:07'
}  + {'route' => '192.168.0.254'}
notice ($iface)
```

When we apply this manifest, we see that the route attribute has been merged into the hash (your results may differ, the order in which the hash prints is unpredictable), as follows:

```
t@cookbook:~/.puppet/manifests$ puppet apply --parser=future hash2.pp
Notice: {route => 192.168.0.254, name => eth0, ip => 192.168.0.1, mac =>
52:54:00:4a:60:07}
Notice: Compiled catalog for cookbook.example.com in environment
production in 0.36 seconds
Notice: Finished catalog run in 0.03 seconds
```

Lambda functions

Lambda functions are iterators applied to arrays or hashes. You iterate through the array or hash and apply an iterator function such as `each`, `map`, `filter`, `reduce`, or `slice` to each element of the array or key of the hash. Some of the lambda functions return a calculated array or value; others such as `each` only return the input array or hash.

Lambda functions such as `map` and `reduce` use temporary variables that are thrown away after the lambda has finished. Use of lambda functions is something best shown by example. In the next few sections, we will show an example usage of each of the lambda functions.

Reduce

Reduce is used to reduce the array to a single value. This can be used to calculate the maximum or minimum of the array, or in this case, the sum of the elements of the array:

```
$count = [1,2,3,4,5]
$sum = reduce($count) | $total, $i | { $total + $i }
notice("Sum is $sum")
```

This preceding code will compute the sum of the `$count` array and store it in the `$sum` variable, as follows:

```
t@cookbook:~/.puppet/manifests$ puppet apply --parser future lambda.pp
Notice: Sum is 15
Notice: Compiled catalog for cookbook.example.com in environment
production in 0.36 seconds
Notice: Finished catalog run in 0.03 seconds
```

Filter

Filter is used to filter the array or hash based upon a test within the lambda function. For instance to filter our `$count` array as follows:

```
$filter = filter ($count) | $i | { $i > 3 }
notice("Filtered array is $filter")
```

When we apply this manifest, we see that only elements 4 and 5 are in the result:

```
Notice: Filtered array is [4, 5]
```

Map

Map is used to apply a function to each element of the array. For instance, if we wanted (for some unknown reason) to compute the square of all the elements of the array, we would use `map` as follows:

```
$map = map ($count) | $i | { $i * $i }
notice("Square of array is $map")
```

The result of applying this manifest is a new array with every element of the original array squared (multiplied by itself), as shown in the following command line output:

```
Notice: Square of array is [1, 4, 9, 16, 25]
```

Slice

Slice is useful when you have related values stored in the same array in a sequential order. For instance, if we had the destination and port information for a firewall in an array, we could split them up into pairs and perform operations on those pairs:

```
$firewall_rules = ['192.168.0.1','80','192.168.0.10','443']
slice ($firewall_rules,2) |$ip, $port| { notice("Allow $ip on
  $port") }
```

When applied, this manifest will produce the following notices:

```
Notice: Allow 192.168.0.1 on 80
Notice: Allow 192.168.0.10 on 443
```

To make this a useful example, create a new firewall resource within the block of the slice instead of notice:

```
slice ($firewall_rules,2) |$ip, $port| {
  firewall {"$port from $ip":
    dport  => $port,
    source => "$ip",
    action => 'accept',
  }
}
```

Each

Each is used to iterate over the elements of the array but lacks the ability to capture the results like the other functions. Each is the simplest case where you simply wish to do something with each element of the array, as shown in the following code snippet:

```
each ($count) |$c| { notice($c) }
```

As expected, this executes the `notice` for each element of the `$count` array, as follows:

```
Notice: 1
Notice: 2
Notice: 3
Notice: 4
Notice: 5
```

Other features

There are other new features of Puppet language available when using the future parser. Some increase readability or compactness of code. For more information, refer to the documentation on puppetlabs website at `http://docs.puppetlabs.com/puppet/latest/reference/experiments_future.html`.

2
Puppet Infrastructure

"Computers in the future may have as few as 1,000 vacuum tubes and weigh only 1.5 tons."

— Popular Mechanics, 1949

In this chapter, we will cover:

- Installing Puppet
- Managing your manifests with Git
- Creating a decentralized Puppet architecture
- Writing a papply script
- Running Puppet from cron
- Bootstrapping Puppet with bash
- Creating a centralized Puppet infrastructure
- Creating certificates with multiple DNS names
- Running Puppet from passenger
- Setting up the environment
- Configuring PuppetDB
- Configuring Hiera
- Setting-node specific data with Hiera
- Storing secret data with hiera-gpg
- Using MessagePack serialization
- Automatic syntax checking with Git hooks
- Pushing code around with Git
- Managing environments with Git

Introduction

In this chapter, we will cover how to deploy Puppet in a centralized and decentralized manner. With each approach, we'll see a combination of best practices, my personal experience, and community solutions.

We'll configure and use both PuppetDB and Hiera. PuppetDB is used with exported resources, which we will cover in *Chapter 5*, *Users and Virtual Resources*. Hiera is used to separate variable data from Puppet code.

Finally, I'll introduce Git and see how to use Git to organize our code and our infrastructure.

Because Linux distributions, such as Ubuntu, Red Hat, and CentOS, differ in the specific details of package names, configuration file paths, and many other things, I have decided that for reasons of space and clarity the best approach for this book is to pick one distribution (*Debian 7* named as *Wheezy*) and stick to that. However, Puppet runs on most popular operating systems, so you should have very little trouble adapting the recipes to your own favorite OS and distribution.

At the time of writing, Puppet 3.7.2 is the latest stable version available, this is the version of Puppet used in the book. The syntax of Puppet commands changes often, so be aware that while older versions of Puppet are still perfectly usable, they may not support all of the features and syntax described in this book. As we saw in *Chapter 1*, *Puppet Language and Style*, the future parser showcases features of the language scheduled to become default in Version 4 of Puppet.

Installing Puppet

In *Chapter 1*, *Puppet Language and Style*, we installed Puppet as a rubygem using the gem install. When deploying to several nodes, this may not be the best approach. Using the package manager of your chosen distribution is the best way to keep your Puppet versions similar on all of the nodes in your deployment. Puppet labs maintain repositories for APT-based and YUM-based distributions.

Getting ready

If your Linux distribution uses APT for package management, go to `http://apt.puppetlabs.com/` and download the appropriate Puppet labs release package for your distribution. For our wheezy cookbook node, we will use `http://apt.puppetlabs.com/puppetlabs-release-wheezy.deb`.

If you are using a Linux distribution that uses YUM for package management, go to `http://yum.puppetlabs.com/` and download the appropriate Puppet labs release package for your distribution.

How to do it...

1. Once you have found the appropriate Puppet labs release package for your distribution, the steps to install Puppet are the same for either APT or YUM:
    - Install Puppet labs release package
    - Install Puppet package
2. Once you have installed Puppet, verify the version of Puppet as shown in the following example:

```
t@ckbk:~ puppet --version 3.7.2
```

Now that we have a method to install Puppet on our nodes, we need to turn our attention to keeping our Puppet manifests organized. In the next section, we will see how to use Git to keep our code organized and consistent.

Managing your manifests with Git

It's a great idea to put your Puppet manifests in a version control system such as Git or Subversion (Git is the de facto standard for Puppet). This gives you several advantages:

- You can undo changes and revert to any previous version of your manifest
- You can experiment with new features using a branch
- If several people need to make changes to the manifests, they can make them independently, in their own working copies, and then merge their changes later
- You can use the `git log` feature to see what was changed, and when (and by whom)

Getting ready

In this section, we'll import your existing manifest files into Git. If you have created a Puppet directory in a previous section use that, otherwise, use your existing manifest directory.

In this example, we'll create a new Git repository on a server accessible from all our nodes. There are several steps we need to take to have our code held in a Git repository:

1. Install Git on a central server.
2. Create a user to run Git and own the repository.
3. Create a repository to hold the code.
4. Create **SSH** keys to allow key-based access to the repository.
5. Install Git on a node and download the latest version from our Git repository.

How to do it...

Follow these steps:

1. First, install Git on your Git server (`git.example.com` in our example). The easiest way to do this is using Puppet. Create the following manifest, call it `git.pp`:

   ```
     package {'git':
       ensure => installed
     }
   ```

2. Apply this manifest using `puppet apply git.pp`, this will install Git.

3. Next, create a Git user that the nodes will use to log in and retrieve the latest code. Again, we'll do this with puppet. We'll also create a directory to hold our repository (`/home/git/repos`) as shown in the following code snippet:

   ```
   group { 'git':
     gid => 1111,
   }
   user {'git':
     uid => 1111,
     gid => 1111,
     comment => 'Git User',
     home => '/home/git',
     require => Group['git'],
   }
   file {'/home/git':
     ensure => 'directory',
     owner => 1111,
     group => 1111,
     require => User['git'],
   }
   file {'/home/git/repos':
     ensure => 'directory',
     owner => 1111,
     group => 1111,
     require => File['/home/git']
   }
   ```

4. After applying that manifest, log in as the Git user and create an empty Git repository using the following command:

   ```
   # sudo -iu git
   git@git $ cd repos
   git@git $ git init --bare puppet.git
   Initialized empty Git repository in /home/git/repos/puppet.git/
   ```

5. Set a password for the Git user, we'll need to log in remotely after the next step:

   ```
   [root@git ~]# passwd git
   Changing password for user git.
   New password:
   Retype new password:
   passwd: all authentication tokens updated successfully.
   ```

6. Now back on your local machine, create an `ssh` key for our nodes to use to update the repository:

   ```
   t@mylaptop ~ $ cd .ssh
   t@mylaptop ~/.ssh $ ssh-keygen -b 4096 -f git_rsa
   Generating public/private rsa key pair.
   Enter passphrase (empty for no passphrase):
   Enter same passphrase again:
   Your identification has been saved in git_rsa.
   Your public key has been saved in git_rsa.pub.
   The key fingerprint is:
   87:35:0e:4e:d2:96:5f:e4:ce:64:4a:d5:76:c8:2b:e4 thomas@mylaptop
   ```

7. Now copy the newly created public key to the `authorized_keys` file. This will allow us to connect to the Git server using this new key:

   ```
   t@mylaptop ~/.ssh $ ssh-copy-id -i git_rsa git@git.example.com
   git@git.example.com's password:
   Number of key(s) added: 1
   ```

8. Now try logging into the machine, with: "ssh 'git@git.example.com'" and check to make sure that only the key(s) you wanted were added.

9. Next, configure `ssh` to use your key when accessing the Git server and add the following to your `~/.ssh/config` file:

   ```
   Host git git.example.com
     User git
     IdentityFile /home/thomas/.ssh/git_rsa
   ```

10. Clone the repo onto your machine into a directory named Puppet (substitute your server name if you didn't use `git.example.com`):

    ```
    t@mylaptop ~$ git clone git@git.example.com:repos/puppet.git
    Cloning into 'puppet'...
    warning: You appear to have cloned an empty repository.
    Checking connectivity... done.
    ```

    We've created a Git repository; before we commit any changes to the repository, it's a good idea to set your name and e-mail in Git. Your name and e-mail will be appended to each commit you make.

11. When you are working in a large team, knowing who made a change is very important; for this, use the following code snippet:

    ```
    t@mylaptop puppet$ git config --global user.email
    "thomas@narrabilis.com"
    t@mylaptop puppet$ git config --global user.name "Thomas
    Uphill"
    ```

12. You can verify your Git settings using the following snippet:

    ```
    t@mylaptop ~$ git config --global --list
    user.name=Thomas Uphill
    user.email=thomas@narrabilis.com
    core.editor=vim
    merge.tool=vimdiff
    color.ui=true
    push.default=simple
    ```

13. Now that we have Git configured properly, change directory to your repository directory and create a new site manifest as shown in the following snippet:

    ```
    t@mylaptop ~$ cd puppet
    t@mylaptop puppet$ mkdir manifests
    t@mylaptop puppet$ vim manifests/site.pp
    node default {
      include base
    }
    ```

14. This site manifest will install our base class on every node; we will create the base class using the Puppet module as we did in *Chapter 1*, *Puppet Language and Style*:

    ```
    t@mylaptop puppet$ mkdir modules
    t@mylaptop puppet$ cd modules
    t@mylaptop modules$ puppet module generate thomas-base
    Notice: Generating module at /home/tuphill/puppet/modules/thomas-
    base
    thomas-base
    thomas-base/Modulefile
    thomas-base/README
    thomas-base/manifests
    thomas-base/manifests/init.pp
    thomas-base/spec
    thomas-base/spec/spec_helper.rb
    ```

```
thomas-base/tests
thomas-base/tests/init.pp
t@mylaptop modules$ ln -s thomas-base base
```

15. As a last step, we create a symbolic link between the `thomas-base` directory and `base`. Now to make sure our module does something useful, add the following to the body of the `base` class defined in `thomas-base/manifests/init.pp`:

```
class base {
  file {'/etc/motd':
    content => "${::fqdn}\nManaged by puppet ${::puppetversion}\n"
  }
}
```

16. Now add the new base module and site manifest to Git using `git add` and `git commit` as follows:

```
t@mylaptop modules$ cd ..
t@mylaptop puppet$ git add modules manifests
t@mylaptop puppet$ git status
On branch master
Initial commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
new file:   manifests/site.pp
new file:   modules/base
new file:   modules/thomas-base/Modulefile
new file:   modules/thomas-base/README
new file:   modules/thomas-base/manifests/init.pp
new file:   modules/thomas-base/spec/spec_helper.rb
new file:   modules/thomas-base/tests/init.pp
t@mylaptop puppet$ git commit -m "Initial commit with simple base
module"
[master (root-commit) 3e1f837] Initial commit with simple base
module
 7 files changed, 102 insertions(+)
 create mode 100644 manifests/site.pp
 create mode 120000 modules/base
 create mode 100644 modules/thomas-base/Modulefile
 create mode 100644 modules/thomas-base/README
 create mode 100644 modules/thomas-base/manifests/init.pp
 create mode 100644 modules/thomas-base/spec/spec_helper.rb
 create mode 100644 modules/thomas-base/tests/init.pp
```

17. At this point your changes to the Git repository have been committed locally; you now need to push those changes back to `git.example.com` so that other nodes can retrieve the updated files:

```
t@mylaptop puppet$ git push origin master
Counting objects: 15, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (15/15), 2.15 KiB | 0 bytes/s, done.
Total 15 (delta 0), reused 0 (delta 0)
To git@git.example.com:repos/puppet.git
 * [new branch]      master -> master
```

How it works...

Git tracks changes to files, and stores a complete history of all changes. The history of the repo is made up of commits. A commit represents the state of the repo at a particular point in time, which you create with the `git commit` command and annotate with a message.

You've now added your Puppet manifest files to the repo and created your first commit. This updates the history of the repo, but only in your local working copy. To synchronize the changes with the `git.example.com` copy, the `git push` command pushes all changes made since the last sync.

There's more...

Now that you have a central Git repository for your Puppet manifests, you can check out multiple copies of it in different places and work on them before committing your changes. For example, if you're working in a team, each member can have their own local copy of the repo and synchronize changes with the others via the central server. You may also choose to use GitHub as your central Git repository server. GitHub offers free Git repository hosting for public repositories, and you can pay for GitHub's premium service if you don't want your Puppet code to be publicly available.

In the next section, we will use our Git repository for both centralized and decentralized Puppet configurations.

Creating a decentralized Puppet architecture

Puppet is a configuration management tool. You can use Puppet to configure and prevent configuration drift in a large number of client computers. If all your client computers are easily reached via a central location, you may choose to have a central Puppet server control all the client computers. In the centralized model, the Puppet server is known as the Puppet master. We will cover how to configure a central Puppet master in a few sections.

If your client computers are widely distributed or you cannot guarantee communication between the client computers and a central location, then a decentralized architecture may be a good fit for your deployment. In the next few sections, we will see how to configure a decentralized Puppet architecture.

As we have seen, we can run the `puppet apply` command directly on a manifest file to have Puppet apply it. The problem with this arrangement is that we need to have the manifests transferred to the client computers.

We can use the Git repository we created in the previous section to transfer our manifests to each new node we create.

Getting ready

Create a new test node, call this new node whatever you wish, I'll use `testnode` for mine. Install Puppet on the machine as we have previously done.

How to do it...

Create a `bootstrap.pp` manifest that will perform the following configuration steps on our new node:

1. Install Git:

   ```
   package {'git':
     ensure => 'installed'
   }
   ```

2. Install the `ssh` key to access `git.example.com` in the Puppet user's home directory (`/var/lib/puppet/.ssh/id_rsa`):

   ```
   File {
     owner => 'puppet',
     group => 'puppet',
   }
   file {'/var/lib/puppet/.ssh':
     ensure => 'directory',
   }
   file {'/var/lib/puppet/.ssh/id_rsa':
   ```

```
  content => "
-----BEGIN RSA PRIVATE KEY-----
…
NIjTXmZUlOKefh4MBilqUU3KQG8GBHjzYl2TkFVGLNYGNA0U8VG8SUJq
-----END RSA PRIVATE KEY-----
",
  mode    => 0600,
  require => File['/var/lib/puppet/.ssh']
}
```

3. Download the `ssh` host key from `git.example.com` (`/var/lib/puppet/.ssh/known_hosts`):

```
exec {'download git.example.com host key':
  command => 'sudo -u puppet ssh-keyscan git.example.com >> /var/
lib/puppet/.ssh/known_hosts',
  path    => '/usr/bin:/usr/sbin:/bin:/sbin',
  unless  => 'grep git.example.com /var/lib/puppet/.ssh/known_
hosts',
  require => File['/var/lib/puppet/.ssh'],
}
```

4. Create a directory to contain the Git repository (`/etc/puppet/cookbook`):

```
file {'/etc/puppet/cookbook':
  ensure => 'directory',
}
```

5. Clone the Puppet repository onto the new machine:

```
exec {'create cookbook':
  command => 'sudo -u puppet git clone git@git.example.com:repos/
puppet.git /etc/puppet/cookbook',
  path    => '/usr/bin:/usr/sbin:/bin:/sbin',
  require => [Package['git'],File['/var/lib/puppet/.ssh/id_
rsa'],Exec['download git.example.com host key']],
  unless  => 'test -f /etc/puppet/cookbook/.git/config',
}
```

6. Now when we run Puppet apply on the new machine, the `ssh` key will be installed for the Puppet user. The Puppet user will then clone the Git repository into `/etc/puppet/cookbook`:

```
root@testnode /tmp# puppet apply bootstrap.pp
Notice: Compiled catalog for testnode.example.com in environment
production in 0.40 seconds
Notice: /Stage[main]/Main/File[/etc/puppet/cookbook]/ensure:
created
Notice: /Stage[main]/Main/File[/var/lib/puppet/.ssh]/ensure:
created
```

```
Notice: /Stage[main]/Main/Exec[download git.example.com host key]/
returns: executed successfully
Notice: /Stage[main]/Main/File[/var/lib/puppet/.ssh/id_rsa]/
ensure: defined content as '{md5}da61ce6ccc79bc6937bd98c798bc9fd3'
Notice: /Stage[main]/Main/Exec[create cookbook]/returns: executed
successfully
Notice: Finished catalog run in 0.82 seconds
```

You may have to disable the `tty` requirement of `sudo`. Comment out the line `Defaults requiretty` at `/etc/sudoers` if you have this line.

Alternatively, you can set `user => Puppet` within the `'create cookbook' exec` type. Beware that using the user attribute will cause any error messages from the command to be lost.

7. Now that your Puppet code is available on the new node, you can apply it using `puppet apply`, specifying that `/etc/puppet/cookbook/modules` will contain the modules:

```
root@testnode ~# puppet apply --modulepath=/etc/puppet/cookbook/
modules /etc/puppet/cookbook/manifests/site.pp
Notice: Compiled catalog for testnode.example.com in environment
production in 0.12 seconds
Notice: /Stage[main]/Base/File[/etc/motd]/content: content changed
'{md5}86d28ff83a8d49d349ba56b5c64b79ee' to '{md5}4c4c3ab7591d94031
8279d78b9c51d4f'
Notice: Finished catalog run in 0.11 seconds
root@testnode /tmp# cat /etc/motd
testnode.example.com
Managed by puppet 3.6.2
```

How it works...

First, our `bootstrap.pp` manifest ensures that Git is installed. The manifest then goes on to ensure that the `ssh` key for the Git user on `git.example.com` is installed into the Puppet user's home directory (`/var/lib/puppet` by default). The manifest then ensures that the host key for `git.example.com` is trusted by the Puppet user. With `ssh` configured, the bootstrap ensures that `/etc/puppet/cookbook` exists and is a directory.

We then use an `exec` to have Git clone the repository into `/etc/puppet/cookbook`. With all the code in place, we then call `puppet apply` a final time to deploy the code from the repository. In a production setting, you would distribute the `bootstrap.pp` manifest to all your nodes, possibly via an internal web server, using a method similar to curl `http://puppet/bootstrap.pp >bootstrap.pp && puppet apply bootstrap.pp`

Writing a papply script

We'd like to make it as quick and easy as possible to apply Puppet on a machine; for this we'll write a little script that wraps the `puppet apply` command with the parameters it needs. We'll deploy the script where it's needed with Puppet itself.

How to do it...

Follow these steps:

1. In your Puppet repo, create the directories needed for a Puppet module:

   ```
   t@mylaptop ~$ cd puppet/modules
   t@mylaptop modules$ mkdir -p puppet/{manifests,files}
   ```

2. Create the `modules/puppet/files/papply.sh` file with the following contents:

   ```
   #!/bin/sh
   sudo puppet apply /etc/puppet/cookbook/manifests/site.pp \
     --modulepath=/etc/puppet/cookbook/modules $*
   ```

3. Create the `modules/puppet/manifests/init.pp` file with the following contents:

   ```
   class puppet {
     file { '/usr/local/bin/papply':
       source => 'puppet:///modules/puppet/papply.sh',
       mode   => '0755',
     }
   }
   ```

4. Modify your `manifests/site.pp` file as follows:

   ```
   node default {
     include base
     include puppet
   }
   ```

5. Add the Puppet module to the Git repository and commit the change as follows:

   ```
   t@mylaptop puppet$ git add manifests/site.pp modules/puppet
   t@mylaptop puppet$ git status
   On branch master
   Your branch is up-to-date with 'origin/master'.
   Changes to be committed:
     (use "git reset HEAD <file>..." to unstage)
   modified:   manifests/site.pp
   ```

```
new file:   modules/puppet/files/papply.sh
new file:   modules/puppet/manifests/init.pp
t@mylaptop puppet$ git commit -m "adding puppet module to include
papply"
[master 7c2e3d5] adding puppet module to include papply
 3 files changed, 11 insertions(+)
 create mode 100644 modules/puppet/files/papply.sh
 create mode 100644 modules/puppet/manifests/init.pp
```

6. Now remember to push the changes to the Git repository on `git.example.com`:

```
t@mylaptop puppet$ git push origin master
Counting objects: 14, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (7/7), done.
Writing objects: 100% (10/10), 894 bytes | 0 bytes/s, done.
Total 10 (delta 0), reused 0 (delta 0)
To git@git.example.com:repos/puppet.git
   23e887c..7c2e3d5  master -> master
```

7. Pull the latest version of the Git repository to your new node (`testnode` for me) as shown in the following command line:

```
root@testnode ~# sudo -iu puppet
puppet@testnode ~$ cd /etc/puppet/cookbook/
puppet@testnode /etc/puppet/cookbook$ git pull origin master
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 10 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (10/10), done.
From git.example.com:repos/puppet
 * branch            master     -> FETCH_HEAD
Updating 23e887c..7c2e3d5
Fast-forward
 manifests/site.pp                   |    1 +
 modules/puppet/files/papply.sh      |    4 ++++
 modules/puppet/manifests/init.pp |    6 ++++++
 3 files changed, 11 insertions(+), 0 deletions(-)
 create mode 100644 modules/puppet/files/papply.sh
 create mode 100644 modules/puppet/manifests/init.pp
```

8. Apply the manifest manually once to install the `papply` script:

```
root@testnode ~# puppet apply /etc/puppet/cookbook/manifests/site.
pp --modulepath /etc/puppet/cookbook/modules
Notice: Compiled catalog for testnode.example.com in environment
production in 0.13 seconds
```

```
Notice: /Stage[main]/Puppet/File[/usr/local/bin/papply]/ensure:
defined content as '{md5}d5c2cdd359306dd6e6441e6fb96e5ef7'
Notice: Finished catalog run in 0.13 seconds
```

9. Finally, test the script:

```
root@testnode ~# papply
Notice: Compiled catalog for testnode.example.com in environment
production in 0.13 seconds
Notice: Finished catalog run in 0.09 seconds
```

Now, whenever you need to run Puppet, you can simply run `papply`. In future, when we apply Puppet changes, I'll ask you to run `papply` instead of the full `puppet apply` command.

How it works...

As you've seen, to run Puppet on a machine and apply a specified manifest file, we use the `puppet apply` command:

```
puppet apply manifests/site.pp
```

When you're using modules (such as the Puppet module we just created), you also need to tell Puppet where to search for modules, using the `modulepath` argument:

```
puppet apply manifests/nodes.pp \
  --modulepath=/home/ubuntu/puppet/modules
```

In order to run Puppet with the root privileges it needs, we have to put `sudo` before everything:

```
sudo puppet apply manifests/nodes.pp \
  --modulepath=/home/ubuntu/puppet/modules
```

Finally, any additional arguments passed to `papply` will be passed through to Puppet itself, by adding the `$*` parameter:

```
sudo puppet apply manifests/nodes.pp \
  --modulepath=/home/ubuntu/puppet/modules $*
```

That's a lot of typing, so putting this in a script makes sense. We've added a Puppet file resource that will deploy the script to `/usr/local/bin` and make it executable:

```
file { '/usr/local/bin/papply':
  source => 'puppet:///modules/puppet/papply.sh',
  mode   => '0755',
}
```

Finally, we include the Puppet module in our default node declaration:

```
node default {
  include base
  include puppet
}
```

You can do the same for any other nodes managed by Puppet.

Running Puppet from cron

You can do a lot with the setup you already have: work on your Puppet manifests as a team, communicate changes via a central Git repository, and manually apply them on a machine using the `papply` script.

However, you still have to log into each machine to update the Git repo and rerun Puppet. It would be helpful to have each machine update itself and apply any changes automatically. Then all you need to do is to push a change to the repo, and it will go out to all your machines within a certain time.

The simplest way to do this is with a **cron** job that pulls updates from the repo at regular intervals and then runs Puppet if anything has changed.

Getting ready

You'll need the Git repo we set up in the *Managing your manifests with Git* and *Creating a decentralized Puppet architecture* recipes, and the `papply` script from the *Writing a papply script* recipe. You'll need to apply the `bootstrap.pp` manifest we created to install `ssh` keys to download the latest repository.

How to do it...

Follow these steps:

1. Copy the `bootstrap.pp` script to any node you wish to enroll. The `bootstrap.pp` manifest includes the private key used to access the Git repository, it should be protected in a production environment.
2. Create the `modules/puppet/files/pull-updates.sh` file with the following contents:

   ```
   #!/bin/sh
   cd /etc/puppet/cookbook
   sudo -u puppet git pull && /usr/local/bin/papply
   ```

3. Modify the `modules/puppet/manifests/init.pp` file and add the following snippet after the `papply` file definition:

   ```
   file { '/usr/local/bin/pull-updates':
     source => 'puppet:///modules/puppet/pull-updates.sh',
   ```

```
  mode    => '0755',
}
cron { 'run-puppet':
  ensure  => 'present',
  user    => 'puppet',
  command => '/usr/local/bin/pull-updates',
  minute  => '*/10',
  hour    => '*',
}
```

4. Commit the changes as before and push to the Git server as shown in the following command line:

```
t@mylaptop puppet$ git add modules/puppet
t@mylaptop puppet$ git commit -m "adding pull-updates"
[master 7e9bac3] adding pull-updates
 2 files changed, 14 insertions(+)
 create mode 100644 modules/puppet/files/pull-updates.sh
t@mylaptop puppet$ git push
Counting objects: 14, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (7/7), done.
Writing objects: 100% (8/8), 839 bytes | 0 bytes/s, done.
Total 8 (delta 0), reused 0 (delta 0)
To git@git.example.com:repos/puppet.git
   7c2e3d5..7e9bac3  master -> master
```

5. Issue a Git pull on the test node:

```
root@testnode ~# cd /etc/puppet/cookbook/
root@testnode /etc/puppet/cookbook# sudo –u puppet git pull
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 8 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (8/8), done.
From git.example.com:repos/puppet
   23e887c..7e9bac3  master     -> origin/master
Updating 7c2e3d5..7e9bac3
Fast-forward
 modules/puppet/files/pull-updates.sh |    3 +++
 modules/puppet/manifests/init.pp     |   11 +++++++++++
 2 files changed, 14 insertions(+), 0 deletions(-)
 create mode 100644 modules/puppet/files/pull-updates.sh
```

6. Run Puppet on the test node:

```
root@testnode /etc/puppet/cookbook# papply

Notice: Compiled catalog for testnode.example.com in environment
production in 0.17 seconds

Notice: /Stage[main]/Puppet/Cron[run-puppet]/ensure: created

Notice: /Stage[main]/Puppet/File[/usr/local/bin/pull-updates]/
ensure: defined content as '{md5}04c023feb5d566a417b519ea51586398'

Notice: Finished catalog run in 0.16 seconds
```

7. Check that the `pull-updates` script works properly:

```
root@testnode /etc/puppet/cookbook# pull-updates

Already up-to-date.

Notice: Compiled catalog for testnode.example.com in environment
production in 0.15 seconds

Notice: Finished catalog run in 0.14 seconds
```

8. Verify the `cron` job was created successfully:

```
root@testnode /etc/puppet/cookbook# crontab -l -u puppet

# HEADER: This file was autogenerated at Tue Sep 09 02:31:00 -0400
2014 by puppet.

# HEADER: While it can still be managed manually, it is definitely
not recommended.

# HEADER: Note particularly that the comments starting with
'Puppet Name' should

# HEADER: not be deleted, as doing so could cause duplicate cron
jobs.

# Puppet Name: run-puppet

*/10 * * * * /usr/local/bin/pull-updates
```

How it works...

When we created the `bootstrap.pp` manifest, we made sure that the Puppet user can checkout the Git repository using an `ssh` key. This enables the Puppet user to run the Git pull in the cookbook directory unattended. We've also added the `pull-updates` script, which does this and runs Puppet if any changes are pulled:

```
#!/bin/sh
cd /etc/puppet/cookbook
sudo –u puppet git pull && papply
```

We deploy this script to the node with Puppet:

```
file { '/usr/local/bin/pull-updates':
  source => 'puppet:///modules/puppet/pull-updates.sh',
  mode   => '0755',
}
```

Finally, we've created a `cron` job that runs `pull-updates` at regular intervals (every 10 minutes, but feel free to change this if you need to):

```
cron { 'run-puppet':
  ensure  => 'present',
  command => '/usr/local/bin/pull-updates',
  minute  => '*/10',
  hour    => '*',
}
```

There's more...

Congratulations, you now have a fully-automated Puppet infrastructure! Once you have applied the `bootstrap.pp` manifest, run Puppet on the repository; the machine will be set up to pull any new changes and apply them automatically.

So, for example, if you wanted to add a new user account to all your machines, all you have to do is add the account in your working copy of the manifest, and commit and push the changes to the central Git repository. Within 10 minutes, it will automatically be applied to every machine that's running Puppet.

Bootstrapping Puppet with bash

Previous versions of this book used Rakefiles to bootstrap Puppet. The problem with using Rake to configure a node is that you are running the commands from your laptop; you assume you already have `ssh` access to the machine. Most bootstrap processes work by issuing an easy to remember command from a node once it has been provisioned. In this section, we'll show how to use bash to bootstrap Puppet with a web server and a bootstrap script.

Getting ready

Install httpd on a centrally accessible server and create a password protected area to store the bootstrap script. In my example, I'll use the Git server I set up previously, `git.example.com`. Start by creating a directory in the root of your web server:

```
# cd /var/www/html
# mkdir bootstrap
```

Now perform the following steps:

1. Add the following location definition to your apache configuration:

   ```
   <Location /bootstrap>
   AuthType basic
   AuthName "Bootstrap"
   AuthBasicProvider file
   AuthUserFile /var/www/puppet.passwd
   Require valid-user
   </Location>
   ```

2. Reload your web server to ensure the location configuration is operating. Verify with curl that you cannot download from the bootstrap directory without authentication:

   ```
   [root@bootstrap-test tmp]# curl http://git.example.com/bootstrap/
   <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
   <html><head>
   <title>401 Authorization Required</title>
   </head><body>
   <h1>Authorization Required</h1>
   ```

3. Create the password file you referenced in the apache configuration (`/var/www/puppet.passwd`):

   ```
   root@git# cd /var/www
   root@git# htpasswd –cb puppet.passwd bootstrap cookbook
   Adding password for user bootstrap
   ```

4. Verify that the username and password permit access to the bootstrap directory as follows:

   ```
   [root@node1 tmp]# curl --user bootstrap:cookbook http://git.
   example.com/bootstrap/
   <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
   <html>
    <head>
     <title>Index of /bootstrap</title>
   ```

How to do it...

Now that you have a safe location to store the bootstrap script, create a bootstrap script for each OS you support in the bootstrap directory. In this example, I'll show you how to do this for a Red Hat Enterprise Linux 6-based distribution.

[Although the bootstrap location requires a password, there is no encryption since we haven't configured SSL on our server. Without encryption, the location is not very safe.]

Create a script named `el6.sh` in the bootstrap directory with the following contents:

```
#!/bin/bash

# bootstrap for EL6 distributions
SERVER=git.example.com
LOCATION=/bootstrap
BOOTSTRAP=bootstrap.pp
USER=bootstrap
PASS=cookbook

# install puppet
curl http://yum.puppetlabs.com/RPM-GPG-KEY-puppetlabs >/etc/pki/rpm-
gpg/RPM-GPG-KEY-puppetlabs
rpm --import /etc/pki/rpm-gpg/RPM-GPG-KEY-puppetlabs
yum -y install http://yum.puppetlabs.com/puppetlabs-release-el-6.
noarch.rpm
yum -y install puppet
# download bootstrap
curl --user $USER:$PASS http://$SERVER/$LOCATION/$BOOTSTRAP >/
tmp/$BOOTSTRAP
# apply bootstrap
cd /tmp
puppet apply /tmp/$BOOTSTRAP
# apply puppet
puppet apply --modulepath /etc/puppet/cookbook/modules /etc/puppet/
cookbook/manifests/site.pp
```

How it works...

The apache configuration only permits access to the bootstrap directory with a username and password combination. We supply these with the `--user` argument to curl, thereby getting access to the file. We use a pipe (|) to redirect the output of curl into bash. This causes bash to execute the script. We write our bash script like we would any other bash script. The bash script downloads our `bootstrap.pp` manifest and applies it. Finally, we apply the Puppet manifest from the Git repository and the machine is configured as a member of our decentralized infrastructure.

There's more...

To support another operating system, we only need to create a new bash script. All Linux distributions will support bash scripting, Mac OS X does as well. Since we placed much of our logic into the `bootstrap.pp` manifest, the bootstrap script is quite minimal and easy to port to new operating systems.

Creating a centralized Puppet infrastructure

A configuration management tool such as Puppet is best used when you have many machines to manage. If all the machines can reach a central location, using a centralized Puppet infrastructure might be a good solution. Unfortunately, Puppet doesn't scale well with a large number of nodes. If your deployment has less than 800 servers, a single Puppet master should be able to handle the load, assuming your catalogs are not complex (take less than 10 seconds to compile each catalog). If you have a larger number of nodes, I suggest a load balancing configuration described in *Mastering Puppet*, *Thomas Uphill*, *Packt Publishing*.

A Puppet master is a Puppet server that acts as an X509 certificate authority for Puppet and distributes catalogs (compiled manifests) to client nodes. Puppet ships with a built-in web server called **WEBrick,** which can handle a very small number of nodes. In this section, we will see how to use that built-in server to control a very small (less than 10) number of nodes.

Getting ready

The Puppet master process is started by running `puppet master`; most Linux distributions have start and stop scripts for the Puppet master in a separate package. To get started, we'll create a new debian server named `puppet.example.com`.

How to do it...

1. Install Puppet on the new server and then use Puppet to install the Puppet master package:

   ```
   # puppet resource package puppetmaster ensure='installed'
   Notice: /Package[puppetmaster]/ensure: created
   package { 'puppetmaster':
     ensure => '3.7.0-1puppetlabs1',
   }
   ```

2. Now start the Puppet master service and ensure it will start at boot:

   ```
   # puppet resource service puppetmaster ensure=true enable=true
   service { 'puppetmaster':
     ensure => 'running',
     enable => 'true',
   }
   ```

How it works...

The Puppet master package includes the start and stop scripts for the Puppet master service. We use Puppet to install the package and start the service. Once the service is started, we can point another node at the Puppet master (you might need to disable the host-based firewall on your machine).

1. From another node, run `puppet agent` to start a `puppet agent`, which will contact the server and request a new certificate:

   ```
   t@ckbk:~$ sudo puppet agent -t
   Info: Creating a new SSL key for cookbook.example.com
   Info: Caching certificate for ca
   Info: Creating a new SSL certificate request for cookbook.example.
   com
   Info: Certificate Request fingerprint (SHA256): 06:C6:2B:C4:97:5D:
   16:F2:73:82:C4:A9:A7:B1:D0:95:AC:69:7B:27:13:A9:1A:4C:98:20:21:C2:
   50:48:66:A2
   Info: Caching certificate for ca
   Exiting; no certificate found and waitforcert is disabled
   ```

2. Now on the Puppet server, sign the new key:

   ```
   root@puppet:~# puppet cert list
   pu  "cookbook.example.com" (SHA256) 06:C6:2B:C4:97:5D:16:F2:73:82:
   C4:A9:A7:B1:D0:95:AC:69:7B:27:13:A9:1A:4C:98:20:21:C2:50:48:66:A2
   root@puppet:~# puppet cert sign cookbook.example.com
   Notice: Signed certificate request for cookbook.example.com
   Notice: Removing file Puppet::SSL::CertificateRequest
   cookbook.example.com at
   '/var/lib/puppet/ssl/ca/requests/cookbook.example.com.pem'
   ```

3. Return to the cookbook node and run Puppet again:

   ```
   t@ckbk:~$ sudo puppet agent -vt
   Info: Caching certificate for cookbook.example.com
   Info: Caching certificate_revocation_list for ca
   Info: Caching certificate for cookbook.example.comInfo: Retrieving
   pluginfacts
   Info: Retrieving plugin
   Info: Caching catalog for cookbook
   Info: Applying configuration version '1410401823'
   Notice: Finished catalog run in 0.04 seconds
   ```

There's more...

When we ran `puppet agent`, Puppet looked for a host named `puppet.example.com` (since our test node is in the `example.com` domain); if it couldn't find that host, it would then look for a host named Puppet. We can specify the server to contact with the `--server` option to `puppet agent`. When we installed the Puppet master package and started the Puppet master service, Puppet created default SSL certificates based on our hostname. In the next section, we'll see how to create an SSL certificate that has multiple DNS names for our Puppet server.

Creating certificates with multiple DNS names

By default, Puppet will create an SSL certificate for your Puppet master that contains the fully qualified domain name of the server only. Depending on how your network is configured, it can be useful for the server to be known by other names. In this recipe, we'll make a new certificate for our Puppet master that has multiple DNS names.

Getting ready

Install the Puppet master package if you haven't already done so. You will then need to start the Puppet master service at least once to create a **certificate authority** (**CA**).

How to do it...

The steps are as follows:

1. Stop the running Puppet master process with the following command:

   ```
   # service puppetmaster stop
   [ ok ] Stopping puppet master.
   ```

2. Delete (`clean`) the current server certificate:

   ```
   # puppet cert clean puppet
   Notice: Revoked certificate with serial 6
   Notice: Removing file Puppet::SSL::Certificate puppet at '/var/
   lib/puppet/ssl/ca/signed/puppet.pem'
   Notice: Removing file Puppet::SSL::Certificate puppet at '/var/
   lib/puppet/ssl/certs/puppet.pem'
   Notice: Removing file Puppet::SSL::Key puppet at '/var/lib/puppet/
   ssl/private_keys/puppet.pem'
   ```

3. Create a new Puppet certificate using Puppet certificate generate with the `--dns-alt-names` option:

```
root@puppet:~# puppet certificate generate puppet --dns-alt-names
puppet.example.com,puppet.example.org,puppet.example.net --ca-
location local
Notice: puppet has a waiting certificate request
true
```

4. Sign the new certificate:

```
root@puppet:~# puppet cert --allow-dns-alt-names sign puppet
Notice: Signed certificate request for puppet
Notice: Removing file Puppet::SSL::CertificateRequest puppet at '/
var/lib/puppet/ssl/ca/requests/puppet.pem'
```

5. Restart the Puppet master process:

```
root@puppet:~# service puppetmaster restart
[ ok ] Restarting puppet master.
```

How it works...

When your puppet agents connect to the Puppet server, they look for a host called `Puppet`, they then look for a host called `Puppet.[your domain]`. If your clients are in different domains, then you need your Puppet master to reply to all the names correctly. By removing the existing certificate and generating a new one, you can have your Puppet master reply to multiple DNS names.

Running Puppet from passenger

The WEBrick server we configured in the previous section is not capable of handling a large number of nodes. To deal with a large number of nodes, a scalable web server is required. Puppet is a ruby process, so we need a way to run a ruby process within a web server. **Passenger** is the solution to this problem. It allows us to run the Puppet master process within a web server (apache by default). Many distributions ship with a puppetmaster-passenger package that configures this for you. In this section, we'll use the package to configure Puppet to run within passenger.

Getting ready

Install the puppetmaster-passenger package:

```
# puppet resource package puppetmaster-passenger ensure=installed
Notice: /Package[puppetmaster-passenger]/ensure: ensure changed 'purged'
```

```
 to 'present'
package { 'puppetmaster-passenger':
  ensure => '3.7.0-1puppetlabs1',
}
```

> Using `puppet resource` to install packages ensures the same command will work on multiple distributions (provided the package names are the same).

How to do it...

The steps are as follows:

1. Ensure the Puppet master site is enabled in your apache configuration. Depending on your distribution this may be at `/etc/httpd/conf.d` or `/etc/apache2/sites-enabled`. The configuration file should be created for you and contain the following information:

   ```
   PassengerHighPerformance on
   PassengerMaxPoolSize 12
   PassengerPoolIdleTime 1500
   # PassengerMaxRequests 1000
   PassengerStatThrottleRate 120
   RackAutoDetect Off
   RailsAutoDetect Off
   Listen 8140
   ```

2. These lines are tuning settings for passenger. The file then instructs apache to listen on port 8140, the Puppet master port. Next a `VirtualHost` definition is created that loads the Puppet CA certificates and the Puppet master's certificate:

   ```
   <VirtualHost *:8140>
           SSLEngine on
           SSLProtocol             ALL -SSLv2 -SSLv3
           SSLCertificateFile      /var/lib/puppet/ssl/certs/puppet.
   pem
           SSLCertificateKeyFile   /var/lib/puppet/ssl/private_keys/
   puppet.pem
           SSLCertificateChainFile /var/lib/puppet/ssl/certs/ca.pem
           SSLCACertificateFile    /var/lib/puppet/ssl/certs/ca.pem
           SSLCARevocationFile     /var/lib/puppet/ssl/ca/ca_crl.pem
   ```

```
              SSLVerifyClient optional
              SSLVerifyDepth  1
              SSLOptions +StdEnvVars +ExportCertData
```

You may have more or less lines of SSL configuration here depending on your version of the puppetmaster-passenger package.

3. Next, a few important headers are set so that the passenger process has access to the SSL information sent by the client node:

```
RequestHeader unset X-Forwarded-For
RequestHeader set X-SSL-Subject %{SSL_CLIENT_S_DN}e
RequestHeader set X-Client-DN %{SSL_CLIENT_S_DN}e
RequestHeader set X-Client-Verify %{SSL_CLIENT_VERIFY}e
```

4. Finally, the location of the passenger configuration file `config.ru` is given with the `DocumentRoot` location as follows:

```
       DocumentRoot /usr/share/puppet/rack/puppetmasterd/public/
        RackBaseURI /
```

5. The `config.ru` file should exist at `/usr/share/puppet/rack/puppetmasterd/` and should have the following content:

```
$0 = "master"
ARGV << "--rack"
ARGV << "--confdir" << "/etc/puppet"
ARGV << "--vardir"  << "/var/lib/puppet"
require 'puppet/util/command_line'
run Puppet::Util::CommandLine.new.execute
```

6. With the passenger apache configuration file in place and the `config.ru` file correctly configured, start the apache server and verify that apache is listening on the Puppet master port (if you configured the standalone Puppet master previously, you must stop that process now using `service puppetmaster stop`):

```
root@puppet:~ # service apache2 start
[ ok ] Starting web server: apache2
root@puppet:~ # lsof -i :8140
COMMAND  PID     USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
apache2 9048     root    8u  IPv6  16842      0t0  TCP *:8140
(LISTEN)
```

```
apache2  9069 www-data    8u   IPv6   16842       0t0   TCP *:8140
(LISTEN)
apache2  9070 www-data    8u   IPv6   16842       0t0   TCP *:8140
(LISTEN)
```

How it works...

The passenger configuration file uses the existing Puppet master certificates to listen on port 8140 and handles all the SSL communication between the server and the client. Once the certificate information has been dealt with, the connection is handed off to a ruby process started from passenger using the command line arguments from the `config.ru` file.

In this case, the `$0` variable is set to `master` and the arguments variable is set to `--rack --confdir /etc/puppet --vardir /var/lib/puppet`; this is equivalent to running the following from the command line:

```
puppet master --rack --confdir /etc/puppet --vardir /var/lib/puppet
```

There's more...

You can add additional configuration parameters to the `config.ru` file to further alter how Puppet runs when it's running through passenger. For instance, to enable debugging on the passenger Puppet master, add the following line to `config.ru` before the run `Puppet::Util::CommandLine.new.execute` line:

```
ARGV << "--debug"
```

Setting up the environment

Environments in Puppet are directories holding different versions of your Puppet manifests. Environments prior to Version 3.6 of Puppet were not a default configuration for Puppet. In newer versions of Puppet, environments are configured by default.

Whenever a node connects to a Puppet master, it informs the Puppet master of its environment. By default, all nodes report to the `production` environment. This causes the Puppet master to look in the production environment for manifests. You may specify an alternate environment with the `--environment` setting when running puppet agent or by setting environment `= newenvironment` in `/etc/puppet/puppet.conf` in the [agent] section.

Getting ready

Set the `environmentpath` function of your installation by adding a line to the `[main]` section of `/etc/puppet/puppet.conf` as follows:

```
[main]
...
environmentpath=/etc/puppet/environments
```

How to do it...

The steps are as follows:

1. Create a `production` directory at `/etc/puppet/environments` that contains both a `modules` and `manifests` directory. Then create a `site.pp` which creates a file in `/tmp` as follows:

   ```
   root@puppet:~# cd /etc/puppet/environments/
   root@puppet:/etc/puppet/environments# mkdir -p production/
   {manifests,modules}
   root@puppet:/etc/puppet/environments# vim production/manifests/
   site.pp
   node default {
     file {'/tmp/production':
       content => "Hello World!\nThis is production\n",
     }
   }
   ```

2. Run puppet agent on the master to connect to it and verify that the production code was delivered:

   ```
   root@puppet:~# puppet agent -vt
   Info: Retrieving pluginfacts
   Info: Retrieving plugin
   Info: Caching catalog for puppet
   Info: Applying configuration version '1410415538'
   Notice: /Stage[main]/Main/Node[default]/File[/tmp/production]/
   ensure: defined content as '{md5}f7ad9261670b9da33a67a5126933044c'
   Notice: Finished catalog run in 0.04 seconds
   # cat /tmp/production
   Hello World!
   This is production
   ```

3. Configure another environment `devel`. Create a new manifest in the `devel` environment:

```
root@puppet:/etc/puppet/environments# mkdir -p devel/
{manifests,modules}
root@puppet:/etc/puppet/environments# vim devel/manifests/site.pp
node default {
  file {'/tmp/devel':
    content => "Good-bye! Development\n",
  }
}
```

4. Apply the new environment by running the `--environment devel` puppet agent using the following command:

```
root@puppet:/etc/puppet/environments# puppet agent -vt
--environment devel
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for puppet
Info: Applying configuration version '1410415890'
Notice: /Stage[main]/Main/Node[default]/File[/tmp/devel]/ensure:
defined content as '{md5}b6313bb89bc1b7d97eae5aa94588eb68'
Notice: Finished catalog run in 0.04 seconds
root@puppet:/etc/puppet/environments# cat /tmp/devel
Good-bye! Development
```

You may need to restart apache2 to enable your new environment, this depends on your version of Puppet and the `environment_timeout` parameter of `puppet.conf`.

There's more...

Each environment can have its own `modulepath` if you create an `environment.conf` file within the environment directory. More information on environments can be found on the Puppet labs website at `https://docs.puppetlabs.com/puppet/latest/reference/environments.html`.

Configuring PuppetDB

PuppetDB is a database for Puppet that is used to store information about nodes connected to a Puppet master. PuppetDB is also a storage area for exported resources. Exported resources are resources that are defined on nodes but applied to other nodes. The simplest way to install PuppetDB is to use the PuppetDB module from Puppet labs. From this point on, we'll assume you are using the `puppet.example.com` machine and have a passenger-based configuration of Puppet.

Getting ready

Install the PuppetDB module in the production environment you created in the previous recipe. If you didn't create directory environments, don't worry, using `puppet module install` will install the module to the correct location for your installation with the following command:

```
root@puppet:~# puppet module install puppetlabs-puppetdb
Notice: Preparing to install into /etc/puppet/environments/production/
modules ...
Notice: Downloading from https://forgeapi.puppetlabs.com ...
Notice: Installing -- do not interrupt ...
/etc/puppet/environments/production/modules
└─┬ puppetlabs-puppetdb (v3.0.1)
  ├── puppetlabs-firewall (v1.1.3)
  ├── puppetlabs-inifile (v1.1.3)
  └─┬ puppetlabs-postgresql (v3.4.2)
    ├─┬ puppetlabs-apt (v1.6.0)
    │ └── puppetlabs-stdlib (v4.3.2)
    └── puppetlabs-concat (v1.1.0)
```

How to do it...

Now that our Puppet master has the PuppetDB module installed, we need to apply the PuppetDB module to our Puppet master, we can do this in the site manifest. Add the following to your (production) `site.pp`:

```
node puppet {
  class { 'puppetdb': }
  class { 'puppetdb::master::config':
    puppet_service_name => 'apache2',
  }
}
```

Run `puppet agent` to apply the `puppetdb` class and the `puppetdb::master::config` class:

```
root@puppet:~# puppet agent -t
Info: Caching catalog for puppet
Info: Applying configuration version '1410416952'
...
Info: Class[Puppetdb::Server::Jetty_ini]: Scheduling refresh of
Service[puppetdb]
Notice: Finished catalog run in 160.78 seconds
```

How it works...

The PuppetDB module is a great example of how a complex configuration task can be puppetized. Simply by adding the `puppetdb` class to our Puppet master node, Puppet installed and configured `postgresql` and `puppetdb`.

When we called the `puppetdb::master::config` class, we set the `puppet_service_name` variable to `apache2`, this is because we are running Puppet through passenger. Without this line our agent would try to start the puppetmaster process instead of `apache2`.

The agent then set up the configuration files for PuppetDB and configured Puppet to use PuppetDB. If you look at `/etc/puppet/puppet.conf`, you'll see the following two new lines:

```
    storeconfigs = true
    storeconfigs_backend = puppetdb
```

There's more...

Now that PuppetDB is configured and we've had a successful agent run, PuppetDB will have data we can query:

```
root@puppet:~# puppet node status puppet
puppet
Currently active
Last catalog: 2014-09-11T06:45:25.267Z
Last facts: 2014-09-11T06:45:22.351Z
```

Configuring Hiera

Hiera is an information repository for Puppet. Using Hiera you can have a hierarchical categorization of data about your nodes that is maintained outside of your manifests. This is very useful for sharing code and dealing with exceptions that will creep into any Puppet deployment.

Getting ready

Hiera should have already been installed as a dependency on your Puppet master. If it has not already, install it using Puppet:

```
root@puppet:~# puppet resource package hiera ensure=installed
package { 'hiera':
  ensure => '1.3.4-1puppetlabs1',
}
```

How to do it...

1. Hiera is configured from a yaml file, `/etc/puppet/hiera.yaml`. Create the file and add the following as a minimal configuration:

   ```
   ---
   :hierarchy:
     - common
   :backends:
     - yaml
   :yaml:
     :datadir: '/etc/puppet/hieradata'
   ```

2. Create the `common.yaml` file referenced in the hierarchy:

   ```
   root@puppet:/etc/puppet# mkdir hieradata
   root@puppet:/etc/puppet# vim hieradata/common.yaml
   ---
   message: 'Default Message'
   ```

3. Edit the `site.pp` file and add a notify resource based on the Hiera value:

   ```
   node default {
     $message = hiera('message','unknown')
     notify {"Message is $message":}
   }
   ```

4. Apply the manifest to a test node:

   ```
   t@ckbk:~$ sudo puppet agent -t
   Info: Retrieving pluginfacts
   Info: Retrieving plugin
   ...
   Info: Caching catalog for cookbook-test
   Info: Applying configuration version '1410504848'
   Notice: Message is Default Message
   ```

```
Notice: /Stage[main]/Main/Node[default]/Notify[Message is Default
Message]/message: defined 'message' as 'Message is Default
Message'

Notice: Finished catalog run in 0.06 seconds
```

How it works...

Hiera uses a hierarchy to search through a set of yaml files to find the appropriate values. We defined this hierarchy in `hiera.yaml` with the single entry for `common.yaml`. We used the `hiera` function in `site.pp` to lookup the value for message and store that value in the variable `$message`. The values used for the definition of the hierarchy can be any facter facts defined about the system. A common hierarchy is shown as:

```
:hierarchy:
  - hosts/%{hostname}
  - os/%{operatingsystem}
  - network/%{network_eth0}
  - common
```

There's more...

Hiera can be used for automatic parameter lookup with parameterized classes. For example, if you have a class named `cookbook::example` with a parameter named `publisher`, you can include the following in a Hiera yaml file to automatically set this parameter:

```
cookbook::example::publisher: 'PacktPub'
```

Another often used fact is `environment` you may reference the `environment` of the client node using `%{environment}` as shown in the following hierarchy:

```
:hierarchy:
hosts/%{hostname}
os/%{operatingsystem}
environment/%{environment}
common
```

[A good rule of thumb is to limit the hierarchy to 8 levels or less. Keep in mind that each time a parameter is searched with Hiera, all the levels are searched until a match is found.]

The default Hiera function returns the first match to the search key, you can also use `hiera_array` and `hiera_hash` to search and return all values stored in Hiera.

Hiera can also be searched from the command line as shown in the following command line (note that currently the command line Hiera utility uses `/etc/hiera.yaml` as its configuration file whereas the Puppet master uses `/etc/puppet/hiera.yaml`):

```
root@puppet:/etc/puppet# rm /etc/hiera.yaml
root@puppet:/etc/puppet# ln -s /etc/puppet/hiera.yaml /etc/
root@puppet:/etc/puppet# hiera message
Default Message
```

For more information, consult the Puppet labs website at `https://docs.puppetlabs.com/hiera/1/`.

Setting node-specific data with Hiera

In our hierarchy defined in `hiera.yaml`, we created an entry based on the hostname fact; in this section, we'll create yaml files in the `hosts` subdirectory of Hiera data with information specific to a particular host.

Getting ready

Install and configure Hiera as in the last section and use the hierarchy defined in the previous recipe that includes a `hosts/%{hostname}` entry.

How to do it...

The following are the steps:

1. Create a file at `/etc/puppet/hieradata/hosts` that is the hostname of your test node. For example if your host is named `cookbook-test`, then the file would be named `cookbook-test.yaml`.
2. Insert a specific message in this file:

   ```
   message: 'This is the test node for the cookbook'
   ```

3. Run Puppet on two different test nodes to note the difference:

   ```
   t@ckbk:~$ sudo puppet agent -t
   Info: Caching catalog for cookbook-test
   Notice: Message is This is the test node for the cookbook
   [root@hiera-test ~]# puppet agent -t
   Info: Caching catalog for hiera-test.example.com
   Notice: Message is Default Message
   ```

How it works...

Hiera searches the hierarchy for files that match the values returned by facter. In this case, the `cookbook-test.yaml` file is found by substituting the hostname of the node into the search path `/etc/puppet/hieradata/hosts/%{hostname}.yaml`.

Using Hiera, it is possible to greatly reduce the complexity of your Puppet code. We will use `yaml` files for separate values, where previously you had large `case` statements or nested `if` statements.

Storing secret data with hiera-gpg

If you're using Hiera to store your configuration data, there's a gem available called **hiera-gpg** that adds an encryption backend to Hiera to allow you to protect values stored in Hiera.

Getting ready

To set up hiera-gpg, follow these steps:

1. Install the `ruby-dev` package; it will be required to build the `hiera-gpg` gem as follows:

   ```
   root@puppet:~# puppet resource package ruby-dev ensure=installed
   Notice: /Package[ruby-dev]/ensure: ensure changed 'purged' to
   'present'
   package { 'ruby-dev':
     ensure => '1:1.9.3',
   }
   ```

2. Install the `hiera-gpg` gem using the gem provider:

   ```
   root@puppet:~# puppet resource package hiera-gpg ensure=installed
   provider=gem
   Notice: /Package[hiera-gpg]/ensure: created
   package { 'hiera-gpg':
     ensure => ['1.1.0'],
   }
   ```

3. Modify your `hiera.yaml` file as follows:

   ```
       :hierarchy:
           - secret
           - common
       :backends:
           - yaml
   ```

```
        - gpg
    :yaml:
        :datadir: '/etc/puppet/hieradata'
    :gpg:
        :datadir: '/etc/puppet/secret'
```

How to do it...

In this example, we'll create a piece of encrypted data and retrieve it using `hiera-gpg` as follows:

1. Create the `secret.yaml` file at `/etc/puppet/secret` with the following contents:

   ```
   top_secret: 'Val Kilmer'
   ```

2. If you don't already have a GnuPG encryption key, follow the steps in the *Using GnuPG to encrypt secrets* recipe in *Chapter 4*, *Working with Files and Packages*.
3. Encrypt the `secret.yaml` file to this key using the following command (replace the `puppet@puppet.example.com` with the e-mail address you specified when creating the key). This will create the `secret.gpg` file:

   ```
   root@puppet:/etc/puppet/secret# gpg -e -o secret.gpg -r puppet@
   puppet.example.com secret.yaml
   root@puppet:/etc/puppet/secret# file secret.gpg
   secret.gpg: GPG encrypted data
   ```

4. Remove the plaintext `secret.yaml` file:

   ```
   root@puppet:/etc/puppet/secret# rm secret.yaml
   ```

5. Modify your default node in the `site.pp` file as follows:

   ```
   node default {
     $message = hiera('top_secret','Deja Vu')
     notify { "Message is $message": }
   }
   ```

6. Now run Puppet on a node:

   ```
   [root@hiera-test ~]# puppet agent -t
   Info: Caching catalog for hiera-test.example.com
   Info: Applying configuration version '1410508276'
   Notice: Message is Deja Vu
   Notice: /Stage[main]/Main/Node[default]/Notify[Message is Deja
   Vu]/message: defined 'message' as 'Message is Deja Vu'
   Notice: Finished catalog run in 0.08 seconds
   ```

How it works...

When you install `hiera-gpg`, it adds to Hiera, the ability to decrypt `.gpg` files. So you can put any secret data into a `.yaml` file that you then encrypt to the appropriate key with GnuPG. Only machines that have the right secret key will be able to access this data.

For example, you might encrypt the MySQL root password using `hiera-gpg` and install the corresponding key only on your database servers. Although other machines may also have a copy of the `secret.gpg` file, it's not readable to them unless they have the decryption key.

There's more...

You might also like to know about `hiera-eyaml`, another secret-data backend for Hiera that supports encryption of individual values within a Hiera data file. This could be handy if you need to mix encrypted and unencrypted facts within a single file. Find out more about hiera-eyaml at `https://github.com/TomPoulton/hiera-eyaml`.

See also

- The *Using GnuPG to encrypt secrets* recipe in *Chapter 4*, *Working with Files and Packages*.

Using MessagePack serialization

Running Puppet in a centralized architecture creates a lot of traffic between nodes. The bulk of this traffic is JSON and yaml data. An experimental feature of the latest releases of Puppet allow for the serialization of this data using **MessagePack** (**msgpack**).

Getting ready

Install the msgpack gem onto your Puppet master and your nodes. Use Puppet to do the work for you with Puppet resource. You may need to install the `ruby-dev` or `ruby-devel` package on your nodes/server at this point:

```
t@ckbk:~$ sudo puppet resource package msgpack ensure=installed
 provider=gem
Notice: /Package[msgpack]/ensure: created
package { 'msgpack':
  ensure => ['0.5.8'],
}
```

How to do it...

Set the `preferred_serialization_format` to `msgpack` in the `[agent]` section of your nodes `puppet.conf` file:

```
[agent]
preferred_serialization_format=msgpack
```

How it works...

The master will be sent this option when the node begins communicating with the master. Any classes that support serialization with `msgpack` will be transmitted with `msgpack`. `Serialization` of the data between nodes and the master will in theory increase the speed at which nodes communicate by optimizing the data that is travelling between them. This feature is still experimental.

Automatic syntax checking with Git hooks

It would be nice if we knew there was a syntax error in the manifest before we even committed it. You can have Puppet check the manifest using the `puppet parser validate` command:

```
t@ckbk:~$ puppet parser validate bootstrap.pp
Error: Could not parse for environment production: Syntax error at
 'File'; expected '}' at /home/thomas/bootstrap.pp:3
```

This is especially useful because a mistake anywhere in the manifest will stop Puppet from running on any node, even on nodes that don't use that particular part of the manifest. So checking in a bad manifest can cause Puppet to stop applying updates to production for some time, until the problem is discovered, and this could potentially have serious consequences. The best way to avoid this is to automate the syntax check, by using a precommit hook in your version control repo.

How to do it...

Follow these steps:

1. In your Puppet repo, create a new `hooks` directory:

   ```
   t@mylaptop:~/puppet$ mkdir hooks
   ```

2. Create the file `hooks/check_syntax.sh` with the following contents (based on a script by Puppet Labs):

```
#!/bin/sh

syntax_errors=0
error_msg=$(mktemp /tmp/error_msg.XXXXXX)

if git rev-parse --quiet --verify HEAD > /dev/null
then
    against=HEAD
else
    # Initial commit: diff against an empty tree object
    against=4b825dc642cb6eb9a060e54bf8d69288fbee4904
fi

# Get list of new/modified manifest and template files
  to check (in git index)
for indexfile in 'git diff-index --diff-filter=AM --
  name-only --cached $against | egrep '\.(pp|erb)''
do
    # Don't check empty files
    if [ 'git cat-file -s :0:$indexfile' -gt 0 ]
    then
        case $indexfile in
            *.pp )
                # Check puppet manifest syntax
                git cat-file blob :0:$indexfile |
                  puppet parser validate > $error_msg ;;
            *.erb )
                # Check ERB template syntax
                git cat-file blob :0:$indexfile |
                  erb -x -T - | ruby -c 2> $error_msg >
                    /dev/null ;;
        esac
        if [ "$?" -ne 0 ]
        then
            echo -n "$indexfile: "
            cat $error_msg
            syntax_errors='expr $syntax_errors + 1'
        fi
    fi
done

rm -f $error_msg

if [ "$syntax_errors" -ne 0 ]
```

```
then
    echo "Error: $syntax_errors syntax errors found,
      aborting commit."
    exit 1
fi
```

3. Set execute permission for the `hook` script with the following command:

```
t@mylaptop:~/puppet$ chmod a+x hooks/check_syntax.sh
```

4. Now either symlink or copy the script to the precommit hook in your hooks directory. If your Git repo is checked out in `~/puppet`, then create the symlink at `~/puppet/hooks/pre-commit` as follows:

```
t@mylaptop:~/puppet$ ln -s ~/puppet/hooks/check_syntax.sh
 .git/hooks/pre-commit
```

How it works...

The `check_syntax.sh` script will prevent you from committing any files with syntax errors when it is used as the pre-commit hook for Git:

```
t@mylaptop:~/puppet$ git commit -m "test commit"
Error: Could not parse for environment production: Syntax error at
  '}' at line 3
Error: Try 'puppet help parser validate' for usage
manifests/nodes.pp: Error: 1 syntax errors found, aborting commit.
```

If you add the `hooks` directory to your Git repo, anyone who has a checkout can copy the script into their local `hooks` directory to get this syntax checking behavior.

Pushing code around with Git

As we have already seen in the decentralized model, Git can be used to transfer files between machines using a combination of `ssh` and `ssh` keys. It can also be useful to have a Git hook do the same on each successful commit to the repository.

There exists a hook called post-commit that can be run after a successful commit to the repository. In this recipe, we'll create a hook that updates the code on our Puppet master with code from our Git repository on the Git server.

Getting ready

Follow these steps to get started:

1. Create an `ssh` key that can access your Puppet user on your Puppet master and install this key into the Git user's account on `git.example.com`:

    ```
    [git@git ~]$ ssh-keygen -f ~/.ssh/puppet_rsa
    Generating public/private rsa key pair.
    Your identification has been saved in /home/git/.ssh/puppet_rsa.
    Your public key has been saved in /home/git/.ssh/puppet_rsa.pub.
    Copy the public key into the authorized_keys file of the puppet
    user on your puppetmaster
    puppet@puppet:~/.ssh$ cat puppet_rsa.pub >>authorized_keys
    ```

2. Modify the Puppet account to allow the Git user to log in as follows:

    ```
    root@puppet:~# chsh puppet -s /bin/bash
    ```

How to do it...

Perform the following steps:

1. Now that the Git user can log in to the Puppet master as the Puppet user, modify the Git user's `ssh` configuration to use the newly created `ssh` key by default:

    ```
    [git@git ~]$ vim .ssh/config
    Host puppet.example.com
      IdentityFile ~/.ssh/puppet_rsa
    ```

2. Add the Puppet master as a remote location for the Puppet repository on the Git server with the following command:

    ```
    [git@git puppet.git]$ git remote add puppetmaster puppet@puppet.
    example.com:/etc/puppet/environments/puppet.git
    ```

3. On the Puppet master, move the `production` directory out of the way and check out your Puppet repository:

    ```
    root@puppet:~# chown -R puppet:puppet /etc/puppet/environments
    root@puppet:~# sudo -iu puppet
    puppet@puppet:~$ cd /etc/puppet/environments/
    puppet@puppet:/etc/puppet/environments$ mv production production.
    orig
    puppet@puppet:/etc/puppet/environments$ git clone git@git.example.
    com:repos/puppet.git
    ```

```
Cloning into 'puppet.git'...
remote: Counting objects: 63, done.
remote: Compressing objects: 100% (52/52), done.
remote: Total 63 (delta 10), reused 0 (delta 0)
Receiving objects: 100% (63/63), 9.51 KiB, done.
Resolving deltas: 100% (10/10), done.
```

4. Now we have a local bare repository on the Puppet server that we can push to, to remotely clone this into the `production` directory:

```
puppet@puppet:/etc/puppet/environments$ git clone puppet.git
production
Cloning into 'production'...
done.
```

5. Now perform a Git push from the Git server to the Puppet master:

```
[git@git ~]$ cd repos/puppet.git/
[git@git puppet.git]$ git push puppetmaster
Everything up-to-date
```

6. Create a post-commit file in the `hooks` directory of the repository on the Git server with the following contents:

```
[git@git puppet.git]$ vim hooks/post-commit
#!/bin/sh
git push puppetmaster
ssh puppet@puppet.example.com "cd /etc/puppet/environments/
production && git pull"
[git@git puppet.git]$ chmod 755 hooks/post-commit
```

7. Commit a change to the repository from your laptop and verify that the change is propagated to the Puppet master as follows:

```
t@mylaptop puppet$ vim README
t@mylaptop puppet$ git add README
t@mylaptop puppet$ git commit -m "Adding README"
[master 8148902] Adding README
 1 file changed, 4 deletions(-)
t@mylaptop puppet$ git push
X11 forwarding request failed on channel 0
Counting objects: 5, done.
Delta compression using up to 4 threads.
```

```
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 371 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: To puppet@puppet.example.com:/etc/puppet/environments/
puppet.git
remote:    377ed44..8148902  master -> master
remote: From /etc/puppet/environments/puppet
remote:    377ed44..8148902  master     -> origin/master
remote: Updating 377ed44..8148902
remote: Fast-forward
remote:  README |    4 ----
remote:  1 file changed, 4 deletions(-)
To git@git.example.com:repos/puppet.git
   377ed44..8148902  master -> master
```

How it works...

We created a bare repository on the Puppet master that we then use as a remote for the repository on `git.example.com` (remote repositories must be bare). We then clone that bare repository into the `production` directory. We add the bare repository on `puppet.example.com` as a remote to the bare repository on `git.example.com`. We then create a post-receive hook in the repository on `git.example.com`.

The hook issues a Git push to the Puppet master bare repository. We then update the `production` directory from the updated bare repository on the Puppet master. In the next section, we'll modify the hook to use branches.

Managing Environments with Git

Branches are a way of keeping several different tracks of development within a single source repository. Puppet environments are a lot like Git branches. You can have the same code with slight variations between branches, just as you can have different modules for different environments. In this section, we'll show how to use Git branches to define environments on the Puppet master.

Getting ready

In the previous section, we created a `production` directory that was based on the master branch; we'll remove that directory now:

```
puppet@puppet:/etc/puppet/environments$ mv production production.master
```

How to do it...

Modify the `post-receive` hook to accept a branch variable. The hook will use this variable to create a directory on the Puppet master as follows:

```
#!/bin/sh

read oldrev newrev refname
branch=${refname#*\/*\/}

git push puppetmaster $branch
ssh puppet@puppet.example.com "if [ ! -d
/etc/puppet/environments/$branch ]; then git clone
 /etc/puppet/environments/puppet.git
 /etc/puppet/environments/$branch; fi; cd
 /etc/puppet/environments/$branch; git checkout $branch; git pull"
```

Modify your `README` file again and push to the repository on `git.example.com`:

```
t@mylaptop puppet$ git add README
t@mylaptop puppet$ git commit -m "Adding README"
[master 539d9f8] Adding README
 1 file changed, 1 insertion(+)
t@mylaptop puppet$ git push
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 374 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: To puppet@puppet.example.com:/etc/puppet/environments/puppet.git
remote:    0d6b49f..539d9f8  master -> master
remote: Cloning into '/etc/puppet/environments/master'...
remote: done.
remote: Already on 'master'
remote: Already up-to-date.
To git@git.example.com:repos/puppet.git
   0d6b49f..539d9f8  master -> master
```

How it works...

The hook now reads in the `refname` and parses out the branch that is being updated. We use that branch variable to clone the repository into a new directory and check out the branch.

There's more...

Now when we want to create a new environment, we can create a new branch in the Git repository. The branch will create a directory on the Puppet master. Each branch of the Git repository represents an environment on the Puppet master:

1. Create the production branch as shown in the following command line:

```
t@mylaptop puppet$ git branch production
t@mylaptop puppet$ git checkout production
Switched to branch 'production'
```

2. Update the production branch and push to the Git server as follows:

```
t@mylaptop puppet$ vim README
t@mylaptop puppet$ git add README
t@mylaptop puppet$ git commit -m "Production Branch"
t@mylaptop puppet$ git push origin production
Counting objects: 7, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 372 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: To puppet@puppet.example.com:/etc/puppet/environments/
puppet.git
remote:    11db6e5..832f6a9  production -> production
remote: Cloning into '/etc/puppet/environments/production'...
remote: done.
remote: Switched to a new branch 'production'
remote: Branch production set up to track remote branch production
from origin.
remote: Already up-to-date.
To git@git.example.com:repos/puppet.git
   11db6e5..832f6a9  production -> production
```

Now whenever we create a new branch, a corresponding directory is created in our environment's directory. A one-to-one mapping is established between environments and branches.

3
Writing Better Manifests

"Measuring programming progress by lines of code is like measuring aircraft building progress by weight."

— Bill Gates

In this chapter, we will cover:

- Using arrays of resources
- Using resource defaults
- Using defined types
- Using tags
- Using run stages
- Using roles and profiles
- Passing parameters to classes
- Passing parameters from Hiera
- Writing reusable, cross-platform manifests
- Getting information about the environment
- Importing dynamic information
- Passing arguments to shell commands

Introduction

Your Puppet manifests are the living documentation for your entire infrastructure. Keeping them tidy and well organized is a great way to make it easier to maintain and understand. Puppet gives you a number of tools to do this, as follows:

- Arrays
- Defaults
- Defined types
- Dependencies
- Class parameters

We'll see how to use all of these and more. As you read through the chapter, try out the examples and look through your own manifests to see where these features might help you simplify and improve your Puppet code.

Using arrays of resources

Anything that you can do to a resource, you can do to an array of resources. Use this idea to refactor your manifests to make them shorter and clearer.

How to do it...

Here are the steps to refactor using arrays of resources:

1. Identify a class in your manifest where you have several instances of the same kind of resource, for example, packages:

```
package { 'sudo' : ensure => installed }
package { 'unzip' : ensure => installed }
package { 'locate' : ensure => installed }
package { 'lsof' : ensure => installed }
package { 'cron' : ensure => installed }
package { 'rubygems' : ensure => installed }
```

2. Group them together and replace them with a single package resource using an array:

```
package
{
  [ 'cron',
  'locate',
  'lsof',
  'rubygems',
  'sudo',
  'unzip' ]:
  ensure => installed,
}
```

How it works...

Most of Puppet's resource types can accept an array instead of a single name, and will create one instance for each of the elements in the array. All the parameters you provide for the resource (for example, `ensure => installed`) will be assigned to each of the new resource instances. This shorthand will only work when all the resources have the same attributes.

See also

- The *Iterating over multiple items* recipe in *Chapter 1*, *Puppet Language and Style*

Using resource defaults

A Puppet module is a group of related resources, usually grouped to configure a specific service. Within a module, you may define multiple resources; resource defaults allow you to specify the default attribute values for a resource. In this example, we'll show you how to specify a resource default for the `File` type.

How to do it...

To show you how to use resource defaults, we'll create an apache module. Within this module we will specify that the default owner and group are the `apache` user as follows:

1. Create an apache module and create a resource default for the `File` type:

```
    class apache {
      File {
        owner => 'apache',
        group => 'apache',
        mode => 0644,
      }
    }
```

2. Create html files within the `/var/www/html` directory:

   ```
   file {'/var/www/html/index.html':
     content => "<html><body><h1><a
       href='cookbook.html'>Cookbook!
       </a></h1></body></html>\n",
   }
   file {'/var/www/html/cookbook.html':
     content =>
       "<html><body><h2>PacktPub</h2></body></html>\n",
   }
   ```

3. Add this class to your default node definition, or use `puppet apply` to apply the module to your node. I will use the method we configured in the previous chapter, pushing our code to the Git repository and using a Git hook to have the code deployed to the Puppet master as follows:

   ```
   t@mylaptop ~/puppet $ git pull origin production
   From git.example.com:repos/puppet
    * branch            production -> FETCH_HEAD
   Already up-to-date.
   t@mylaptop ~/puppet $ cd modules
   t@mylaptop ~/puppet/modules $ mkdir -p apache/manifests
   t@mylaptop ~/puppet/modules $ vim apache/manifests/init.pp
   t@mylaptop ~/puppet/modules $ cd ..
   t@mylaptop ~/puppet $ vim manifests/site.pp
   t@mylaptop ~/puppet $ git status
   On branch production
   Changes not staged for commit:
   modified:   manifests/site.pp
   Untracked files:
   modules/apache/
   t@mylaptop ~/puppet $ git add manifests/site.pp modules/apache
   t@mylaptop ~/puppet $ git commit -m 'adding apache module'
   [production d639a86] adding apache module
    2 files changed, 14 insertions(+)
    create mode 100644 modules/apache/manifests/init.pp
   t@mylaptop ~/puppet $ git push origin production
   ```

```
Counting objects: 13, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (8/8), 885 bytes | 0 bytes/s, done.
Total 8 (delta 0), reused 0 (delta 0)
remote: To puppet@puppet.example.com:/etc/puppet/environments/
puppet.git
remote:    832f6a9..d639a86  production -> production
remote: Already on 'production'
remote: From /etc/puppet/environments/puppet
remote:    832f6a9..d639a86  production -> origin/production
remote: Updating 832f6a9..d639a86
remote: Fast-forward
remote:  manifests/site.pp                  |    1 +
remote:  modules/apache/manifests/init.pp |   13 +++++++++++++
remote:  2 files changed, 14 insertions(+)
remote:  create mode 100644 modules/apache/manifests/init.pp
To git@git.example.com:repos/puppet.git
   832f6a9..d639a86  production -> production
```

4. Apply the module to a node or run Puppet:

```
Notice: /Stage[main]/Apache/File[/var/www/html/cookbook.html]/
ensure: defined content as '{md5}493473fb5bde778ca93d034900348c5d'
Notice: /Stage[main]/Apache/File[/var/www/html/index.html]/ensure:
defined content as '{md5}184f22c181c5632b86ebf9a0370685b3'
Notice: Finished catalog run in 2.00 seconds
[root@hiera-test ~]# ls -l /var/www/html
total 8
-rw-r--r--. 1 apache apache 44 Sep 15 12:00 cookbook.html
-rw-r--r--. 1 apache apache 73 Sep 15 12:00 index.html
```

How it works...

The resource default we defined specifies the owner, group, and mode for all file resources within this class (also known as within this scope). Unless you specifically override a resource default, the value for an attribute will be taken from the default.

There's more...

You can specify resource defaults for any resource type. You can also specify resource defaults in `site.pp`. I find it useful to specify the default action for `Package` and `Service` resources as follows:

```
Package { ensure => 'installed' }
Service {
  hasrestart => true,
  enable     => true,
  ensure     => true,
}
```

With these defaults, whenever you specify a package, the package will be installed. Whenever you specify a service, the service will be started and enabled to run at boot. These are the usual reasons you specify packages and services, most of the time these defaults will do what you prefer and your code will be cleaner. When you need to disable a service, simply override the defaults.

Using defined types

In the previous example, we saw how to reduce redundant code by grouping identical resources into arrays. However, this technique is limited to resources where all the parameters are the same. When you have a set of resources that have some parameters in common, you need to use a defined type to group them together.

How to do it...

The following steps will show you how to create a definition:

1. Add the following code to your manifest:

```
define tmpfile() {
  file { "/tmp/${name}":
    content => "Hello, world\n",
  }
}
tmpfile { ['a', 'b', 'c']: }
```

2. Run Puppet:

```
[root@hiera-test ~]# vim tmp.pp
[root@hiera-test ~]# puppet apply tmp.pp
```

```
Notice: Compiled catalog for hiera-test.example.com in environment
production in 0.11 seconds
Notice: /Stage[main]/Main/Tmpfile[a]/File[/tmp/a]/ensure: defined
content as '{md5}a7966bf58e23583c9a5a4059383ff850'
Notice: /Stage[main]/Main/Tmpfile[b]/File[/tmp/b]/ensure: defined
content as '{md5}a7966bf58e23583c9a5a4059383ff850'
Notice: /Stage[main]/Main/Tmpfile[c]/File[/tmp/c]/ensure: defined
content as '{md5}a7966bf58e23583c9a5a4059383ff850'
Notice: Finished catalog run in 0.09 seconds
[root@hiera-test ~]# cat /tmp/{a,b,c}
Hello, world
Hello, world
Hello, world
```

How it works...

You can think of a defined type (introduced with the `define` keyword) as a cookie-cutter. It describes a pattern that Puppet can use to create lots of similar resources. Any time you declare a `tmpfile` instance in your manifest, Puppet will insert all the resources contained in the `tmpfile` definition.

In our example, the definition of `tmpfile` contains a single `file` resource whose content is `Hello, world\n` and whose path is `/tmp/${name}`. If you declared an instance of `tmpfile` with the name `foo`:

```
tmpfile { 'foo': }
```

Puppet will create a file with the path `/tmp/foo`. In other words, `${name}` in the definition will be replaced by the `name` of any actual instance that Puppet is asked to create. It's almost as though we created a new kind of resource: `tmpfile`, which has one parameter—its `name`.

Just like with regular resources, we don't have to pass just one title; as in the preceding example, we can provide an array of titles and Puppet will create as many resources as required.

A word on name, the namevar: Every resource you create must have a unique name, the namevar. This is different than the title, which is how puppet refers to the resource internally (although they are often the same).

There's more...

In the example, we created a definition where the only parameter that varies between instances is the `name` parameter. But we can add whatever parameters we want, so long as we declare them in the definition in parentheses after the `name` parameter, as follows:

```
define tmpfile($greeting) {
  file { "/tmp/${name}":
    content => $greeting,
  }
}
```

Next, pass values to them when we declare an instance of the resource:

```
tmpfile{ 'foo':
  greeting => "Good Morning\n",
}
```

You can declare multiple parameters as a comma-separated list:

```
define webapp($domain,$path,$platform) {
  ...
}
webapp { 'mywizzoapp':
  domain   => 'mywizzoapp.com',
  path     => '/var/www/apps/mywizzoapp',
  platform => 'Rails',
}
```

You can also declare default values for any parameters that aren't supplied, thus making them optional:

```
define tmpfile($greeting,$mode='0644') {
  ...
}
```

This is a powerful technique for abstracting out everything that's common to certain resources, and keeping it in one place so that you *don't repeat yourself*. In the preceding example, there might be many individual resources contained within `webapp`: packages, config files, source code checkouts, virtual hosts, and so on. But all of them are the same for every instance of `webapp` except the parameters we provide. These might be referenced in a template, for example, to set the domain for a virtual host.

See also

- The *Passing parameters to classes* recipe, in this chapter

Using tags

Sometimes one Puppet class needs to know about another or at least to know whether or not it's present. For example, a class that manages the firewall may need to know whether or not the node is a web server.

Puppet's `tagged` function will tell you whether a named class or resource is present in the catalog for this node. You can also apply arbitrary tags to a node or class and check for the presence of these tags. Tags are another metaparameter, similar to `require` and `notify` we introduced in *Chapter 1*, *Puppet Language and Style*. Metaparameters are used in the compilation of the Puppet catalog but are not an attribute of the resource to which they are attached.

How to do it...

To help you find out if you're running on a particular node or class of nodes all nodes are automatically tagged with the node name and the names of any classes they include. Here's an example that shows you how to use `tagged` to get this information:

1. Add the following code to your `site.pp` file (replacing `cookbook` with your machine's `hostname`):

```
node 'cookbook' {
  if tagged('cookbook') {
    notify { 'tagged cookbook': }
  }
}
```

2. Run Puppet:

```
root@cookbook:~# puppet agent -vt
Info: Caching catalog for cookbook
Info: Applying configuration version '1410848350'
Notice: tagged cookbook
Notice: Finished catalog run in 1.00 seconds
```

Nodes are also automatically tagged with the names of all the classes they include in addition to several other automatic tags. You can use `tagged` to find out what classes are included on the node.

You're not just limited to checking the tags automatically applied by Puppet. You can also add your own. To set an arbitrary tag on a node, use the `tag` function, as in the following example:

3. Modify your `site.pp` file as follows:

```
node 'cookbook' {
  tag('tagging')
  class {'tag_test': }
}
```

4. Add a `tag_test` module with the following `init.pp` (or be lazy and add the following definition to your `site.pp`):

```
class tag_test {
  if tagged('tagging') {
    notify { 'containing node/class was tagged.': }
  }
}
```

5. Run Puppet:

```
root@cookbook:~# puppet agent -vt
Info: Caching catalog for cookbook
Info: Applying configuration version '1410851300'
Notice: containing node/class was tagged.
Notice: Finished catalog run in 0.22 seconds
```

6. You can also use tags to determine which parts of the manifest to apply. If you use the `--tags` option on the Puppet command line, Puppet will apply only those classes or resources tagged with the specific tags you include. For example, we can define our `cookbook` class with two classes:

```
node cookbook {
  class {'first_class': }
  class {'second_class': }
}
class first_class {
  notify { 'First Class': }
}
```

```
    class second_class {
      notify {'Second Class': }
    }
```

7. Now when we run `puppet agent` on the `cookbook` node, we see both notifies:

```
root@cookbook:~# puppet agent -t
Notice: Second Class
Notice: First Class
Notice: Finished catalog run in 0.22 seconds
```

8. Now apply the `first_class` and `add --tags` function to the command line:

```
root@cookbook:~# puppet agent -t --tags first_class
Notice: First Class
Notice: Finished catalog run in 0.07 seconds
```

There's more...

You can use tags to create a collection of resources, and then make the collection a dependency for some other resource. For example, say some service depends on a config file that is built from a number of file snippets, as in the following example:

```
class firewall::service {
  service { 'firewall':
    ...
  }
  File <| tag == 'firewall-snippet' |> ~> Service['firewall']
}
class myapp {
  file { '/etc/firewall.d/myapp.conf':
    tag => 'firewall-snippet',
    ...
  }
}
```

Here, we've specified that the `firewall` service should be notified if any file resource tagged `firewall-snippet` is updated. All we need to do to add a `firewall` config snippet for any particular application or service is to tag it `firewall-snippet`, and Puppet will do the rest.

Although we could add a `notify => Service["firewall"]` function to each snippet resource if our definition of the `firewall` service were ever to change, we would have to hunt down and update all the snippets accordingly. The tag lets us encapsulate the logic in one place, making future maintenance and refactoring much easier.

What's `<| tag == 'firewall-snippet' |>` syntax? This is called a resource collector, and it's a way of specifying a group of resources by searching for some piece of data about them; in this case, the value of a tag. You can find out more about resource collectors and the `<| |>` operator (sometimes known as the spaceship operator) on the Puppet Labs website: `http://docs.puppetlabs.com/puppet/3/reference/lang_collectors.html`.

Using run stages

A common requirement is to apply a certain group of resources before other groups (for example, installing a package repository or a custom Ruby version), or after others (for example, deploying an application once its dependencies are installed). Puppet's run stages feature allows you to do this.

By default, all resources in your manifest are applied in a single stage named `main`. If you need a resource to be applied before all others, you can assign it to a new run stage that is specified to come before `main`. Similarly, you could define a run stage that comes after `main`. In fact, you can define as many run stages as you need and tell Puppet which order they should be applied in.

In this example, we'll use stages to ensure one class is applied first and another last.

How to do it...

Here are the steps to create an example of using run `stages`:

1. Create the file `modules/admin/manifests/stages.pp` with the following contents:

```
class admin::stages {
  stage { 'first': before => Stage['main'] }
  stage { 'last': require => Stage['main'] }
  class me_first {
    notify { 'This will be done first': }
  }
```

```
      class me_last {
        notify { 'This will be done last': }
      }
      class { 'me_first':
        stage => 'first',
      }
      class { 'me_last':
        stage => 'last',
      }
    }
```

2. Modify your `site.pp` file as follows:

```
    node 'cookbook' {
      class {'first_class': }
      class {'second_class': }
      include admin::stages
    }
```

3. Run Puppet:

```
root@cookbook:~# puppet agent -t
Info: Applying configuration version '1411019225'
Notice: This will be done first
Notice: Second Class
Notice: First Class
Notice: This will be done last
Notice: Finished catalog run in 0.43 seconds
```

How it works...

Let's examine this code in detail to see what's happening. First, we declare the run stages `first` and `last`, as follows:

```
stage { 'first': before => Stage['main'] }
stage { 'last': require => Stage['main'] }
```

For the `first` stage, we've specified that it should come before `main`. That is, every resource marked as being in the `first` stage will be applied before any resource in the `main` stage (the default stage).

The `last` stage requires the `main` stage, so no resource in the `last` stage can be applied until after every resource in the `main` stage.

We then declare some classes that we'll later assign to these run stages:

```
class me_first {
  notify { 'This will be done first': }
}
class me_last {
  notify { 'This will be done last': }
}
```

We can now put it all together and include these classes on the node, specifying the run stages for each as we do so:

```
class { 'me_first':
  stage => 'first',
}
class { 'me_last':
  stage => 'last',
}
```

Note that in the `class` declarations for `me_first` and `me_last`, we didn't have to specify that they take a `stage` parameter. The `stage` parameter is another metaparameter, which means it can be applied to any class or resource without having to be explicitly declared. When we ran `puppet agent` on our Puppet node, the notify from the `me_first` class was applied before the notifies from `first_class` and `second_class`. The notify from `me_last` was applied after the `main` stage, so it comes after the two notifies from `first_class` and `second_class`. If you run `puppet agent` multiple times, you will see that the notifies from `first_class` and `second_class` may not always appear in the same order but the `me_first` class will always come first and the `me_last` class will always come last.

There's more...

You can define as many run stages as you like, and set up any ordering for them. This can greatly simplify a complicated manifest that would otherwise require lots of explicit dependencies between resources. Beware of accidentally introducing dependency cycles, though; when you assign something to a run stage you're automatically making it dependent on everything in prior stages.

You may like to define your stages in the `site.pp` file instead, so that at the top level of the manifest, it's easy to see what stages are available.

Gary Larizza has written a helpful introduction to using run stages, with some real-world examples, on his website:

`http://garylarizza.com/blog/2011/03/11/using-run-stages-with-puppet/`

A caveat: many people don't like to use run stages, feeling that Puppet already provides sufficient resource ordering control, and that using run stages indiscriminately can make your code very hard to follow. The use of run stages should be kept to a minimum wherever possible. There are a few key examples where the use of stages creates less complexity. The most notable is when a resource modifies the system used to install packages on the system. It helps to have a package management stage that comes before the main stage. When packages are defined in the `main` (default) stage, your manifests can count on the updated package management configuration information being present. For instance, for a Yum-based system, you would create a `yumrepos` stage that comes before `main`. You can specify this dependency using chaining arrows as shown in the following code snippet:

```
stage {'yumrepos': }
Stage['yumrepos'] -> Stage['main']
```

We can then create a class that creates a Yum repository (`yumrepo`) resource and assign it to the `yumrepos` stage as follows:

```
class {'yums':
  stage => 'yumrepos',
}
class yums {
  notify {'always before the rest': }
  yumrepo {'testrepo':
    baseurl => 'file:///var/yum',
    ensure  => 'present',
  }
}
```

For Apt-based systems, the same example would be a stage where Apt sources are defined. The key with stages is to keep their definitions in your `site.pp` file where they are highly visible and to only use them sparingly where you can guarantee
that you will not introduce dependency cycles.

See also

- The *Using tags* recipe, in this chapter
- The *Drawing dependency graphs* recipe in *Chapter 10*, *Monitoring, Reporting, and Troubleshooting*

Using roles and profiles

Well organized Puppet manifests are easy to read; the purpose of a module should be evident in its name. The purpose of a node should be defined in a single class. This single class should include all classes that are required to perform that purpose. Craig Dunn wrote a post about such a classification system, which he dubbed "roles and profiles" (`http://www.craigdunn.org/2012/05/239/`). In this model, roles are the single purpose of a node, a node may only have one role, a role may contain more than one profile, and a profile contains all the resources related to a single service. In this example, we will create a web server role that uses several profiles.

How to do it...

We'll create two modules to store our roles and profiles. Roles will contain one or more profiles. Each role or profile will be defined as a subclass, such as `profile::base`

1. Decide on a naming strategy for your roles and profiles. In our example, we will create two modules, `roles` and `profiles` that will contain our roles and profiles respectively:

   ```
   $ puppet module generate thomas-profiles
   $ ln -s thomas-profiles profiles
   $ puppet module generate thomas-roles
   $ ln -s thomas-roles roles
   ```

2. Begin defining the constituent parts of our `webserver` role as profiles. To keep this example simple, we will create two profiles. First, a `base` profile to include our basic server configuration classes. Second, an `apache` class to install and configure the apache web server (`httpd`) as follows:

   ```
   $ vim profiles/manifests/base.pp
   class profiles::base {
     include base
   }
   $ vim profiles/manifests/apache.pp
   class profiles::apache {
     $apache = $::osfamily ? {
       'RedHat' => 'httpd',
       'Debian' => 'apache2',
       }
     service { "$apache":
   ```

```
        enable => true,
        ensure => true,
      }
      package { "$apache":
        ensure => 'installed',
      }
    }
```

3. Define a `roles::webserver` class for our `webserver` role as follows:

```
$ vim roles/manifests/webserver.pp
class roles::webserver {
  include profiles::apache
  include profiles::base
}
```

4. Apply the `roles::webserver` class to a node. In a centralized installation, you would use either an **External Node Classifier** (**ENC**) to apply the class to the node, or you would use Hiera to define the role:

```
node 'webtest' {
  include roles::webserver
}
```

How it works...

Breaking down the parts of the web server configuration into different profiles allows us to apply those parts independently. We created a base profile that we can expand to include all the resources we would like applied to all nodes. Our `roles::webserver` class simply includes the `base` and `apache` classes.

There's more...

As we'll see in the next section, we can pass parameters to classes to alter how they work. In our `roles::webserver` class, we can use the class instantiation syntax instead of `include`, and override it with `parameters` in the classes. For instance, to pass a parameter to the `base` class, we would use:

```
class {'profiles::base':
  parameter => 'newvalue'
}
```

where we previously used:

```
include profiles::base
```

In previous versions of this book, node and class inheritance were used to achieve a similar goal, code reuse. Node inheritance is deprecated in Puppet Version 3.7 and higher. Node and class inheritance should be avoided. Using roles and profiles achieves the same level of readability and is much easier to follow.

Passing parameters to classes

Sometimes it's very useful to parameterize some aspect of a class. For example, you might need to manage different versions of a `gem` package, and rather than making separate classes for each that differ only in the version number, you can pass in the version number as a parameter.

How to do it...

In this example, we'll create a definition that accepts parameters:

1. Declare the parameter as a part of the class definition:

   ```
   class eventmachine($version) {
     package { 'eventmachine':
       provider => gem,
       ensure   => $version,
     }
   }
   ```

2. Use the following syntax to include the class on a node:

   ```
   class { 'eventmachine':
     version => '1.0.3',
   }
   ```

How it works...

The class definition `class eventmachine($version) {` is just like a normal class definition except it specifies that the class takes one parameter: `$version`. Inside the class, we've defined a `package` resource:

```
package { 'eventmachine':
  provider => gem,
  ensure   => $version,
}
```

This is a `gem` package, and we're requesting to install version `$version`.

Include the class on a node, instead of the usual `include` syntax:

```
include eventmachine
```

On doing so, there will be a `class` statement:

```
class { 'eventmachine':
  version => '1.0.3',
}
```

This has the same effect but also sets a value for the parameter as `version`.

There's more...

You can specify multiple parameters for a class as:

```
class mysql($package, $socket, $port) {
```

Then supply them in the same way:

```
class { 'mysql':
  package => 'percona-server-server-5.5',
  socket  => '/var/run/mysqld/mysqld.sock',
  port    => '3306',
}
```

Specifying default values

You can also give default values for some of your parameters. When you include the class without setting a parameter, the default value will be used. For instance, if we created a `mysql` class with three parameters, we could provide default values for any or all of the parameters as shown in the code snippet:

```
class mysql($package, $socket, $port='3306') {
```

or all:

```
class mysql(
  package = "percona-server-server-5.5",
  socket  = '/var/run/mysqld/mysqld.sock',
  port    = '3306') {
```

Defaults allow you to use a default value and override that default where you need it.

Unlike a definition, only one instance of a parameterized class can exist on a node. So where you need to have several different instances of the resource, use `define` instead.

Passing parameters from Hiera

Like the parameter `defaults` we introduced in the previous chapter, Hiera may be used to provide default values to classes. This feature requires Puppet Version 3 and higher.

Getting ready

Install and configure `hiera` as we did in *Chapter 2*, *Puppet Infrastructure*. Create a global or common `yaml` file; this will serve as the default for all values.

How to do it...

1. Create a class with parameters and no default values:

   ```
   t@mylaptop ~/puppet $ mkdir -p modules/mysql/manifests t@mylaptop
   ~/puppet $ vim modules/mysql/manifests/init.pp
   class mysql ( $port, $socket, $package ) {
     notify {"Port: $port Socket: $socket Package: $package": }
   }
   ```

2. Update your common `.yaml` file in Hiera with the default values for the `mysql` class:

   ```
   ---
   mysql::port: 3306
   mysql::package: 'mysql-server'
   mysql::socket: '/var/lib/mysql/mysql.sock'
   ```

   Apply the class to a node, you can add the mysql class to your default node for now.

   ```
   node default {
     class {'mysql': }
   }
   ```

3. Run `puppet agent` and verify the output:

   ```
   [root@hiera-test ~]# puppet agent -t
   Info: Caching catalog for hiera-test.example.com
   Info: Applying configuration version '1411182251'
   Notice: Port: 3306 Socket: /var/lib/mysql/mysql.sock Package:
   mysql-server
   ```

```
Notice: /Stage[main]/Mysql/Notify[Port: 3306 Socket: /var/lib/
mysql/mysql.sock Package: mysql-server]/message: defined 'message'
as 'Port: 3306 Socket: /var/lib/mysql/mysql.sock Package: mysql-
server'
Notice: Finished catalog run in 1.75 seconds
```

How it works...

When we instantiate the `mysql` class in our manifest, we provided no values for any of the attributes. Puppet knows to look for a value in Hiera that matches `class_name::parameter_name:` or `::class_name::parameter_name:`.

When Puppet finds a value, it uses it as the parameter for the class. If Puppet fails to find a value in Hiera and no default is defined, a catalog failure will result in the following command line:

```
Error: Could not retrieve catalog from remote server: Error 400 on
SERVER: Must pass package to Class[Mysql] at /etc/puppet/environments/
production/manifests/site.pp:6 on node hiera-test.example.com
```

This error indicates that Puppet would like a value for the parameter `package`.

There's more...

You can define a Hiera hierarchy and supply different values for parameters based on facts. You could, for instance, have `%{::osfamily}` in your hierarchy and have different `yaml` files based on the `osfamily` parameter (RedHat, Suse, and Debian).

Writing reusable, cross-platform manifests

Every system administrator dreams of a unified, homogeneous infrastructure of identical machines all running the same version of the same OS. As in other areas of life, however, the reality is often messy and doesn't conform to the plan.

You are probably responsible for a bunch of assorted servers of varying age and architecture running different kernels from different OS distributions, often scattered across different data centers and ISPs.

This situation should strike terror into the hearts of the sysadmins of the SSH in a `for` loop persuasion, because executing the same commands on every server can have different, unpredictable, and even dangerous results.

We should certainly strive to bring older servers up to date and get working as far as possible on a single reference platform to make administration simpler, cheaper, and more reliable. But until we get there, Puppet makes coping with heterogeneous environments slightly easier.

How to do it...

Here are some examples of how to make your manifests more portable:

1. Where you need to apply the same manifest to servers with different OS distributions, the main differences will probably be the names of packages and services, and the location of config files. Try to capture all these differences into a single class by using selectors to set global variables:

   ```
   $ssh_service = $::operatingsystem? {
     /Ubuntu|Debian/ => 'ssh',
     default         => 'sshd',
   }
   ```

   You needn't worry about the differences in any other part of the manifest; when you refer to something, use the variable with confidence that it will point to the right thing in each environment:

   ```
   service { $ssh_service:
     ensure => running,
   }
   ```

2. Often we need to cope with mixed architectures; this can affect the paths to shared libraries, and also may require different versions of packages. Again, try to encapsulate all the required settings in a single architecture class that sets global variables:

   ```
   $libdir = $::architecture ? {
     /amd64|x86_64/    => '/usr/lib64',
     default => '/usr/lib',
   }
   ```

   Then you can use these wherever an architecture-dependent value is required in your manifests or even in templates:

   ```
   ; php.ini
   [PHP]
   ; Directory in which the loadable extensions (modules) reside.
   extension_dir = <%= @libdir %>/php/modules
   ```

How it works...

The advantage of this approach (which could be called top-down) is that you only need to make your choices once. The alternative, bottom-up approach would be to have a selector or `case` statement everywhere a setting is used:

```
service { $::operatingsystem? {
  /Ubuntu|Debian/ => 'ssh',
  default         => 'sshd' }:
  ensure => running,
}
```

This not only results in lots of duplication, but makes the code harder to read. And when a new operating system is added to the mix, you'll need to make changes throughout the whole manifest, instead of just in one place.

There's more...

If you are writing a module for public distribution (for example, on Puppet Forge), making your module as cross-platform as possible will make it more valuable to the community. As far as you can, test it on many different distributions, platforms, and architectures, and add the appropriate variables so that it works everywhere.

If you use a public module and adapt it to your own environment, consider updating the public version with your changes if you think they might be helpful to other people.

Even if you are not thinking of publishing a module, bear in mind that it may be in production use for a long time and may have to adapt to many changes in the environment. If it's designed to cope with this from the start, it'll make life easier for you or whoever ends up maintaining your code.

> *"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live."*
>
> — *Dave Carhart*

See also

- The *Using public modules* recipe in *Chapter 7*, *Managing Applications*
- The *Configuring Hiera* recipe in *Chapter 2*, *Puppet Infrastructure*

Getting information about the environment

Often in a Puppet manifest, you need to know some local information about the machine you're on. Facter is the tool that accompanies Puppet to provide a standard way of getting information (facts) from the environment about things such as these:

- Operating system
- Memory size
- Architecture
- Processor count

To see a complete list of the facts available on your system, run:

```
$ sudo facter
architecture => amd64
augeasversion => 0.10.0
domain => compute-1.internal
ec2_ami_id => ami-137bcf7a
ec2_ami_launch_index => 0
```

While it can be handy to get this information from the command line, the real power of Facter lies in being able to access these facts in your Puppet manifests.

Some modules define their own facts; to see any facts that have been defined locally, add the `-p (pluginsync)` option to facter as follows:

```
$ sudo facter -p
```

How to do it...

Here's an example of using Facter facts in a manifest:

1. Reference a Facter fact in your manifest like any other variable. Facts are global variables in Puppet, so they should be prefixed with a double colon (`::`), as in the following code snippet:

   ```
   notify { "This is $::operatingsystem version
   $::operatingsystemrelease, on $::architecture architecture, kernel
   version $::kernelversion": }
   ```

2. When Puppet runs, it will fill in the appropriate values for the current node:

```
[root@hiera-test ~]# puppet agent -t
...
Info: Applying configuration version '1411275985'Notice: This is
RedHat version 6.5, on x86_64 architecture, kernel version 2.6.32
...
Notice: Finished catalog run in 0.40 seconds
```

How it works...

Facter provides a standard way for manifests to get information about the nodes to which they are applied. When you refer to a fact in a manifest, Puppet will query Facter to get the current value and insert it into the manifest. Facter facts are top scope variables.

Always refer to facts with leading double colons to ensure that you are using the fact and not a local variable:

`$::hostname` NOT `$hostname`

There's more...

You can also use facts in ERB templates. For example, you might want to insert the node's hostname into a file, or change a configuration setting for an application based on the memory size of the node. When you use fact names in templates, remember that they don't need a dollar sign because this is Ruby, not Puppet:

```
$KLogPath <%= case @kernelversion when '2.6.31' then
'/var/run/rsyslog/kmsg' else '/proc/kmsg' end %>
```

When referring to facts, use the @ syntax. Variables that are defined at the same scope as the function call to template can also be referenced with the @ syntax. Out of scope variables should use the `scope` function. For example, to reference the `mysql::port` variable we defined earlier in the `mysql` modules, use the following:

```
MySQL Port = <%= scope['::mysql::port'] %>
```

Applying this template results in the following file:

```
[root@hiera-test ~]# puppet agent -t
...
Info: Caching catalog for hiera-test.example.com
Notice: /Stage[main]/Erb/File[/tmp/template-test]/ensure: defined content
as '{md5}96edacaf9747093f73084252c7ca7e67'
Notice: Finished catalog run in 0.41 seconds [root@hiera-test ~]# cat /
tmp/template-test
MySQL Port = 3306
```

See also

- The *Creating custom facts* recipe in *Chapter 9*, *External Tools and the Puppet Ecosystem*

Importing dynamic information

Even though some system administrators like to wall themselves off from the rest of the office using piles of old printers, we all need to exchange information with other departments from time to time. For example, you may want to insert data into your Puppet manifests that is derived from some outside source. The generate function is ideal for this. Functions are executed on the machine compiling the catalog (the master for centralized deployments); an example like that shown here will only work in a masterless configuration.

Getting ready

Follow these steps to prepare to run the example:

1. Create the script `/usr/local/bin/message.rb` with the following contents:

   ```
   #!/usr/bin/env ruby
   puts "This runs on the master if you are centralized"
   ```

2. Make the script executable:

   ```
   $ sudo chmod a+x /usr/local/bin/message.rb
   ```

How to do it...

This example calls the external script we created previously and gets its output:

1. Create a `message.pp` manifest containing the following:

   ```
   $message = generate('/usr/local/bin/message.rb')
   notify { $message: }
   ```

2. Run Puppet:

   ```
   $ puppet apply message.pp
   ...
   Notice: /Stage[main]/Main/Notify[This runs on the master if you
   are centralized
   ]/message: defined 'message' as 'This runs on the master if you
   are centralized
   ```

How it works...

The `generate` function runs the specified script or program and returns the result, in this case, a cheerful message from Ruby.

This isn't terribly useful as it stands but you get the idea. Anything a script can do, print, fetch, or calculate, for example, the results of a database query, can be brought into your manifest using `generate`. You can also, of course, run standard UNIX utilities such as `cat` and `grep`.

There's more...

If you need to pass arguments to the executable called by generate, add them as extra arguments to the function call:

```
$message = generate('/bin/cat', '/etc/motd')
```

Puppet will try to protect you from malicious shell calls by restricting the characters you can use in a call to generate, so shell pipes and redirection aren't allowed, for example. The simplest and safest thing to do is to put all your logic into a script and then call that script.

See also

- The *Creating custom facts* recipe in *Chapter 9*, *External Tools and the Puppet Ecosystem*
- The *Configuring Hiera* recipe in *Chapter 2*, *Puppet Infrastructure*

Passing arguments to shell commands

If you want to insert values into a command line (to be run by an `exec` resource, for example), they often need to be quoted, especially if they contain spaces. The `shellquote` function will take any number of arguments, including arrays, and quote each of the arguments and return them all as a space-separated string that you can pass to commands.

In this example, we would like to set up an `exec` resource that will rename a file; but both the source and the target name contain spaces, so they need to be correctly quoted in the command line.

How to do it...

Here's an example of using the `shellquote` function:

1. Create a `shellquote.pp` manifest with the following command:

   ```
   $source = 'Hello Jerry'
   $target = 'Hello... Newman'
   $argstring = shellquote($source, $target)
   $command = "/bin/mv ${argstring}"
   notify { $command: }
   ```

2. Run Puppet:

   ```
   $ puppet apply shellquote.pp
   ...
   Notice: /bin/mv "Hello Jerry" "Hello... Newman"
   Notice: /Stage[main]/Main/Notify[/bin/mv "Hello Jerry" "Hello...
   Newman"]/message: defined 'message' as '/bin/mv "Hello Jerry"
   "Hello... Newman"'
   ```

How it works...

First we define the `$source` and `$target` variables, which are the two filenames we want to use in the command line:

```
$source = 'Hello Jerry'
$target = 'Hello... Newman'
```

Then we call `shellquote` to concatenate these variables into a quoted, space-separated string as follows:

```
$argstring = shellquote($source, $target)
```

Then we put together the final command line:

```
$command = "/bin/mv ${argstring}"
```

The result will be:

```
/bin/mv "Hello Jerry" "Hello... Newman"
```

This command line can now be run with an exec resource. What would happen if we didn't use `shellquote`?

```
$source = 'Hello Jerry'
$target = 'Hello... Newman'
$command = "/bin/mv ${source} ${target}"
notify { $command: }

Notice: /bin/mv Hello Jerry Hello... Newman
```

This won't work because `mv` expects space-separated arguments, so it will interpret this as a request to move three files `Hello`, `Jerry`, and `Hello...` into a directory named `Newman`, which probably isn't what we want.

4
Working with Files and Packages

"A writer has the duty to be good, not lousy; true, not false; lively, not dull; accurate, not full of error."

— E.B. White

In this chapter, we will cover the following recipes:

- Making quick edits to config files
- Editing INI style files with puppetlabs-inifile
- Using Augeas to reliably edit config files
- Building config files using snippets
- Using ERB templates
- Using array iteration in templates
- Using EPP templates
- Using GnuPG to encrypt secrets
- Installing packages from a third-party repository
- Comparing package versions

Introduction

In this chapter, we'll see how to make small edits to files, how to make larger changes in a structured way using the **Augeas** tool, how to construct files from concatenated snippets, and how to generate files from templates. We'll also learn how to install packages from additional repositories, and how to manage those repositories. In addition, we'll see how to store and decrypt secret data with Puppet.

Making quick edits to config files

When you need to have Puppet change a particular setting in a config file, it's common to simply deploy the whole file with Puppet. This isn't always possible, though; especially if it's a file that several different parts of your Puppet manifest may need to modify.

What would be useful is a simple recipe to add a line to a config file if it's not already present, for example, adding a module name to `/etc/modules` to tell the kernel to load that module at boot. There are several ways to do this, the simplest is to use the `file_line` type provided by the `puppetlabs-stdlib` module. In this example, we install the `stdlib` module and use this type to append a line to a text file.

Getting ready

Install the `puppetlabs-stdlib` module using puppet:

```
t@mylaptop ~ $ puppet module install puppetlabs-stdlib
Notice: Preparing to install into /home/thomas/.puppet/modules ...
Notice: Downloading from https://forgeapi.puppetlabs.com ...
Notice: Installing -- do not interrupt ...
/home/thomas/.puppet/modules
└── puppetlabs-stdlib (v4.5.1)
```

This installs the module from the forge into my user's puppet directory; to install into the system directory, run the command as root or use `sudo`. For the purpose of this example, we'll continue working as our own user.

How to do it...

Using the `file_line` resource type, we can ensure that a line exists or is absent in a config file. Using `file_line` we can quickly make edits to files without controlling the entire file.

1. Create a manifest named `oneline.pp` that will use `file_line` on a file in `/tmp`:

   ```
   file {'/tmp/cookbook':
     ensure => 'file',
   }
   file_line {'cookbook-hello':
   ```

```
    path    => '/tmp/cookbook',
    line    => 'Hello World!',
    require => File['/tmp/cookbook'],
  }
```

2. Run `puppet apply` on the `oneline.pp` manifest:

```
t@mylaptop ~/.puppet/manifests $ puppet apply oneline.pp
Notice: Compiled catalog for mylaptop in environment production in 0.39 seconds
Notice: /Stage[main]/Main/File[/tmp/cookbook]/ensure: created
Notice: /Stage[main]/Main/File_line[cookbook-hello]/ensure: created
Notice: Finished catalog run in 0.02 seconds
```

3. Now verify that `/tmp/cookbook` contains the line we defined:

```
t@mylaptop ~/.puppet/manifests $ cat /tmp/cookbook
Hello World!
```

How it works...

We installed the `puppetlabs-stdlib` module into the default module path for Puppet, so when we ran `puppet apply`, Puppet knew where to find the `file_line` type definition. Puppet then created the `/tmp/cookbook` file if it didn't exist. The line `Hello World!` was not found in the file, so Puppet added the line to the file.

There's more...

We can define more instances of `file_line` and add more lines to the file; we can have multiple resources modifying a single file.

Modify the `oneline.pp` file and add another `file_line` resource:

```
file {'/tmp/cookbook':
  ensure => 'file',
}
file_line {'cookbook-hello':
  path    => '/tmp/cookbook',
  line    => 'Hello World!',
  require => File['/tmp/cookbook'],
}
file_line {'cookbook-goodbye':
  path    => '/tmp/cookbook',
  line    => 'So long, and thanks for all the fish.',
  require => File['/tmp/cookbook'],
}
```

Now apply the manifest again and verify whether the new line is appended to the file:

```
t@mylaptop ~/.puppet/manifests $ puppet apply oneline.pp
Notice: Compiled catalog for mylaptop in environment production in 0.36 seconds
Notice: /Stage[main]/Main/File_line[cookbook-goodbye]/ensure: created
Notice: Finished catalog run in 0.02 seconds
t@mylaptop ~/.puppet/manifests $ cat /tmp/cookbook
Hello World!
So long, and thanks for all the fish.
```

The `file_line` type also supports pattern matching and line removal as we'll show you in the following example:

```
  file {'/tmp/cookbook':
    ensure => 'file',
  }
  file_line {'cookbook-remove':
    ensure  => 'absent',
    path    => '/tmp/cookbook',
    line    => 'Hello World!',
    require => File['/tmp/cookbook'],
  }
  file_line {'cookbook-match':
    path    => '/tmp/cookbook',
    line    => 'Oh freddled gruntbuggly, thanks for all the
      fish.',
    match   => 'fish.$',
    require => File['/tmp/cookbook'],
  }
```

Verify the contents of `/tmp/cookbook` before your Puppet run:

```
t@mylaptop ~/.puppet/manifests $ cat /tmp/cookbook
Hello World!
So long, and thanks for all the fish.
```

Apply the updated manifest:

```
t@mylaptop ~/.puppet/manifests $ puppet apply oneline.pp
Notice: Compiled catalog for mylaptop in environment production in 0.30 
seconds
Notice: /Stage[main]/Main/File_line[cookbook-match]/ensure: created
Notice: /Stage[main]/Main/File_line[cookbook-remove]/ensure: removed
Notice: Finished catalog run in 0.02 seconds
```

Verify that the line has been removed and the goodbye line has been replaced:

```
t@mylaptop ~/.puppet/manifests $ cat /tmp/cookbook 
Oh freddled gruntbuggly, thanks for all the fish.
```

Editing files with `file_line` works well if the file is unstructured. Structured files may have similar lines in different sections that have different meanings. In the next section, we'll show you how to deal with one particular type of structured file, a file using **INI syntax**.

Editing INI style files with puppetlabs-inifile

INI files are used throughout many systems, Puppet uses INI syntax for the `puppet.conf` file. The `puppetlabs-inifile` module creates two types, `ini_setting` and `ini_subsetting`, which can be used to edit INI style files.

Getting ready

Install the module from the forge as follows:

```
t@mylaptop ~ $ puppet module install puppetlabs-inifile
Notice: Preparing to install into /home/tuphill/.puppet/modules ...
Notice: Downloading from https://forgeapi.puppetlabs.com ...
Notice: Installing -- do not interrupt ...
/home/tuphill/.puppet/modules
└── puppetlabs-inifile (v1.1.3)
```

How to do it...

In this example, we will create a `/tmp/server.conf` file and ensure that the `server_true` setting is set in that file:

1. Create an `initest.pp` manifest with the following contents:

```
ini_setting {'server_true':
  path    => '/tmp/server.conf',
  section => 'main',
  setting => 'server',
  value   => 'true',
}
```

2. Apply the manifest:

```
t@mylaptop ~/.puppet/manifests $ puppet apply initest.pp
Notice: Compiled catalog for burnaby in environment production in
0.14 seconds
Notice: /Stage[main]/Main/Ini_setting[server_true]/ensure: created
Notice: Finished catalog run in 0.02 seconds
```

3. Verify the contents of the `/tmp/server.conf` file:

```
t@mylaptop ~/.puppet/manifests $ cat /tmp/server.conf

[main]
server = true
```

How it works...

The `inifile` module defines two types, `ini_setting` and `ini_subsetting`. Our manifest defines an `ini_setting` resource that creates a server = true setting within the main section of the `ini` file. In our case, the file didn't exist, so Puppet created the file, then created the `main` section, and finally added the setting to the `main` section.

There's more...

Using `ini_subsetting`, you can have several resources added to a setting. For instance, our `server.conf` file has a server's line, we could have each node append its own hostname to a server's line. Add the following to the end of the `initest.pp` file:

```
ini_subsetting {'server_name':
  path    => '/tmp/server.conf',
  section => 'main',
  setting => 'server_host',
  subsetting => "$hostname",
}
```

Apply the manifest:

```
t@mylaptop ~/.puppet/manifests $ puppet apply initest.pp
Notice: Compiled catalog for mylaptop in environment production in 0.34
seconds
Notice: /Stage[main]/Main/Ini_subsetting[server_name]/ensure: created
Notice: Finished catalog run in 0.02 seconds
t@mylaptop ~/.puppet/manifests $ cat /tmp/server.conf
[main]
server = true
server_host = mylaptop
```

Now temporarily change your hostname and rerun Puppet:

```
t@mylaptop ~/.puppet/manifests $ sudo hostname inihost
t@mylaptop ~/.puppet/manifests $ puppet apply initest.pp
Notice: Compiled catalog for inihost in environment production in 0.43
seconds
Notice: /Stage[main]/Main/Ini_subsetting[server_name]/ensure: created
Notice: Finished catalog run in 0.02 seconds
t@mylaptop ~/.puppet/manifests $ cat /tmp/server.conf
[main]
server = true
server_host = mylaptop inihost
```

When working with INI syntax files, using the `inifile` module is an excellent choice.

If your configuration files are not in INI syntax, another tool, Augeas, can be used. In the following section, we will use `augeas` to modify files.

Using Augeas to reliably edit config files

Sometimes it seems like every application has its own subtly different config file format, and writing regular expressions to parse and modify all of them can be a tiresome business.

Thankfully, Augeas is here to help. Augeas is a system that aims to simplify working with different config file formats by presenting them all as a simple tree of values. Puppet's Augeas support allows you to create `augeas` resources that can make the required config changes intelligently and automatically.

How to do it...

Follow these steps to create an example `augeas` resource:

1. Modify your `base` module as follows:

```
class base {
  augeas { 'enable-ip-forwarding':
    incl    => '/etc/sysctl.conf',
    lens    => 'Sysctl.lns',
    changes => ['set net.ipv4.ip_forward 1'],
  }
}
```

2. Run Puppet:

```
[root@cookbook ~]# puppet agent -t
Info: Applying configuration version '1412130479'
Notice: Augeas[enable-ip-forwarding](provider=augeas):
--- /etc/sysctl.conf      2014-09-04 03:41:09.000000000 -0400
+++ /etc/sysctl.conf.augnew      2014-09-30 22:28:03.503000039
-0400
@@ -4,7 +4,7 @@
 # sysctl.conf(5) for more details.

 # Controls IP packet forwarding
-net.ipv4.ip_forward = 0
+net.ipv4.ip_forward = 1

 # Controls source route verification
 net.ipv4.conf.default.rp_filter = 1
```

```
Notice: /Stage[main]/Base/Augeas[enable-ip-forwarding]/returns:
executed successfully

Notice: Finished catalog run in 2.27 seconds
```

3. Check whether the setting has been correctly applied:

```
[root@cookbook ~]# sysctl -p |grep ip_forward

net.ipv4.ip_forward = 1
```

How it works...

We declare an `augeas` resource named `enable-ip-forwarding`:

```
augeas { 'enable-ip-forwarding':
```

We specify that we want to make changes in the file `/etc/sysctl.conf`:

```
incl => '/etc/sysctl.conf',
```

Next we specify the lens to use on this file. Augeas uses files called lenses to translate a configuration file into an object representation. Augeas ships with several lenses, they are located in `/usr/share/augeas/lenses` by default. When specifying the lens in an `augeas` resource, the name of the lens is capitalized and has the `.lns` suffix. In this case, we will specify the `Sysctl` lens as follows:

```
lens => 'Sysctl.lns',
```

The `changes` parameter specifies the changes we want to make. Its value is an array, because we can supply several changes at once. In this example, there is only change, so the value is an array of one element:

```
changes => ['set net.ipv4.ip_forward 1'],
```

In general, Augeas changes take the following form:

```
set <parameter> <value>
```

In this case, the setting will be translated into a line like this in `/etc/sysctl.conf`:

```
net.ipv4.ip_forward=1
```

There's more...

I've chosen `/etc/sysctl.conf` as the example because it can contain a wide variety of kernel settings and you may want to change these settings for all sorts of different purposes and in different Puppet classes. You might want to enable IP forwarding, as in the example, for a router class but you might also want to tune the value of `net.core.somaxconn` for a load-balancer class.

This means that simply puppetizing the `/etc/sysctl.conf` file and distributing it as a text file won't work because you might have several different and conflicting versions depending on the setting you want to modify. Augeas is the right solution here because you can define `augeas` resources in different places, which modify the same file and they won't conflict.

For more information about using Puppet and Augeas, see the page on the Puppet Labs website `http://projects.puppetlabs.com/projects/1/wiki/Puppet_Augeas`.

Another project that uses Augeas is **Augeasproviders**. Augeasproviders uses Augeas to define several types. One of these types is `sysctl`, using this type you can make sysctl changes without knowing how to write the changes in Augeas. More information is available on the forge at `https://forge.puppetlabs.com/domcleal/augeasproviders`.

Learning how to use Augeas can be a little confusing at first. Augeas provides a command line tool, `augtool`, which can be used to get acquainted with making changes in Augeas.

Building config files using snippets

Sometimes you can't deploy a whole config file in one piece, yet making line by line edits isn't enough. Often, you need to build a config file from various bits of configuration managed by different classes. You may run into a situation where local information needs to be imported into the file as well. In this example, we'll build a config file using a local file as well as snippets defined in our manifests.

Getting ready

Although it's possible to create our own system to build files from pieces, we'll use the puppetlabs supported `concat` module. We will start by installing the `concat` module, in a previous example we installed the module to our local machine. In this example, we'll modify the Puppet server configuration and download the module to the Puppet server.

In your Git repository create an `environment.conf` file with the following contents:

```
modulepath = public:modules
manifest = manifests/site.pp
```

Create the public directory and download the module into that directory as follows:

```
t@mylaptop ~/puppet $ mkdir public && cd public
t@mylaptop ~/puppet/public $ puppet module install puppetlabs-concat
--modulepath=.
Notice: Preparing to install into /home/thomas/puppet/public ...
Notice: Downloading from https://forgeapi.puppetlabs.com ...
Notice: Installing -- do not interrupt ...
```

```
/home/thomas/puppet/public
└─┬ puppetlabs-concat (v1.1.1)
  └── puppetlabs-stdlib (v4.3.2)
```

Now add the new modules to our Git repository:

```
t@mylaptop ~/puppet/public $ git add .
t@mylaptop ~/puppet/public $ git commit -m "adding concat"
[production 50c6fca] adding concat
 407 files changed, 20089 insertions(+)
```

Then push to our Git server:

```
t@mylaptop ~/puppet/public $ git push origin production
```

How to do it...

Now that we have the `concat` module available on our server, we can create a `concat` container resource in our `base` module:

```
concat {'hosts.allow':
  path => '/etc/hosts.allow',
  mode => 0644
}
```

Create a `concat::fragment` module for the header of the new file:

```
concat::fragment {'hosts.allow header':
  target  => 'hosts.allow',
  content => "# File managed by puppet\n",
  order   => '01'
}
```

Create a `concat::fragment` that includes a local file:

```
concat::fragment {'hosts.allow local':
  target => 'hosts.allow',
  source => '/etc/hosts.allow.local',
  order  => '10',
}
```

Create a `concat::fragment` module that will go at the end of the file:

```
concat::fragment {'hosts.allow tftp':
  target  => 'hosts.allow',
  content => "in.ftpd: .example.com\n",
  order   => '50',
}
```

On the node, create `/etc/hosts.allow.local` with the following contents:

```
    in.tftpd: .example.com
```

Run Puppet to have the file created:

```
[root@cookbook ~]# puppet agent -t
Info: Caching catalog for cookbook.example.com
Info: Applying configuration version '1412138600'
Notice: /Stage[main]/Base/Concat[hosts.allow]/File[hosts.allow]/ensure:
defined content as '{md5}b151c8bbc32c505f1c4a98b487f7d249'
Notice: Finished catalog run in 0.29 seconds
```

Verify the contents of the new file as:

```
[root@cookbook ~]# cat /etc/hosts.allow
# File managed by puppet
in.tftpd: .example.com
in.ftpd: .example.com
```

How it works...

The `concat` resource defines a container that will hold all the subsequent `concat::fragment` resources. Each `concat::fragment` resource references the `concat` resource as the target. Each `concat::fragment` also includes an `order` attribute. The `order` attribute is used to specify the order in which the fragments are added to the final file. Our `/etc/hosts.allow` file is built with the header line, the contents of the local file, and finally the `in.tftpd` line we defined.

Using ERB templates

While you can deploy config files easily with Puppet as simple text files, templates are much more powerful. A template file can do calculations, execute Ruby code, or reference the values of variables from your Puppet manifests. Anywhere you might deploy a text file using Puppet, you can use a template instead.

In the simplest case, a template can just be a static text file. More usefully, you can insert variables into it using the ERB (embedded Ruby) syntax. For example:

```
    <%= @name %>, this is a very large drink.
```

If the template is used in a context where the variable `$name` contains `Zaphod Beeblebrox`, the template will evaluate to:

```
Zaphod Beeblebrox, this is a very large drink.
```

This simple technique is very useful to generate lots of files that only differ in the values of one or two variables, for example, virtual hosts, and for inserting values into a script such as database names and passwords.

How to do it...

In this example, we'll use an ERB template to insert a password into a backup script:

1. Create the file `modules/admin/templates/backup-mysql.sh.erb` with the following contents:

   ```
   #!/bin/sh
   /usr/bin/mysqldump -uroot \
     -p<%= @mysql_password %> \
     --all-databases | \
     /bin/gzip > /backup/mysql/all-databases.sql.gz
   ```

2. Modify your `site.pp` file as follows:

   ```
   node 'cookbook' {
     $mysql_password = 'secret'
     file { '/usr/local/bin/backup-mysql':
       content => template('admin/backup-mysql.sh.erb'),
       mode    => '0755',
     }
   }
   ```

3. Run Puppet:

   ```
   [root@cookbook ~]# puppet agent -t
   Info: Caching catalog for cookbook.example.com
   Info: Applying configuration version '1412140971'
   Notice: /Stage[main]/Main/Node[cookbook]/File[/usr/local/
   bin/backup-mysql]/ensure: defined content as '{md5}
   c12af56559ef36529975d568ff52dca5'
   Notice: Finished catalog run in 0.31 seconds
   ```

4. Check whether Puppet has correctly inserted the password into the template:

   ```
   [root@cookbook ~]# cat /usr/local/bin/backup-mysql
   #!/bin/sh
   /usr/bin/mysqldump -uroot \
   ```

```
        -psecret \
        --all-databases | \
        /bin/gzip > /backup/mysql/all-databases.sql.gz
```

How it works...

Wherever a variable is referenced in the template, for example `<%= @mysql_password %>`, Puppet will replace it with the corresponding value, `secret`.

There's more...

In the example, we only used one variable in the template, but you can have as many as you like. These can also be facts:

```
ServerName <%= @fqdn %>
```

Or Ruby expressions:

```
MAILTO=<%= @emails.join(',') %>
```

Or any Ruby code you want:

```
ServerAdmin <%= @sitedomain == 'coldcomfort.com' ?
 'seth@coldcomfort.com' : 'flora@poste.com' %>
```

See also

- The *Using GnuPG to encrypt secrets* recipe in this chapter
- `https://docs.puppetlabs.com/guides/templating.html`

Using array iteration in templates

In the previous example, we saw that you can use Ruby to interpolate different values in templates depending on the result of an expression. But you're not limited to getting one value at a time. You can put lots of them in a Puppet array and then have the template generate some content for each element of the array using a loop.

How to do it...

Follow these steps to build an example of iterating over arrays:

1. Modify your `site.pp` file as follows:

    ```
    node 'cookbook' {
      $ipaddresses = ['192.168.0.1',
        '158.43.128.1', '10.0.75.207' ]
      file { '/tmp/addresslist.txt':
        content => template('base/addresslist.erb')
      }
    }
    ```

2. Create the file `modules/base/templates/addresslist.erb` with the following contents:

    ```
    <% @ipaddresses.each do |ip| -%>
    IP address <%= ip %> is present
    <% end -%>
    ```

3. Run Puppet:

    ```
    [root@cookbook ~]# puppet agent -t
    Info: Caching catalog for cookbook.example.com
    Info: Applying configuration version '1412141917'
    Notice: /Stage[main]/Main/Node[cookbook]/File[/tmp/addresslist.
    txt]/ensure: defined content as '{md5}073851229d7b2843830024afb2b3
    902d'
    Notice: Finished catalog run in 0.30 seconds
    ```

4. Check the contents of the generated file:

    ```
    [root@cookbook ~]# cat /tmp/addresslist.txt
      IP address 192.168.0.1 is present.
      IP address 158.43.128.1 is present.
      IP address 10.0.75.207 is present.
    ```

How it works...

In the first line of the template, we reference the array `ipaddresses`, and call its `each` method:

```
<% @ipaddresses.each do |ip| -%>
```

In Ruby, this creates a loop that will execute once for each element of the array. Each time round the loop, the variable `ip` will be set to the value of the current element.

In our example, the `ipaddresses` array contains three elements, so the following line will be executed three times, once for each element:

```
IP address <%= ip %> is present.
```

This will result in three output lines:

```
IP address 192.168.0.1 is present.
IP address 158.43.128.1 is present.
IP address 10.0.75.207 is present.
```

The final line ends the loop:

```
<% end -%>
```

Note that the first and last lines end with `-%>` instead of just `%>` as we saw before. The effect of the `-` is to suppress the new line that would otherwise be generated on each pass through the loop, giving us unwanted blank lines in the file.

There's more...

Templates can also iterate over hashes, or arrays of hashes:

```
$interfaces = [ {name => 'eth0', ip => '192.168.0.1'},
  {name => 'eth1', ip => '158.43.128.1'},
  {name => 'eth2', ip => '10.0.75.207'} ]

<% @interfaces.each do |interface| -%>
Interface <%= interface['name'] %> has the address <%= interface['ip']
%>.
<% end -%>

Interface eth0 has the address 192.168.0.1.
Interface eth1 has the address 158.43.128.1.
Interface eth2 has the address 10.0.75.207.
```

See also

- The *Using ERB templates* recipe in this chapter

Using EPP templates

EPP templates are a new feature in Puppet 3.5 and newer versions. EPP templates use a syntax similar to ERB templates but are not compiled through Ruby. Two new functions are defined to call EPP templates, `epp`, and `inline_epp`. These functions are the EPP equivalents of the ERB functions `template` and `inline_template`, respectively. The main difference with EPP templates is that variables are referenced using the Puppet notation, `$variable` instead of `@variable`.

How to do it...

1. Create an EPP template in `~/puppet/epp-test.epp` with the following content:

```
This is <%= $message %>.
```

2. Create an `epp.pp` manifest, which uses the `epp` and `inline_epp` functions:

```
$message = "the message"
file {'/tmp/epp-test':
  content => epp('/home/thomas/puppet/epp-test.epp')
}
notify {inline_epp('Also prints <%= $message %>'):}
```

3. Apply the manifest making sure to use the future parser (the future parser is required for the `epp` and `inline_epp` functions to be defined):

```
t@mylaptop ~/puppet $ puppet apply epp.pp --parser=future
Notice: Compiled catalog for mylaptop in environment production in 1.03 seconds
Notice: /Stage[main]/Main/File[/tmp/epp-test]/ensure: defined content as '{md5}999ccc2507d79d50fae0775d69b63b8c'
Notice: Also prints the message
```

4. Verify that the template worked as intended:

```
t@mylaptop ~/puppet $ cat /tmp/epp-test
This is the message.
```

How it works...

Using the future parser, the `epp` and `inline_epp` functions are defined. The main difference between EPP templates and ERB templates is that variables are referenced in the same way they are within Puppet manifests.

There's more...

Both `epp` and `inline_epp` allow for variables to be overridden within the function call. A second parameter to the function call can be used to specify values for variables used within the scope of the function call. For example, we can override the value of `$message` with the following code:

```
file {'/tmp/epp-test':
  content => epp('/home/tuphill/puppet/epp-test.epp',
    { 'message' => "override $message"} )
}
notify {inline_epp('Also prints <%= $message %>',
  { 'message' => "inline override $message"}):}
```

Now when we run Puppet and verify the output we see that the value of `$message` has been overridden:

```
t@mylaptop ~/puppet $ puppet apply epp.pp --parser=future
Notice: Compiled catalog for mylaptop.pan.costco.com in environment
production in 0.85 seconds
Notice: Also prints inline override the message
Notice: Finished catalog run in 0.05 seconds
t@mylaptop ~/puppet $ cat /tmp/epp-test
This is override the message.
```

Using GnuPG to encrypt secrets

We often need Puppet to have access to secret information, such as passwords or crypto keys, for it to configure systems properly. But how do you avoid putting such secrets directly into your Puppet code, where they're visible to anyone who has read access to your repository?

It's a common requirement for third-party developers and contractors to be able to make changes via Puppet, but they definitely shouldn't see any confidential information. Similarly, if you're using a distributed Puppet setup like that described in *Chapter 2*, *Puppet Infrastructure*, every machine has a copy of the whole repo, including secrets for other machines that it doesn't need and shouldn't have. How can we prevent this?

One answer is to encrypt the secrets using the **GnuPG** tool, so that any secret information in the Puppet repo is undecipherable (for all practical purposes) without the appropriate key. Then we distribute the key securely to the people or machines that need it.

Getting ready

First you'll need an encryption key, so follow these steps to generate one. If you already have a GnuPG key that you'd like to use, go on to the next section. To complete this section, you will need to install the gpg command:

1. Use `puppet` resource to install gpg:

   ```
   # puppet resource package gnupg ensure=installed
   ```

   > You may need to use gnupg2 as the package name, depending on your target OS.

2. Run the following command. Answer the prompts as shown, except to substitute your name and e-mail address for mine. When prompted for a passphrase, just hit *Enter*:

   ```
   t@mylaptop ~/puppet $ gpg --gen-key
   gpg (GnuPG) 1.4.18; Copyright (C) 2014 Free Software Foundation,
   Inc.
   This is free software: you are free to change and redistribute it.
   There is NO WARRANTY, to the extent permitted by law.
   Please select what kind of key you want:
      (1) RSA and RSA (default)
      (2) DSA and Elgamal
      (3) DSA (sign only)
      (4) RSA (sign only)
   Your selection? 1
   RSA keys may be between 1024 and 4096 bits long.
   What keysize do you want? (2048) 2048
   Requested keysize is 2048 bits
   Please specify how long the key should be valid.
            0 = key does not expire
         <n>  = key expires in n days
         <n>w = key expires in n weeks
         <n>m = key expires in n months
         <n>y = key expires in n years
   Key is valid for? (0) 0
   Key does not expire at all
   Is this correct? (y/N) y
   ```

```
You need a user ID to identify your key; the software constructs
the user ID
from the Real Name, Comment and Email Address in this form:
    "Heinrich Heine (Der Dichter) <heinrichh@duesseldorf.de>"

Real name: Thomas Uphill
Email address: thomas@narrabilis.com
Comment: <enter>
You selected this USER-ID:
    "Thomas Uphill <thomas@narrabilis.com>"

Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? o
You need a Passphrase to protect your secret key.
```

Hit enter twice here to have an empty passphrase

```
You don't want a passphrase - this is probably a *bad* idea!
I will do it anyway.  You can change your passphrase at any time,
using this program with the option "--edit-key".

gpg: key F1C1EE49 marked as ultimately trusted
public and secret key created and signed.

gpg: checking the trustdb
gpg: 3 marginal(s) needed, 1 complete(s) needed, PGP trust model
gpg: depth: 0  valid:   1  signed:   0  trust: 0-, 0q, 0n, 0m, 0f,
1u
pub   2048R/F1C1EE49 2014-10-01
      Key fingerprint = 461A CB4C 397F 06A7 FB82  3BAD 63CF 50D8
F1C1 EE49
uid                  Thomas Uphill <thomas@narrabilis.com>
sub   2048R/E2440023 2014-10-01
```

3. You may see a message like this if your system is not configured with a source of randomness:

```
We need to generate a lot of random bytes. It is a good idea to
perform
some other action (type on the keyboard, move the mouse, utilize
the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
```

4. In this case, install and start a random number generator daemon such as `haveged` or `rng-tools`. Copy the gpg key you just created into the `puppet` user's account on your Puppet master:

```
t@mylaptop ~ $ scp -r .gnupg puppet@puppet.example.com:
gpg.conf                                      100% 7680
7.5KB/s   00:00
random_seed                                   100%  600
0.6KB/s   00:00
pubring.gpg                                   100% 1196
1.2KB/s   00:00
secring.gpg                                   100% 2498
2.4KB/s   00:00
trustdb.gpg                                   100% 1280
1.3KB/s   00:00
```

How to do it...

With your encryption key installed on the `puppet` user's keyring (the key generation process described in the previous section will do this for you), you're ready to set up Puppet to decrypt secrets.

1. Create the following directory:

```
t@cookbook:~/puppet$ mkdir -p modules/admin/lib/puppet/parser/
functions
```

2. Create the file `modules/admin/lib/puppet/parser/functions/secret.rb` with the following contents:

```
module Puppet::Parser::Functions
  newfunction(:secret, :type => :rvalue) do |args|
    'gpg --no-tty -d #{args[0]}'
  end
end
```

3. Create the file `secret_message` with the following contents:

```
For a moment, nothing happened.
Then, after a second or so, nothing continued to happen.
```

4. Encrypt this file with the following command (use the e-mail address you supplied when creating the GnuPG key):

```
t@mylaptop ~/puppet $ gpg -e -r thomas@narrabilis.com secret_
message
```

5. Move the resulting encrypted file into your Puppet repo:

```
t@mylaptop:~/puppet$ mv secret_message.gpg modules/admin/files/
```

6. Remove the original (plaintext) file:

```
t@mylaptop:~/puppet$ rm secret_message
```

7. Modify your `site.pp` file as follows:

```
node 'cookbook' {
  $message = secret('
    /etc/puppet/environments/production/
    modules/admin/files/secret_message.gpg')
  notify { "The secret message is: ${message}": }
}
```

8. Run Puppet:

```
[root@cookbook ~]# puppet agent -t
Info: Caching catalog for cookbook.example.com
Info: Applying configuration version '1412145910'
Notice: The secret message is: For a moment, nothing happened.
Then, after a second or so, nothing continued to happen.
Notice: Finished catalog run in 0.27 seconds
```

How it works...

First, we've created a custom function to allow Puppet to decrypt the secret files using GnuPG:

```
module Puppet::Parser::Functions
  newfunction(:secret, :type => :rvalue) do |args|
    'gpg --no-tty -d #{args[0]}'
  end
end
```

The preceding code creates a function named `secret` that takes a file path as an argument and returns the decrypted text. It doesn't manage encryption keys so you need to ensure that the `puppet` user has the necessary key installed. You can check this with the following command:

```
puppet@puppet:~ $ gpg --list-secret-keys
/var/lib/puppet/.gnupg/secring.gpg
----------------------------------
sec   2048R/F1C1EE49 2014-10-01
uid                  Thomas Uphill <thomas@narrabilis.com>
ssb   2048R/E2440023 2014-10-01
```

Having set up the `secret` function and the required key, we now encrypt a message to this key:

```
tuphill@mylaptop ~/puppet $ gpg -e -r thomas@narrabilis.com secret_
message
```

This creates an encrypted file that can only be read by someone with access to the secret key (or Puppet running on a machine that has the secret key).

We then call the `secret` function to decrypt this file and get the contents:

```
    $message = secret(' /etc/puppet/environments/production/modules/admin/
    files/secret_message.gpg')
```

There's more...

You should use the `secret` function, or something like it, to protect any confidential data in your Puppet repo: passwords, AWS credentials, license keys, even other secret keys such as SSL host keys.

You may decide to use a single key, which you push to machines as they're built, perhaps as part of a bootstrap process like that described in the *Bootstrapping Puppet with Bash* recipe in *Chapter 2*, *Puppet Infrastructure*. For even greater security, you might like to create a new key for each machine, or group of machines, and encrypt a given secret only for the machines that need it.

For example, your web servers might need a certain secret that you don't want to be accessible on any other machine. You could create a key for web servers, and encrypt the data only for this key.

If you want to use encrypted data with Hiera, there is a GnuPG backend for Hiera available at `http://www.craigdunn.org/2011/10/secret-variables-in-puppet-with-hiera-and-gpg/`.

See also

- The *Configuring Hiera* recipe in *Chapter 2*, *Puppet Infrastructure*
- The *Storing secret data with hiera-gpg* recipe in *Chapter 2*, *Puppet Infrastructure*

Installing packages from a third-party repository

Most often you will want to install packages from the main distribution repo, so a simple package resource will do:

```
package { 'exim4': ensure => installed }
```

Sometimes, you need a package that is only found in a third-party repository (an Ubuntu PPA, for example), or it might be that you need a more recent version of a package than that provided by the distribution, which is available from a third party.

On a manually-administered machine, you would normally do this by adding the repo source configuration to `/etc/apt/sources.list.d` (and, if necessary, a gpg key for the repo) before installing the package. We can automate this process easily with Puppet.

How to do it…

In this example, we'll use the popular Percona APT repo (Percona is a MySQL consulting firm who maintain and release their own specialized version of MySQL, more information is available at `http://www.percona.com/software/repositories`):

1. Create the file `modules/admin/manifests/percona_repo.pp` with the following contents:

```
# Install Percona APT repo
class admin::percona_repo {
  exec { 'add-percona-apt-key':
    unless  => '/usr/bin/apt-key list |grep percona',
    command => '/usr/bin/gpg --keyserver
      hkp://keys.gnupg.net --recv-keys 1C4CBDCDCD2EFD2A
      && /usr/bin/gpg -a --export CD2EFD2A |
      apt-key add -',
    notify  => Exec['percona-apt-update'],
  }

  exec { 'percona-apt-update':
    command     => '/usr/bin/apt-get update',
    require     => [File['/etc/apt/sources.list.d/percona.list'],
File['/etc/apt/preferences.d/00percona.pref']],
    refreshonly => true,
  }

  file { '/etc/apt/sources.list.d/percona.list':
    content => 'deb http://repo.percona.com/apt wheezy
      main',
    notify  => Exec['percona-apt-update'],
  }

  file { '/etc/apt/preferences.d/00percona.pref':
    content => "Package: *\nPin: release o=Percona
    Development Team\nPin-Priority: 1001",
    notify  => Exec['percona-apt-update'],
  }
}
```

2. Modify your `site.pp` file as follows:

```
node 'cookbook' {
  include admin::percona_repo

  package { 'percona-server-server-5.5':
    ensure  => installed,
    require => Class['admin::percona_repo'],
  }
}
```

3. Run Puppet:

```
root@cookbook-deb:~# puppet agent -t
Info: Caching catalog for cookbook-deb
Notice: /Stage[main]/Admin::Percona_repo/Exec[add-percona-apt-
key]/returns: executed successfully
Info: /Stage[main]/Admin::Percona_repo/Exec[add-percona-apt-key]:
Scheduling refresh of Exec[percona-apt-update]
Notice: /Stage[main]/Admin::Percona_repo/File[/etc/apt/
sources.list.d/percona.list]/ensure: defined content as '{md5}
b8d479374497255804ffbf0a7bcdf6c2'
Info: /Stage[main]/Admin::Percona_repo/File[/etc/apt/sources.
list.d/percona.list]: Scheduling refresh of Exec[percona-apt-
update]
Notice: /Stage[main]/Admin::Percona_repo/File[/etc/apt/
preferences.d/00percona.pref]/ensure: defined content as '{md5}1d8
ca6c1e752308a9bd3018713e2d1ad'
Info: /Stage[main]/Admin::Percona_repo/File[/etc/apt/
preferences.d/00percona.pref]: Scheduling refresh of Exec[percona-
apt-update]
Notice: /Stage[main]/Admin::Percona_repo/Exec[percona-apt-update]:
Triggered 'refresh' from 3 events
```

How it works...

In order to install any Percona package, we first need to have the repository configuration installed on the machine. This is why the `percona-server-server-5.5` package (Percona's version of the standard MySQL server) requires the `admin::percona_repo` class:

```
package { 'percona-server-server-5.5':
  ensure  => installed,
  require => Class['admin::percona_repo'],
}
```

So, what does the `admin::percona_repo` class do? It:

- Installs the Percona APT key with which the packages are signed
- Configures the Percona repo URL as a file in `/etc/apt/sources.list.d`
- Runs `apt-get update` to retrieve the repo metadata
- Adds an APT pin configuration in `/etc/apt/preferences.d`

First of all, we install the APT key:

```
exec { 'add-percona-apt-key':
  unless  => '/usr/bin/apt-key list |grep percona',
  command => '/usr/bin/gpg --keyserver  hkp://keys.gnupg.net --
    recv-keys 1C4CBDCDCD2EFD2A && /usr/bin/gpg -a --export
    CD2EFD2A | apt-key add -',
  notify  => Exec['percona-apt-update'],
}
```

The `unless` parameter checks the output of `apt-key list` to make sure that the Percona key is not already installed, in which case we need not do anything. Assuming it isn't, the `command` runs:

```
/usr/bin/gpg --keyserver  hkp://keys.gnupg.net --recv-keys
1C4CBDCDCD2EFD2A && /usr/bin/gpg -a --export CD2EFD2A | apt-key add -
```

This command retrieves the key from the GnuPG keyserver, exports it in the ASCII format, and pipes this into the `apt-key add` command, which adds it to the system keyring. You can use a similar pattern for most third-party repos that require an APT signing key.

Having installed the key, we add the repo configuration:

```
file { '/etc/apt/sources.list.d/percona.list':
  content => 'deb http://repo.percona.com/apt wheezy main',
  notify  => Exec['percona-apt-update'],
}
```

Then run `apt-get update` to update the system's APT cache with the metadata from the new repo:

```
exec { 'percona-apt-update':
  command     => '/usr/bin/apt-get update',
  require     => [File['/etc/apt/sources.list.d/percona.list'],
File['/etc/apt/preferences.d/00percona.pref']],
  refreshonly => true,
}
```

Finally, we configure the APT pin priority for the repo:

```
file { '/etc/apt/preferences.d/00percona.pref':
  content => "Package: *\nPin: release o=Percona Development Team\
nPin-Priority: 1001",
  notify  => Exec['percona-apt-update'],
}
```

This ensures that packages installed from the Percona repo will never be superseded by packages from somewhere else (the main Ubuntu distro, for example). Otherwise, you could end up with broken dependencies and be unable to install the Percona packages automatically.

There's more...

The APT package framework is specific to the Debian and Ubuntu systems. There is a forge module for managing apt repos, `https://forge.puppetlabs.com/puppetlabs/apt`. If you're on a Red Hat or CentOS-based system, you can use the `yumrepo` resources to manage RPM repositories directly:

`http://docs.puppetlabs.com/references/latest/type.html#yumrepo`

Comparing package versions

Package version numbers are odd things. They look like decimal numbers, but they're not: a version number is often in the form of `2.6.4`, for example. If you need to compare one version number with another, you can't do a straightforward string comparison: `2.6.4` would be interpreted as greater than `2.6.12`. And a numeric comparison won't work because they're not valid numbers.

Puppet's `versioncmp` function comes to the rescue. If you pass two things that look like version numbers, it will compare them and return a value indicating which is greater:

```
versioncmp( A, B )
```

returns:

- 0 if A and B are equal
- Greater than 1 if A is higher than B
- Less than 0 if A is less than B

How to do it...

Here's an example using the `versioncmp` function:

1. Modify your `site.pp` file as follows:

```
node 'cookbook' {
  $app_version = '1.2.2'
  $min_version = '1.2.10'

  if versioncmp($app_version, $min_version) >= 0 {
    notify { 'Version OK': }
  } else {
    notify { 'Upgrade needed': }
  }
}
```

2. Run Puppet:

```
[root@cookbook ~]# puppet agent -t
Info: Caching catalog for cookbook.example.com
Notice: Upgrade needed
```

3. Now change the value of `$app_version`:

```
$app_version = '1.2.14'
```

4. Run Puppet again:

```
[root@cookbook ~]# puppet agent -t
Info: Caching catalog for cookbook.example.com
Notice: Version OK
```

How it works...

We've specified that the minimum acceptable version (`$min_version`) is `1.2.10`. So, in the first example, we want to compare it with `$app_version` of `1.2.2`. A simple alphabetic comparison of these two strings (in Ruby, for example) would give the wrong result, but `versioncmp` correctly determines that `1.2.2` is less than `1.2.10` and alerts us that we need to upgrade.

In the second example, `$app_version` is now `1.2.14`, which `versioncmp` correctly recognizes as greater than `$min_version` and so we get the message **Version OK**.

5
Users and Virtual Resources

"Nothing is a problem, until it's a problem."

In this chapter, we will cover the following recipes:

- Using virtual resources
- Managing users with virtual resources
- Managing users' SSH access
- Managing users' customization files
- Using exported resources

Introduction

Users can be a real pain. I don't mean the people, though doubtless that's sometimes true. But keeping UNIX user accounts and file permissions in sync across a network of machines, some of them running different operating systems, can be very challenging without some kind of centralized configuration management.

Each new developer who joins the organization needs an account on every machine, along with `sudo` privileges and group memberships, and needs their SSH key authorized for a bunch of different accounts. The system administrator who has to take care of this manually will be at the job all day, while the system administrator who uses Puppet will be done in minutes, and head out for an early lunch.

In this chapter, we'll look at some handy patterns and techniques to manage users and their associated resources. Users are also one of the most common applications for virtual resources, so we'll find out all about those. In the final section, we'll introduce exported resources, which are related to virtual resources.

Using virtual resources

Virtual resources in Puppet might seem complicated and confusing but, in fact, they're very simple. They're exactly like regular resources, but they don't actually take effect until they're realized (in the sense of "made real"); whereas a regular resource can only be declared once per node (so two classes can't declare the same resource, for example). A virtual resource can be realized as many times as you like.

This comes in handy when you need to move applications and services between machines. If two applications that use the same resource end up sharing a machine, they would cause a conflict unless you make the resource virtual.

To clarify this, let's look at a typical situation where virtual resources might come in handy.

You are responsible for two popular web applications: WordPress and Drupal. Both are web apps running on Apache, so they both require the Apache package to be installed. The definition for WordPress might look something like the following:

```
class wordpress {
  package {'httpd':
    ensure => 'installed',
  }
  service {'httpd':
    ensure => 'running',
    enable => true,
  }
}
```

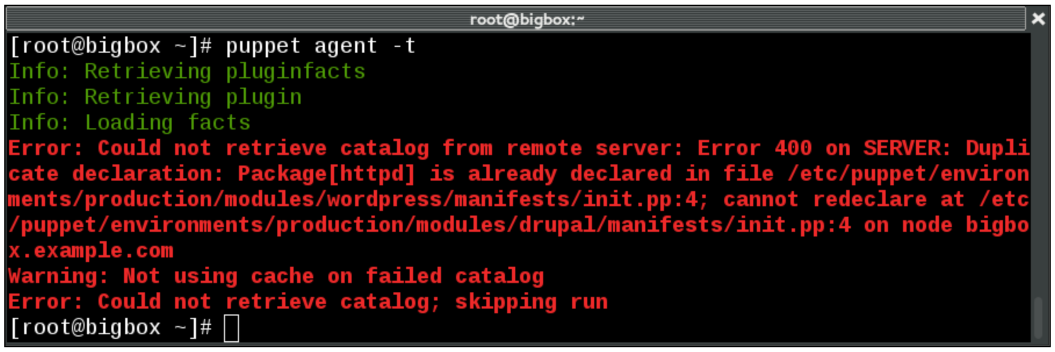
The definition for Drupal might look like this:

```
class drupal {
  package {'httpd':
    ensure => 'installed',
  }
  service {'httpd':
    ensure => 'running',
    enable => true,
  }
}
```

All is well until you need to consolidate both apps onto a single server:

```
node 'bigbox' {
  include wordpress
  include drupal
}
```

Now Puppet will complain because you tried to define two resources with the same name: `httpd`.



```
[root@bigbox ~]# puppet agent -t
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Loading facts
Error: Could not retrieve catalog from remote server: Error 400 on SERVER: Dupli
cate declaration: Package[httpd] is already declared in file /etc/puppet/environ
ments/production/modules/wordpress/manifests/init.pp:4; cannot redeclare at /etc
/puppet/environments/production/modules/drupal/manifests/init.pp:4 on node bigbo
x.example.com
Warning: Not using cache on failed catalog
Error: Could not retrieve catalog; skipping run
[root@bigbox ~]#
```

You could remove the duplicate Apache package definition from one of the classes, but then nodes without the class including Apache would fail. You can get around this problem by putting the Apache package in its own class and then using `include apache` everywhere it's needed; Puppet doesn't mind you including the same class multiple times. In reality, putting Apache in its own class solves most problems but, in general, this method has the disadvantage that every potentially conflicting resource must have its own class.

Virtual resources can be used to solve this problem. A virtual resource is just like a normal resource, except that it starts with an `@` character:

```
@package { 'httpd': ensure => installed }
```

You can think of it as being like a placeholder resource; you want to define it but you aren't sure you are going to use it yet. Puppet will read and remember virtual resource definitions, but won't actually create the resource until you realize the resource.

To create the resource, use the `realize` function:

```
realize(Package['httpd'])
```

You can call `realize` as many times as you want on the resource and it won't result in a conflict. So virtual resources are the way to go when several different classes all require the same resource, and they may need to coexist on the same node.

How to do it...

Here's how to build the example using virtual resources:

1. Create the virtual module with the following contents:

```
class virtual {
  @package {'httpd': ensure => installed }
  @service {'httpd':
    ensure  => running,
    enable  => true,
    require => Package['httpd']
  }
}
```

2. Create the Drupal module with the following contents:

```
class drupal {
  include virtual
  realize(Package['httpd'])
  realize(Service['httpd'])
}
```

3. Create the WordPress module with the following contents:

```
class wordpress {
  include virtual
  realize(Package['httpd'])
  realize(Service['httpd'])
}
```

4. Modify your `site.pp` file as follows:

```
node 'bigbox' {
  include drupal
  include wordpress
}
```

5. Run Puppet:

```
bigbox# puppet agent -t
Info: Caching catalog for bigbox.example.com
Info: Applying configuration version '1413179615'
Notice: /Stage[main]/Virtual/Package[httpd]/ensure: created
Notice: /Stage[main]/Virtual/Service[httpd]/ensure: ensure changed
'stopped' to 'running'
```

```
Info: /Stage[main]/Virtual/Service[httpd]: Unscheduling refresh on
Service[httpd]
Notice: Finished catalog run in 6.67 seconds
```

How it works...

You define the package and service as virtual resources in one place: the `virtual` class. All nodes can include this class and you can put all your virtual services and packages in it. None of the packages will actually be installed on a node or services started until you call `realize`:

```
class virtual {
  @package { 'httpd': ensure => installed }
}
```

Every class that needs the Apache package can call `realize` on this virtual resource:

```
class drupal {
  include virtual
  realize(Package['httpd'])
}
```

Puppet knows, because you made the resource virtual, that you intended to have multiple references to the same package, and didn't just accidentally create two resources with the same name. So it does the right thing.

There's more...

To realize virtual resources, you can also use the collection *spaceship* syntax:

```
Package <| title = 'httpd' |>
```

The advantage of this syntax is that you're not restricted to the resource name; you could also use a tag, for example:

```
Package <| tag = 'web' |>
```

Alternatively, you can just specify all instances of the resource type, by leaving the query section blank:

```
Package <| |>
```

Managing users with virtual resources

Users are a great example of a resource that may need to be realized by multiple classes. Consider the following situation. To simplify administration of a large number of machines, you defined classes for two kinds of users: `developers` and `sysadmins`. All machines need to include `sysadmins`, but only some machines need `developers`:

```
node 'server' {
  include user::sysadmins
}

node 'webserver' {
  include user::sysadmins
  include user::developers
}
```

However, some users may be members of both groups. If each group simply declares its members as regular `user` resources, this will lead to a conflict when a node includes both `developers` and `sysadmins`, as in the `webserver` example.

To avoid this conflict, a common pattern is to make all users virtual resources, defined in a single class `user::virtual` that every machine includes, and then realizing the users where they are needed. This way, there will be no conflict if a user is a member of multiple groups.

How to do it...

Follow these steps to create a `user::virtual` class:

1. Create the file `modules/user/manifests/virtual.pp` with the following contents:

   ```
   class user::virtual {
     @user { 'thomas':  ensure => present }
     @user { 'theresa': ensure => present }
     @user { 'josko':   ensure => present }
     @user { 'nate':    ensure => present }
   }
   ```

2. Create the file `modules/user/manifests/developers.pp` with the following contents:

   ```
   class user::developers {
     realize(User['theresa'])
     realize(User['nate'])
   }
   ```

3. Create the file `modules/user/manifests/sysadmins.pp` with the following contents:

   ```
   class user::sysadmins {
     realize(User['thomas'])
     realize(User['theresa'])
     realize(User['josko'])
   }
   ```

4. Modify your `nodes.pp` file as follows:

   ```
   node 'cookbook' {
     include user::virtual
     include user::sysadmins
     include user::developers
   }
   ```

5. Run Puppet:

   ```
   cookbook# puppet agent -t
   Info: Caching catalog for cookbook.example.com
   Info: Applying configuration version '1413180590'
   Notice: /Stage[main]/User::Virtual/User[theresa]/ensure: created
   Notice: /Stage[main]/User::Virtual/User[nate]/ensure: created
   Notice: /Stage[main]/User::Virtual/User[thomas]/ensure: created
   Notice: /Stage[main]/User::Virtual/User[josko]/ensure: created
   Notice: Finished catalog run in 0.47 seconds
   ```

How it works...

When we include the `user::virtual` class, all the users are declared as virtual resources (because we included the `@` symbol):

```
@user { 'thomas':  ensure => present }
@user { 'theresa': ensure => present }
@user { 'josko':   ensure => present }
@user { 'nate':    ensure => present }
```

That is to say, the resources exist in Puppet's catalog; they can be referred to by and linked with other resources, and they are in every respect identical to regular resources, except that Puppet doesn't actually create the corresponding users on the machine.

In order for that to happen, we need to call `realize` on the virtual resources. When we include the `user::sysadmins` class, we get the following code:

```
realize(User['thomas'])
realize(User['theresa'])
realize(User['josko'])
```

Calling `realize` on a virtual resource tells Puppet, "I'd like to use that resource now". This is what it does, as we can see from the run output:

```
Notice: /Stage[main]/User::Virtual/User[theresa]/ensure: created
```

However, Theresa is in both the `developers` and `sysadmins` classes! Won't that mean we end up calling `realize` twice on the same resource?

```
realize(User['theresa'])
...
realize(User['theresa'])
```

Yes, it does, and that's fine. You're explicitly allowed to realize resources multiple times, and there will be no conflict. So long as some class, somewhere, calls `realize` on Theresa's account, it will be created. Unrealized resources are simply discarded during catalog compilation.

There's more...

When you use this pattern to manage your own users, every node should include the `user::virtual` class, as a part of your basic housekeeping configuration. This class will declare all users (as virtual) in your organization or site. This should also include any users who exist only to run applications or services (such as `Apache`, `www-data`, or `deploy`, for example). Then, you can realize them as needed on individual nodes or in specific classes.

For production use, you'll probably also want to specify a UID and GID for each user or group, so that these numeric identifiers are synchronized across your network. You can do this using the `uid` and `gid` parameters for the `user` resource.

If you don't specify a user's UID, for example, you'll just get whatever is the next ID number available on a given machine, so the same user on different machines will have a different UID. This can lead to permission problems when using shared storage, or moving files between machines.

A common pattern when defining users as virtual resources is to assign tags to the users based on their assigned roles within your organization. You can then use the `collector` syntax instead of `realize` to collect users with specific tags applied.

For example, see the following code snippet:

```
@user { 'thomas':  ensure => present, tag => 'sysadmin' }
@user { 'theresa': ensure => present, tag => 'sysadmin' }
@user { 'josko':   ensure => present, tag => 'dev' }
User <| tag == 'sysadmin' |>
```

In the previous example, only users `thomas` and `theresa` would be included.

See also

- The *Using virtual resources* recipe in this chapter
- The *Managing users' customization files* recipe in this chapter

Managing users' SSH access

A sensible approach to access control for servers is to use named user accounts with passphrase-protected SSH keys, rather than having users share an account with a widely known password. Puppet makes this easy to manage thanks to the built-in `ssh_authorized_key` type.

To combine this with virtual users, as described in the previous section, you can create a `define`, which includes both the `user` and `ssh_authorized_key` resources. This will also come in handy when adding customization files and other resources to each user.

How to do it...

Follow these steps to extend your virtual users' class to include SSH access:

1. Create a new module `ssh_user` to contain our `ssh_user` definition. Create the `modules/ssh_user/manifests/init.pp` file as follows:

```
define ssh_user($key,$keytype) {
  user { $name:
    ensure     => present,
  }

  file { "/home/${name}":
    ensure => directory,
    mode   => '0700',
    owner  => $name,
    require => User["$name"]
  }
  file { "/home/${name}/.ssh":
    ensure => directory,
    mode   => '0700',
    owner  => "$name",
    require => File["/home/${name}"],
  }

  ssh_authorized_key { "${name}_key":
    key     => $key,
    type    => "$keytype",
    user    => $name,
    require => File["/home/${name}/.ssh"],
  }
}
```

2. Modify your `modules/user/manifests/virtual.pp` file, comment out the previous definition for user `thomas`, and replace it with the following:

```
@ssh_user { 'thomas':
  key     => 'AAAAB3NzaC1yc2E...XaWM5sX0z',
  keytype => 'ssh-rsa'
}
```

3. Modify your `modules/user/manifests/sysadmins.pp` file as follows:

```
class user::sysadmins {
    realize(Ssh_user['thomas'])
}
```

4. Modify your `site.pp` file as follows:

```
node 'cookbook' {
  include user::virtual
  include user::sysadmins
}
```

5. Run Puppet:

```
cookbook# puppet agent -t
Info: Caching catalog for cookbook.example.com
Info: Applying configuration version '1413254461'
Notice: /Stage[main]/User::Virtual/Ssh_user[thomas]/File[/home/
thomas/.ssh]/ensure: created
Notice: /Stage[main]/User::Virtual/Ssh_user[thomas]/Ssh_
authorized_key[thomas_key]/ensure: created
Notice: Finished catalog run in 0.11 seconds
```

How it works...

For each user in our `user::virtual` class, we need to create:

- The user account itself
- The user's home directory and `.ssh` directory
- The user's `.ssh/authorized_keys` file

We could declare separate resources to implement all of these for each user, but it's much easier to create a definition instead, which wraps them into a single resource. By creating a new module for our definition, we can refer to `ssh_user` from anywhere (in any scope):

```
define ssh_user ($key, $keytype) {
  user { $name:
    ensure     => present,
  }
```

After we create the user, we can then create the home directory; we need the user first so that when we assign ownership, we can use the username, `owner => $name`:

```
  file { "/home/${name}":
    ensure => directory,
    mode => '0700',
    owner => $name,
    require => User["$name"]
  }
```

Puppet can create the users' home directory using the `managehome` attribute to the user resource. Relying on this mechanism is problematic in practice, as it does not account for users that were created outside of Puppet without home directories.

Next, we need to ensure that the `.ssh` directory exists within the home directory of the user. We require the home directory, `File["/home/${name}"]`, since that needs to exist before we create this subdirectory. This implies that the user already exists because the home directory required the user:

```
  file { "/home/${name}/.ssh":
    ensure => directory,
    mode   => '0700',
    owner  => $name ,
    require => File["/home/${name}"],
  }
```

Finally, we create the `ssh_authorized_key` resource, again requiring the containing folder (`File["/home/${name}/.ssh"]`). We use the `$key` and `$keytype` variables to assign the key and type parameters to the `ssh_authorized_key` type as follows:

```
      ssh_authorized_key { "${name}_key":
        key     => $key,
        type    => "$keytype",
        user    => $name,
        require => File["/home/${name}/.ssh"],
      }
    }
```

We passed the `$key` and `$keytype` variables when we defined the `ssh_user` resource for `thomas`:

```
    @ssh_user { 'thomas':
      key => 'AAAAB3NzaC1yc2E...XaWM5sX0z',
      keytype => 'ssh-rsa'
    }
```

The value for `key`, in the preceding code snippet, is the ssh key's public key value; it is usually stored in an `id_rsa.pub` file.

Now, with everything defined, we just need to call `realize` on `thomas` for all these resources to take effect:

```
    realize(Ssh_user['thomas'])
```

Notice that this time the virtual resource we're realizing is not simply the `user` resource, as before, but the `ssh_user` defined type we created, which includes the user and the related resources needed to set up the SSH access:

```
Notice: /Stage[main]/User::Virtual/Ssh_user[thomas]/User[thomas]/ensure:
created
Notice: /Stage[main]/User::Virtual/Ssh_user[thomas]/File[/home/thomas]/
ensure: created
Notice: /Stage[main]/User::Virtual/Ssh_user[thomas]/File[/home/thomas/.
ssh]/ensure: created
Notice: /Stage[main]/User::Virtual/Ssh_user[thomas]/Ssh_authorized_
key[thomas_key]/ensure: created
```

There's more...

Of course, you can add whatever resources you like to the `ssh_user` definition to have Puppet automatically create them for new users. We'll see an example of this in the next recipe, *Managing users' customization files*.

Managing users' customization files

Users tend to customize their shell environments, terminal colors, aliases, and so forth. This is usually achieved by a number of **dotfiles** in their home directory, for example, `.bash_profile` or `.vimrc`.

You can use Puppet to synchronize and update each user's dotfiles across a number of machines by extending the virtual user setup we developed throughout this chapter. We'll start a new module, `admin_user` and use the file types, `recurse` attribute to copy files into each user's home directory.

How to do it...

Here's what you need to do:

1. Create the `admin_user` defined type (`define admin_user`) in the `modules/admin_user/manifests/init.pp` file as follows:

```
define admin_user ($key, $keytype, $dotfiles = false) {
  $username = $name
  user { $username:
    ensure     => present,
  }
  file { "/home/${username}/.ssh":
    ensure  => directory,
    mode    => '0700',
    owner   => $username,
    group   => $username,
    require => File["/home/${username}"],
  }
  ssh_authorized_key { "${username}_key":
    key     => $key,
    type    => "$keytype",
    user    => $username,
    require => File["/home/${username}/.ssh"],
  }
  # dotfiles
  if $dotfiles == false {
    # just create the directory
    file { "/home/${username}":
      ensure  => 'directory',
      mode    => '0700',
      owner   => $username,
      group   => $username,
      require => User["$username"]
    }
  } else {
    # copy in all the files in the subdirectory
    file { "/home/${username}":
      recurse => true,
      mode    => '0700',
      owner   => $username,
      group   => $username,
      source  => "puppet:///modules/admin_user/${username}",
      require => User["$username"]
    }
  }
}
```

2. Modify the file `modules/user/manifests/sysadmins.pp` as follows:

```
class user::sysadmins {
  realize(Admin_user['thomas'])
}
```

3. Alter the definition of `thomas` in `modules/user/manifests/virtual.pp` as follows:

```
@ssh_user { 'thomas':
  key => 'AAAAB3NzaC1yc2E...XaWM5sX0z',
  keytype => 'ssh-rsa',
  dotfiles => true
}
```

4. Create a subdirectory in the `admin_user` module for the file of user `thomas`:

```
$ mkdir -p modules/admin_user/files/thomas
```

5. Create dotfiles for the user `thomas` in the directory you just created:

```
$ echo "alias vi=vim" > modules/admin_user/files/thomas/.bashrc
$ echo "set tabstop=2" > modules/admin_user/files/thomas/.vimrc
```

6. Make sure your `site.pp` file reads as follows:

```
node 'cookbook' {
  include user::virtual
```

```
      include user::sysadmins
    }
```

7. Run Puppet:

```
cookbook# puppet agent -t
Info: Caching catalog for cookbook.example.com
Info: Applying configuration version '1413266235'
Notice: /Stage[main]/User::Virtual/Admin_user[thomas]/
User[thomas]/ensure: created
Notice: /Stage[main]/User::Virtual/Admin_user[thomas]/File[/home/
thomas]/ensure: created
Notice: /Stage[main]/User::Virtual/Admin_user[thomas]/
File[/home/thomas/.vimrc]/ensure: defined content as '{md5}
cb2af2d35b18b5ac2539057bd429d3ae'
Notice: /Stage[main]/User::Virtual/Admin_user[thomas]/File[/home/
thomas/.bashrc]/ensure: defined content as '{md5}033c3484e4b276e06
41becc3aa268a3a'
Notice: /Stage[main]/User::Virtual/Admin_user[thomas]/File[/home/
thomas/.ssh]/ensure: created
Notice: /Stage[main]/User::Virtual/Admin_user[thomas]/Ssh_
authorized_key[thomas_key]/ensure: created
Notice: Finished catalog run in 0.36 seconds
```

How it works...

We created a new `admin_user` definition, which defines the home directory recursively if `$dotfiles` is not `false` (the default value):

```
if $dotfiles == 'false' {
  # just create the directory
  file { "/home/${username}":
    ensure  => 'directory',
    mode    => '0700',
    owner   => $username,
    group   => $username,
    require => User["$username"]
  }
} else {
  # copy in all the files in the subdirectory
  file { "/home/${username}":
    recurse => true,
    mode    => '0700',
    owner   => $username,
    group   => $username,
    source  => "puppet:///modules/admin_user/${username}",
    require => User["$username"]
  }
}
```

We created a directory to hold the user's dotfiles within the `admin_user` module; all the files within that directory will be copied into the user's home directory, as shown in the puppet run output in the following command line:

```
Notice: /Stage[main]/User::Virtual/Admin_user[thomas]/File[/home/thomas/.
vimrc]/ensure: defined content as '{md5}cb2af2d35b18b5ac2539057bd429d3ae'
Notice: /Stage[main]/User::Virtual/Admin_user[thomas]/File[/home/thomas/.
bashrc]/ensure: defined content as '{md5}033c3484e4b276e0641becc3aa268a
3a'
```

Using the `recurse` option allows us to add as many dotfiles as we wish for each user without having to modify the definition of the user.

There's more...

We could specify that the `source` attribute of the home directory is a directory where users can place their own dotfiles. This way, each user could modify their own dotfiles and have them transferred to all the nodes in the network without our involvement.

See also

- The *Managing users with virtual resources* recipe in this chapter

Using exported resources

All our recipes up to this point have dealt with a single machine. It is possible with Puppet to have resources from one node affect another node. This interaction is managed with **exported resources**. Exported resources are just like any resource you might define for a node but instead of applying to the node on which they were created, they are exported for use by all nodes in the environment. Exported resources can be thought of as virtual resources that go one step further and exist beyond the node on which they were defined.

There are two actions with exported resources. When an exported resource is created, it is said to be defined. When all the exported resources are harvested, they are said to be collected. Defining exported resources is similar to virtual resources; the resource in question has two `@` symbols prepended. For example, to define a file resource as external, use `@@file`. Collecting resources is done with the space ship operator, `<<| |>>`; this is thought to look like a spaceship. To collect the exported file resource (`@@file`), you would use `File <<| |>>`.

There are many examples that use exported resources; the most common one involves SSH host keys. Using exported resources, it is possible to have every machine that is running Puppet share their SSH host keys with the other connected nodes. The idea here is that each machine exports its own host key and then collects all the keys from the other machines. In our example, we will create two classes; first, a class that exports the SSH host key from every node. We will include this class in our base class. The second class will be a collector class, which collects the SSH host keys. We will apply this class to our Jumpboxes or SSH login servers.

Jumpboxes are machines that have special firewall rules to allow them to log in to different locations.

Getting ready

To use exported resources, you will need to enable storeconfigs on your Puppet masters. It is possible to use exported resources with a masterless (decentralized) deployment; however, we will assume you are using a centralized model for this example. In *Chapter 2, Puppet Infrastructure*, we configured puppetdb using the puppetdb module from the forge. It is possible to use other backends if you desire; however, all of these except puppetdb are deprecated. More information is available at the following link: `http://projects.puppetlabs.com/projects/puppet/wiki/Using_Stored_Configuration`.

Ensure your Puppet masters are configured to use puppetdb as a storeconfigs container.

How to do it...

We'll create an `ssh_host` class to export the `ssh` keys of a host and ensure that it is included in our base class.

1. Create the first class, `base::ssh_host`, which we will include in our base class:

```
class base::ssh_host {
  @@sshkey{"$::fqdn":
    ensure       => 'present',
    host_aliases => ["$::hostname","$::ipaddress"],
    key          => $::sshdsakey,
    type         => 'dsa',
  }
}
```

2. Remember to include this class from inside the base class definition:

   ```
   class base {
     ...
     include ssh_host
   }
   ```

3. Create a definition for `jumpbox`, either in a class or within the node definition for `jumpbox`:

   ```
   node 'jumpbox' {
     Sshkey <<| |>>
   }
   ```

4. Now run Puppet on a few nodes to create the exported resources. In my case, I ran Puppet on my Puppet server and my second example node (`node2`). Finally, run Puppet on `jumpbox` to verify that the SSH host keys for our other nodes are collected:

   ```
   [root@jumpbox ~]# puppet agent -t
   Info: Caching catalog for jumpbox.example.com
   Info: Applying configuration version '1413176635'
   Notice: /Stage[main]/Main/Node[jumpbox]/Sshkey[node2.example.com]/
   ensure: created
   Notice: /Stage[main]/Main/Node[jumpbox]/Sshkey[puppet]/ensure:
   created
   Notice: Finished catalog run in 0.08 seconds
   ```

How it works...

We created an `sshkey` resource for the node using the facter facts `fqdn`, `hostname`, `ipaddress`, and `sshdsakey`. We use the `fqdn` as the title for our exported resource because each exported resource must have a unique name. We can assume the `fqdn` of a node will be unique within our organization (although sometimes they may not be; Puppet can be good at finding out such things when you least expect it). We then go on to define aliases by which our node may be known. We use the hostname variable for one alias and the main IP address of the machine as the other. If you had other naming conventions for your nodes, you could include other aliases here. We assume that hosts are using DSA keys, so we use the `sshdsakey` variable in our definition. In a large installation, you would wrap this definition in tests to ensure the DSA keys existed. You would also use the RSA keys if they existed as well.

With the `sshkey` resource defined and exported, we then created a `jumpbox` node definition. In this definition, we used the spaceship syntax `Sshkey <<| |>>` to collect all defined exported `sshkey` resources.

There's more...

When defining the exported resources, you can add tag attributes to the resource to create subsets of exported resources. For example, if you had a development and production area of your network, you could create different groups of `sshkey` resources for each area as shown in the following code snippet:

```
@@sshkey{"$::fqdn":
    host_aliases => ["$::hostname","$::ipaddress"],
    key          => $::sshdsakey,
    type         => 'dsa',
    tag          => "$::environment",
  }
```

You could then modify `jumpbox` to only collect resources for production, for example, as follows:

```
Sshkey <<| tag == 'production' |>>
```

Two important things to remember when working with exported resources: first, every resource must have a unique name across your installation. Using the `fqdn` domain name within the title is usually enough to keep your definitions unique. Second, any resource can be made virtual. Even defined types that you created may be exported. Exported resources can be used to achieve some fairly complex configurations that automatically adjust when machines change.

> One word of caution when working with an extremely large number of nodes (more than 5,000) is that exported resources can take a long time to collect and apply, particularly if each exported resource creates a file.

6
Managing Resources and Files

"The art of simplicity is a puzzle of complexity".

– Douglas Horton

In this chapter, we will cover the following recipes:

- Distributing cron jobs efficiently
- Scheduling when resources are applied
- Using host resources
- Using exported host resources
- Using multiple file sources
- Distributing and merging directory trees
- Cleaning up old files
- Auditing resources
- Temporarily disabling resources

Introduction

In the previous chapter, we introduced virtual and exported resources. Virtual and exported resources are ways to manage the way in which resources are applied to a node. In this chapter, we will deal with when and how to apply resources. In some cases, you may only wish to apply a resource off hours, while in others, you may wish to only audit the resource but change nothing. In other cases, you may wish to apply completely different resources based on which node is using the code. As we will see, Puppet has the flexibility to deal with all these scenarios.

Distributing cron jobs efficiently

When you have many servers executing the same cron job, it's usually a good idea not to run them all at the same time. If all the jobs access a common server (for example, when running backups), it may put too much load on that server, and even if they don't, all the servers will be busy at the same time, which may affect their capacity to provide other services.

As usual, Puppet can help; this time, using the `inline_template` function to calculate a unique time for each job.

How to do it...

Here's how to have Puppet schedule the same job at a different time for each machine:

1. Modify your `site.pp` file as follows:

```
node 'cookbook' {
  cron { 'run-backup':
    ensure  => present,
    command => '/usr/local/bin/backup',
    hour    => inline_template('<%= @hostname.sum % 24 %>'),
    minute  => '00',
  }
}
```

2. Run Puppet:

```
[root@cookbook ~]# puppet agent -t
Info: Caching catalog for cookbook.example.com
Info: Applying configuration version '1413730771'
Notice: /Stage[main]/Main/Node[cookbook]/Cron[run-backup]/ensure: created
Notice: Finished catalog run in 0.11 seconds
```

3. Run `crontab` to see how the job has been configured:

```
[root@cookbook ~]# crontab -l
# HEADER: This file was autogenerated at Sun Oct 19 10:59:32 -0400
2014 by puppet.
# HEADER: While it can still be managed manually, it is definitely
not recommended.
# HEADER: Note particularly that the comments starting with
'Puppet Name' should
# HEADER: not be deleted, as doing so could cause duplicate cron
jobs.
# Puppet Name: run-backup
0 15 * * * /usr/local/bin/backup
```

How it works...

We want to distribute the hour of the cron job runs across all our nodes. We choose something that is unique across all the machines and convert it to a number. This way, the value will be distributed across the nodes and will not change per node.

We can do the conversion using Ruby's `sum` method, which computes a numerical value from a string that is unique to the machine (in this case, the machine's hostname). The `sum` function will generate a large integer (in the case of the string `cookbook`, the sum is 855), and we want values for `hour` between 0 and 23, so we use Ruby's `%` (modulo) operator to restrict the result to this range. We should get a reasonably good (though not statistically uniform) distribution of values, depending on your hostnames. Another option here is to use the `fqdn_rand()` function, which works in much the same way as our example.

If all your machines have the same name (it does happen), don't expect this trick to work! In this case, you can use some other string that is unique to the machine, such as `ipaddress` or `fqdn`.

There's more...

If you have several cron jobs per machine and you want to run them a certain number of hours apart, add this number to the `hostname.sum` resource before taking the modulus. Let's say we want to run the `dump_database` job at some arbitrary time and the `run_backup` job an hour later, this can be done using the following code snippet:

```
cron { 'dump-database':
  ensure  => present,
  command => '/usr/local/bin/dump_database',
  hour    => inline_template('<%= @hostname.sum % 24 %>'),
  minute  => '00',
}

cron { 'run-backup':
  ensure  => present,
  command => '/usr/local/bin/backup',
  hour    => inline_template('<%= ( @hostname.sum + 1) % 24 %>'),
  minute  => '00',
}
```

The two jobs will end up with different `hour` values for each machine Puppet runs on, but `run_backup` will always be one hour after `dump_database`.

Most cron implementations have directories for hourly, daily, weekly, and monthly tasks. The directories `/etc/cron.hourly`, `/etc/cron.daily`, `/etc/cron.weekly`, and `/etc/cron.monthly` exist on both our Debian and Enterprise Linux machines. These directories hold executables, which will be run on the referenced schedule (hourly, daily, weekly, or monthly). I find it better to describe all the jobs in these folders and push the jobs as `file` resources. An admin on the box searching for your script will be able to find it with `grep` in these directories. To use the same trick here, we would push a cron task into `/etc/cron.hourly` and then verify that the hour is the correct hour for the task to run. To create the cron jobs using the cron directories, follow these steps:

1. First, create a `cron` class in `modules/cron/init.pp`:

   ```
   class cron {
     file { '/etc/cron.hourly/run-backup':
       content => template('cron/run-backup'),
       mode    => 0755,
     }
   }
   ```

2. Include the `cron` class in your cookbook node in `site.pp`:

   ```
   node cookbook {
     include cron
   }
   ```

3. Create a template to hold the cron task:

   ```
   #!/bin/bash

   runhour=<%= @hostname.sum%24 %>
   hour=$(date +%H)
   if [ "$runhour" -ne "$hour" ]; then
     exit 0
   fi

   echo run-backup
   ```

4. Then, run Puppet:

   ```
   [root@cookbook ~]# puppet agent -t
   Info: Caching catalog for cookbook.example.com
   Info: Applying configuration version '1413732254'
   Notice: /Stage[main]/Cron/File[/etc/cron.hourly/run-backup]/
   ensure: defined content as '{md5}5e50a7b586ce774df23301ee72904dda'
   Notice: Finished catalog run in 0.11 seconds
   ```

5. Verify that the script has the same value we calculated before, `15`:

   ```
   #!/bin/bash

   runhour=15
   hour=$(date +%H)
   if [ "$runhour" -ne "$hour" ]; then
     exit 0
   fi

   echo run-backup
   ```

Now, this job will run every hour but only when the hour, returned by `$(date +%H)`, is equal to `15` will the rest of the script run. Creating your cron jobs as file resources in a large organization makes it easier for your fellow administrators to find them. When you have a very large number of machines, it can be advantageous to add another random wait at the beginning of your job. You would need to modify the line before `echo run-backup` and add the following:

```
MAXWAIT=600
sleep $((RANDOM%MAXWAIT))
```

This will sleep a maximum of `600` seconds but will sleep a different amount each time it runs (assuming your random number generator is working). This sort of random wait is useful when you have thousands of machines, all running the same task and you need to stagger the runs as much as possible.

See also

- The *Running Puppet from cron* recipe in *Chapter 2*, *Puppet Infrastructure*

Scheduling when resources are applied

So far, we looked at what Puppet can do, and the order that it does things in, but not when it does them. One way to control this is to use the `schedule` metaparameter. When you need to limit the number of times a resource is applied within a specified period, `schedule` can help. For example:

```
exec { "/usr/bin/apt-get update":
    schedule => daily,
}
```

The most important thing to understand about `schedule` is that it can only stop a resource being applied. It doesn't guarantee that the resource will be applied with a certain frequency. For example, the `exec` resource shown in the preceding code snippet has `schedule => daily`, but this just represents an upper limit on the number of times the `exec` resource can run per day. It won't be applied more than once a day. If you don't run Puppet at all, the resource won't be applied at all. Using the hourly schedule, for instance, is meaningless on a machine configured to run the agent every 4 hours (via the `runinterval` configuration setting).

That being said, `schedule` is best used to restrict resources from running when they shouldn't, or don't need to; for example, you might want to make sure that `apt-get update` isn't run more than once an hour. There are some built-in schedules available for you to use:

- `hourly`
- `daily`
- `weekly`
- `monthly`
- `never`

However, you can modify these and create your own custom schedules, using the `schedule` resource. We'll see how to do this in the following example. Let's say we want to make sure that an `exec` resource representing a maintenance job won't run during office hours, when it might interfere with production.

How to do it...

In this example, we'll create a custom `schedule` resource and assign this to the resource:

1. Modify your `site.pp` file as follows:

```
schedule { 'outside-office-hours':
  period => daily,
  range  => ['17:00-23:59','00:00-09:00'],
  repeat => 1,
}
node 'cookbook' {
  notify { 'Doing some maintenance':
    schedule => 'outside-office-hours',
  }
}
```

2. Run Puppet. What you'll see will depend on the time of the day. If it's currently outside the office hours period you defined, Puppet will apply the resource as follows:

```
[root@cookbook ~]# date
Fri Jan  2 23:59:01 PST 2015
[root@cookbook ~]# puppet agent -t
Info: Caching catalog for cookbook.example.com
Info: Applying configuration version '1413734477'
Notice: Doing some maintenance
Notice: /Stage[main]/Main/Node[cookbook]/Notify[Doing some
maintenance]/message: defined 'message' as 'Doing some
maintenance'
Notice: Finished catalog run in 0.07 seconds
```

3. If the time is within the office hours period, Puppet will do nothing:

```
[root@cookbook ~]# date
Fri Jan  2 09:59:01 PST 2015
[root@cookbook ~]# puppet agent -t
Info: Caching catalog for cookbook.example.com
Info: Applying configuration version '1413734289'
Notice: Finished catalog run in 0.09 seconds
```

How it works...

A schedule consists of three bits of information:

- The `period` (`hourly`, `daily`, `weekly`, or `monthly`)
- The `range` (defaults to the whole period, but can be a smaller part of it)
- The `repeat` count (how often the resource is allowed to be applied within the range; the default is 1 or once per period)

Our custom schedule named `outside-office-hours` supplies these three parameters:

```
schedule { 'outside-office-hours':
  period => daily,
  range  => ['17:00-23:59','00:00-09:00'],
  repeat => 1,
}
```

The `period` is `daily`, and `range` is defined as an array of two time intervals:

```
17:00-23:59
00:00-09:00
```

The schedule named `outside-office-hours` is now available for us to use with any resource, just as though it were built into Puppet such as the `daily` or `hourly` schedules. In our example, we assign this schedule to the `exec` resource using the `schedule` metaparameter:

```
notify { 'Doing some maintenance':
  schedule => 'outside-office-hours',
}
```

Without this `schedule` parameter, the resource would be applied every time Puppet runs. With it, Puppet will check the following parameters to decide whether or not to apply the resource:

- Whether the time is in the permitted range
- Whether the resource has already been run the maximum permitted number of times in this period

For example, let's consider what happens if Puppet runs at 4 p.m., 5 p.m., and 6 p.m. on a given day:

- **4 p.m.**: It's outside the permitted time range, so Puppet will do nothing
- **5 p.m.**: It's inside the permitted time range, and the resource hasn't been run yet in this period, so Puppet will apply the resource
- **6 p.m.**: It's inside the permitted time range, but the resource has already been run the maximum number of times in this period, so Puppet will do nothing

And so on until the next day.

There's more...

The `repeat` parameter governs how many times the resource will be applied given the other constraints of the schedule. For example, to apply a resource no more than six times an hour, use a schedule as follows:

```
period => hourly,
repeat => 6,
```

Remember that this won't guarantee that the job is run six times an hour. It just sets an upper limit; no matter how often Puppet runs or anything else happens, the job won't be run if it has already run six times this hour. If Puppet only runs once a day, the job will just be run once. So `schedule` is best used to make sure things don't happen at certain times (or don't exceed a given frequency).

Using host resources

It's not always practical or convenient to use DNS to map your machine names to IP addresses, especially in cloud infrastructures, where those addresses may change all the time. However, if you use entries in the `/etc/hosts` file instead, you then have the problem of how to distribute these entries to all machines and keep them up to date.

Here's a better way to do it; Puppet's host resource type controls a single `/etc/hosts` entry, and you can use this to map a hostname to an IP address easily across your whole network. For example, if all your machines need to know the address of the main database server, you can manage it with a `host` resource.

How to do it...

Follow these steps to create an example `host` resource:

1. Modify your `site.pp` file as follows:

   ```
   node 'cookbook' {
     host { 'packtpub.com':
       ensure => present,
       ip     => '83.166.169.231',
     }
   }
   ```

2. Run Puppet:

   ```
   [root@cookbook ~]# puppet agent -t
   Info: Caching catalog for cookbook.example.com
   Info: Applying configuration version '1413781153'
   Notice: /Stage[main]/Main/Node[cookbook]/Host[packtpub.com]/
   ensure: created
   Info: Computing checksum on file /etc/hosts
   Notice: Finished catalog run in 0.12 seconds
   ```

How it works...

Puppet will check the `target` file (usually `/etc/hosts`) to see whether the host entry already exists, and if not, add it. If an entry for that hostname already exists with a different address, Puppet will change the address to match the manifest.

There's more...

Organizing your host resources into classes can be helpful. For example, you could put the host resources for all your DB servers into one class called `admin::dbhosts`, which is included by all web servers.

Where machines may need to be defined in multiple classes (for example, a database server might also be a repository server), virtual resources can solve this problem. For example, you could define all your hosts as virtual in a single class:

```
class admin::allhosts {
  @host { 'db1.packtpub.com':
    tag => 'database'
    ...
  }
}
```

You could then realize the hosts you need in the various classes:

```
class admin::dbhosts {
  Host <| tag=='database' |>
}

class admin::webhosts {
  Host <| tag=='web' |>
}
```

Using exported host resources

In the previous example, we used the spaceship syntax to collect virtual host resources for hosts of type database or type web. You can use the same trick with exported resources. The advantage to using exported resources is that as you add more database servers, the collector syntax will automatically pull in the newly created exported host entries for those servers. This makes your `/etc/hosts` entries more dynamic.

Getting ready

We will be using exported resources. If you haven't already done so, set up puppetdb and enable storeconfigs to use puppetdb as outlined in *Chapter 2*, *Puppet Infrastructure*.

How to do it...

In this example, we will configure database servers and clients to communicate with each other. We'll make use of exported resources to do the configuration.

1. Create a new database module, `db`:

   ```
   t@mylaptop ~/puppet/modules $ mkdir -p db/manifests
   ```

2. Create a new class for your database servers, `db::server`:

   ```
   class db::server {
     @@host {"$::fqdn":
       host_aliases => $::hostname,
       ip           => $::ipaddress,
       tag          => 'db::server',
     }
     # rest of db class
   }
   ```

3. Create a new class for your database clients:

   ```
   class db::client {
     Host <<| tag == 'db::server' |>>
   }
   ```

4. Apply the database server module to some nodes, in `site.pp`, for example:

   ```
   node 'dbserver1.example.com' {
     class {'db::server': }
   }
   node 'dbserver2.example.com' {
     class {'db::server': }
   }
   ```

5. Run Puppet on the nodes with the database server module to create the exported resources.

6. Apply the database client module to cookbook:

   ```
   node 'cookbook' {
     class {'db::client': }
   }
   ```

7. Run Puppet:

```
[root@cookbook ~]# puppet agent -t
Info: Caching catalog for cookbook.example.com
Info: Applying configuration version '1413782501'
Notice: /Stage[main]/Db::Client/Host[dbserver2.example.com]/
ensure: created
Info: Computing checksum on file /etc/hosts
Notice: /Stage[main]/Db::Client/Host[dbserver1.example.com]/
ensure: created
Notice: Finished catalog run in 0.10 seconds
```

8. Verify the host entries in `/etc/hosts`:

```
[root@cookbook ~]# cat /etc/hosts
# HEADER: This file was autogenerated at Mon Oct 20 01:21:42 -0400
2014
# HEADER: by puppet.  While it can still be managed manually, it
# HEADER: is definitely not recommended.
127.0.0.1    localhost  localhost.localdomain localhost4
localhost4.localdomain4
::1  localhost  localhost.localdomain localhost6 localhost6.
localdomain6
83.166.169.231  packtpub.com
192.168.122.150  dbserver2.example.com  dbserver2
192.168.122.151  dbserver1.example.com  dbserver1
```

How it works...

In the `db::server` class, we create an exported host resource:

```
@@host {"$::fqdn":
  host_aliases => $::hostname,
  ip           => $::ipaddress,
  tag          => 'db::server',
}
```

This resource uses the fully qualified domain name (`$::fqdn`) of the node on which it is applied. We also use the short hostname (`$::hostname`) as an alias of the node. Aliases are printed after `fqdn` in `/etc/hosts`. We use the node's `$::ipaddress` fact as the IP address for the host entry. Finally, we add a tag to the resource so that we can collect based on that tag later.

The important thing to remember here is that if the IP address should change for the host, the exported resource will be updated, and nodes that collect the exported resource will update their host records accordingly.

We created a collector in `db::client`, which only collects exported host resources that have been tagged with `'db::server'`:

```
Host <<| tag == 'db::server' |>>
```

We applied the `db::server` class for a couple of nodes, dbserver1 and dbserver2, which we then collected on cookbook by applying the `db::client` class. The host entries were placed in `/etc/hosts` (the default file). We can see that the host entry contains both the fqdn and the short hostname for dbserver1 and dbserver2.

There's more...

Using exported resources in this manner is very useful. Another similar system would be to create an NFS server class, which creates exported resources for the mount points that it exports (via NFS). You can then use tags to have clients collect the appropriate mount points from the server. In the previous example, we made use of a tag to aid in our collection of exported resources. It is worth noting that there are several tags automatically added to resources when they are created, one of which is the scope where the resource was created.

Using multiple file sources

A neat feature of Puppet's `file` resource is that you can specify multiple values for the `source` parameter. Puppet will search them in order. If the first source isn't found, it moves on to the next, and so on. You can use this to specify a default substitute if the particular file isn't present, or even a series of increasingly generic substitutes.

How to do it...

This example demonstrates using multiple file sources:

1. Create a new greeting module as follows:

```
class greeting {
  file { '/tmp/greeting':
    source => [ 'puppet:///modules/greeting/hello.txt',
                'puppet:///modules/greeting/universal.txt'],
  }
}
```

2. Create the file `modules/greeting/files/hello.txt` with the following contents:

   ```
   Hello, world.
   ```

3. Create the file `modules/greeting/files/universal.txt` with the following contents:

   ```
   Bah-weep-Graaaaagnah wheep ni ni bong
   ```

4. Add the class to a node:

   ```
   node cookbook {
     class {'greeting': }
   }
   ```

5. Run Puppet:

   ```
   [root@cookbook ~]# puppet agent -t
   Info: Caching catalog for cookbook.example.com
   Info: Applying configuration version '1413784347'
   Notice: /Stage[main]/Greeting/File[/tmp/greeting]/ensure: defined
   content as '{md5}54098b367d2e87b078671fad4afb9dbb'
   Notice: Finished catalog run in 0.43 seconds
   ```

6. Check the contents of the `/tmp/greeting` file:

   ```
   [root@cookbook ~]# cat /tmp/greeting
   Hello, world.
   ```

7. Now remove the `hello.txt` file from your Puppet repository and rerun the agent:

   ```
   [root@cookbook ~]# puppet agent -t
   Info: Caching catalog for cookbook.example.com
   Info: Applying configuration version '1413784939'
   Notice: /Stage[main]/Greeting/File[/tmp/greeting]/content:
   --- /tmp/greeting   2014-10-20 01:52:28.117999991 -0400
   +++ /tmp/puppet-file20141020-4960-1o9g344-0    2014-10-20
   02:02:20.695999979 -0400
   @@ -1 +1 @@
   -Hello, world.
   +Bah-weep-Graaaaagnah wheep ni ni bong

   Info: Computing checksum on file /tmp/greeting
   Info: /Stage[main]/Greeting/File[/tmp/greeting]: Filebucketed /
   tmp/greeting to puppet with sum 54098b367d2e87b078671fad4afb9dbb
   ```

```
Notice: /Stage[main]/Greeting/File[/tmp/greeting]/content: content
changed '{md5}54098b367d2e87b078671fad4afb9dbb' to '{md5}933c7f04d
501b45456e830de299b5521'
Notice: Finished catalog run in 0.77 seconds
```

How it works...

On the first Puppet run, puppet searches for the available file sources in the order given:

```
source => [
  'puppet:///modules/greeting/hello.txt',
  'puppet:///modules/greeting/universal.txt'
],
```

The file `hello.txt` is first in the list, and is present, so Puppet uses that as the source for `/tmp/greeting`:

```
Hello, world.
```

On the second Puppet run, `hello.txt` is missing, so Puppet goes on to look for the next file, `universal.txt`. This is present, so it becomes the source for `/tmp/greeting`:

```
Bah-weep-Graaaaagnah wheep ni ni bong
```

There's more...

You can use this trick anywhere you have a `file` resource. A common example is a service that is deployed on all nodes, such as rsyslog. The `rsyslog` configuration is the same on every host except for the rsyslog server. Create an `rsyslog` class with a file resource for the `rsyslog` configuration file:

```
class rsyslog {
  file { '/etc/rsyslog.conf':
    source => [
      "puppet:///modules/rsyslog/rsyslog.conf.${::hostname}",
      'puppet:///modules/rsyslog/rsyslog.conf' ],
  }
```

Then, you put the default configuration in `rsyslog.conf`. For your rsyslog server, `logger`, create an `rsyslog.conf.logger` file. On the machine logger, `rsyslog.conf.logger` will be used before `rsyslog.conf` because it is listed first in the array of sources.

See also

- The *Passing parameters to classes* recipe in *Chapter 3*, *Writing Better Manifests*

Distributing and merging directory trees

As we saw in the previous chapter, the file resource has a `recurse` parameter, which allows Puppet to transfer entire directory trees. We used this parameter to copy an admin user's dotfiles into their home directory. In this section, we'll show how to use `recurse` and another parameter `sourceselect` to extend our previous example.

How to do it...

Modify our admin user example as follows:

1. Remove the `$dotfiles` parameter, remove the condition based on `$dotfiles`. Add a second source to the home directory `file` resource:

```
define admin_user ($key, $keytype) {
  $username = $name
  user { $username:
    ensure     => present,
  }
  file { "/home/${username}/.ssh":
    ensure  => directory,
    mode    => '0700',
    owner   => $username,
    group   => $username,
    require => File["/home/${username}"],
  }
  ssh_authorized_key { "${username}_key":
    key     => $key,
    type    => "$keytype",
    user    => $username,
    require => File["/home/${username}/.ssh"],
  }
  # copy in all the files in the subdirectory
  file { "/home/${username}":
    recurse => true,
    mode    => '0700',
    owner   => $username,
    group   => $username,
    source  => [
      "puppet:///modules/admin_user/${username}",
      'puppet:///modules/admin_user/base' ],
    sourceselect => 'all',
    require      => User["$username"],
  }
}
```

2. Create a base directory and copy all the system default files from `/etc/skel`:

```
t@mylaptop ~/puppet/modules/admin_user/files $ cp -a /etc/skel
base
```

3. Create a new `admin_user` resource, one that will not have a directory defined:

```
node 'cookbook' {
  admin_user {'steven':
    key     => 'AAAAB3N...',
    keytype => 'dsa',
  }
}
```

4. Run Puppet:

```
[root@cookbook ~]# puppet agent -t
Info: Caching catalog for cookbook.example.com
Info: Applying configuration version '1413787159'
Notice: /Stage[main]/Main/Node[cookbook]/Admin_user[steven]/
User[steven]/ensure: created
Notice: /Stage[main]/Main/Node[cookbook]/Admin_user[steven]/File[/
home/steven]/ensure: created
Notice: /Stage[main]/Main/Node[cookbook]/Admin_user[steven]/File[/
home/steven/.bash_logout]/ensure: defined content as '{md5}6a5bc1c
c5f80a48b540bc09d082b5855'
Notice: /Stage[main]/Main/Node[cookbook]/Admin_user[steven]/
File[/home/steven/.emacs]/ensure: defined content as '{md5}
de7ee35f4058681a834a99b5d1b048b3'
Notice: /Stage[main]/Main/Node[cookbook]/Admin_user[steven]/File[/
home/steven/.bashrc]/ensure: defined content as '{md5}2f8222b4f275
c4f18e69c34f66d2631b'
Notice: /Stage[main]/Main/Node[cookbook]/Admin_user[steven]/File[/
home/steven/.bash_profile]/ensure: defined content as '{md5}
f939eb71a81a9da364410b799e817202'
```

```
Notice: /Stage[main]/Main/Node[cookbook]/Admin_user[steven]/File[/
home/steven/.ssh]/ensure: created
Notice: /Stage[main]/Main/Node[cookbook]/Admin_user[steven]/Ssh_
authorized_key[steven_key]/ensure: created
Notice: Finished catalog run in 1.11 seconds
```

How it works...

If a `file` resource has the `recurse` parameter set on it, and it is a directory, Puppet will deploy not only the directory itself, but all its contents (including subdirectories and their contents). As we saw in the previous example, when a file has more than one source, the first source file found is used to satisfy the request. This applies to directories as well.

There's more...

By specifying the parameter `sourceselect` as 'all', the contents of all the source directories will be combined. For example, add `thomas admin_user` back into your node definition in `site.pp` for cookbook:

```
admin_user {'thomas':
    key     => 'ABBA...',
    keytype => 'rsa',
  }
```

Now run Puppet again on cookbook:

```
[root@cookbook thomas]# puppet agent -t
Info: Caching catalog for cookbook.example.com
Info: Applying configuration version '1413787770'
Notice: /Stage[main]/Main/Node[cookbook]/Admin_user[thomas]/File[/home/
thomas/.bash_profile]/content: content changed '{md5}3e8337f44f84b298a8a9
9869ae8ca76a' to '{md5}f939eb71a81a9da364410b799e817202'
Notice: /Stage[main]/Main/Node[cookbook]/Admin_user[thomas]/File[/home/
thomas/.bash_profile]/group: group changed 'root' to 'thomas'
Notice: /Stage[main]/Main/Node[cookbook]/Admin_user[thomas]/File[/home/
thomas/.bash_profile]/mode: mode changed '0644' to '0700'
Notice: /File[/home/thomas/.bash_profile]/seluser: seluser changed
'system_u' to 'unconfined_u'
Notice: /Stage[main]/Main/Node[cookbook]/Admin_user[thomas]/File[/home/
thomas/.bash_logout]/ensure: defined content as '{md5}6a5bc1cc5f80a48b540
bc09d082b5855'
```

```
Notice: /Stage[main]/Main/Node[cookbook]/Admin_user[thomas]/
File[/home/thomas/.bashrc]/content: content changed '{md5}
db2a20b2b9cdf36cca1ca4672622ddd2' to '{md5}033c3484e4b276e0641becc3aa268a
3a'
Notice: /Stage[main]/Main/Node[cookbook]/Admin_user[thomas]/File[/home/
thomas/.bashrc]/group: group changed 'root' to 'thomas'
Notice: /Stage[main]/Main/Node[cookbook]/Admin_user[thomas]/File[/home/
thomas/.bashrc]/mode: mode changed '0644' to '0700'
Notice: /File[/home/thomas/.bashrc]/seluser: seluser changed 'system_u'
to 'unconfined_u'
Notice: /Stage[main]/Main/Node[cookbook]/Admin_user[thomas]/
File[/home/thomas/.emacs]/ensure: defined content as '{md5}
de7ee35f4058681a834a99b5d1b048b3'
Notice: Finished catalog run in 0.86 seconds
```

Because we previously applied the `thomas admin_user` to cookbook, the user existed. The two files defined in the `thomas` directory on the Puppet server were already in the home directory, so only the additional files, `.bash_logout`, `.bash_profile`, and `.emacs` were created. Using these two parameters together, you can have default files that can be overridden easily.

Sometimes you want to deploy files to an existing directory but remove any files which aren't managed by Puppet. A good example would be if you are using `mcollective` in your environment. The directory holding client credentials should only have certificates that come from Puppet.

The `purge` parameter will do this for you. Define the directory as a resource in Puppet:

```
file { '/etc/mcollective/ssl/clients':
  purge   => true,
  recurse => true,
}
```

The combination of `recurse` and `purge` will remove all files and subdirectories in `/etc/mcollective/ssl/clients` that are not deployed by Puppet. You can then deploy your own files to that location by placing them in the appropriate directory on the Puppet server.

If there are subdirectories that contain files you don't want to purge, just define the subdirectory as a Puppet resource, and it will be left alone:

```
file { '/etc/mcollective/ssl/clients':
  purge => true,
  recurse => true,
}
file { '/etc/mcollective/ssl/clients/local':
  ensure => directory,
}
```

Be aware that, at least in current implementations of Puppet, recursive file copies can be quite slow and place a heavy memory load on the server. If the data doesn't change very often, it might be better to deploy and unpack a `tar` file instead. This can be done with a file resource for the `tar` file and an exec, which requires the file resource and unpacks the archive. Recursive directories are less of a problem when filled with small files. Puppet is not a very efficient file server, so creating large tar files and distributing them with Puppet is not a good idea either. If you need to copy large files around, using the Operating Systems packager is a better solution.

Cleaning up old files

Puppet's `tidy` resource will help you clean up old or out-of-date files, reducing disk usage. For example, if you have Puppet reporting enabled as described in the section on generating reports, you might want to regularly delete old report files.

How to do it...

Let's get started.

1. Modify your `site.pp` file as follows:

```
node 'cookbook' {
  tidy { '/var/lib/puppet/reports':
    age     => '1w',
    recurse => true,
  }
}
```

2. Run Puppet:

```
[root@cookbook clients]# puppet agent -t
Info: Caching catalog for cookbook.example.com
Notice: /Stage[main]/Main/Node[cookbook]/File[/var/lib/puppet/
reports/cookbook.example.com/201409090637.yaml]/ensure: removed
Notice: /Stage[main]/Main/Node[cookbook]/File[/var/lib/puppet/
reports/cookbook.example.com/201409100556.yaml]/ensure: removed
Notice: /Stage[main]/Main/Node[cookbook]/File[/var/lib/puppet/
reports/cookbook.example.com/201409090631.yaml]/ensure: removed
Notice: /Stage[main]/Main/Node[cookbook]/File[/var/lib/puppet/
reports/cookbook.example.com/201408210557.yaml]/ensure: removed
Notice: /Stage[main]/Main/Node[cookbook]/File[/var/lib/puppet/
reports/cookbook.example.com/201409080557.yaml]/ensure: removed
```

```
Notice: /Stage[main]/Main/Node[cookbook]/File[/var/lib/puppet/
reports/cookbook.example.com/201409100558.yaml]/ensure: removed
Notice: /Stage[main]/Main/Node[cookbook]/File[/var/lib/puppet/
reports/cookbook.example.com/201408210546.yaml]/ensure: removed
Notice: /Stage[main]/Main/Node[cookbook]/File[/var/lib/puppet/
reports/cookbook.example.com/201408210539.yaml]/ensure: removed
Notice: Finished catalog run in 0.80 seconds
```

How it works...

Puppet searches the specified path for any files matching the `age` parameter; in this case, `2w` (two weeks). It also searches subdirectories (`recurse => true`).

Any files matching your criteria will be deleted.

There's more...

You can specify file ages in seconds, minutes, hours, days, or weeks by using a single character to specify the time unit, as follows:

- `60s`
- `180m`
- `24h`
- `30d`
- `4w`

You can specify that files greater than a given size should be removed, as follows:

```
size => '100m',
```

This removes files of 100 megabytes and over. For kilobytes, use `k`, and for bytes, use `b`.

Note that if you specify both age and size parameters, they are treated as independent criteria. For example, if you specify the following, Puppet will remove all files that are either at least one day old, or at least 512 KB in size:

```
age  => "1d",
size => "512k",
```

Auditing resources

Dry run mode, using the `--noop` switch, is a simple way to audit any changes to a machine under Puppet's control. However, Puppet also has a dedicated audit feature, which can report changes to resources or specific attributes.

How to do it...

Here's an example showing Puppet's auditing capabilities:

1. Modify your `site.pp` file as follows:

```
node 'cookbook' {
  file { '/etc/passwd':
    audit => [ owner, mode ],
  }
}
```

2. Run Puppet:

```
[root@cookbook clients]# puppet agent -t
Info: Caching catalog for cookbook.example.com
Info: Applying configuration version '1413789080'
Notice: /Stage[main]/Main/Node[cookbook]/File[/etc/passwd]/owner:
audit change: newly-recorded value 0
Notice: /Stage[main]/Main/Node[cookbook]/File[/etc/passwd]/mode:
audit change: newly-recorded value 644
Notice: Finished catalog run in 0.55 seconds
```

How it works...

The `audit` metaparameter tells Puppet that you want to record and monitor certain things about the resource. The value can be a list of the parameters that you want to audit.

In this case, when Puppet runs, it will now record the owner and mode of the `/etc/passwd` file. In future runs, Puppet will spot whether either of these has changed. For example, if you run:

```
[root@cookbook ~]# chmod 666 /etc/passwd
```

Puppet will pick up this change and log it on the next run:

```
Notice: /Stage[main]/Main/Node[cookbook]/File[/etc/passwd]/mode: audit
change: previously recorded value 0644 has been changed to 0666
```

There's more...

This feature is very useful to audit large networks for any changes to machines, either malicious or accidental. It's also very handy to keep an eye on things that aren't managed by Puppet, for example, application code on production servers. You can read more about Puppet's auditing capability here:

`http://puppetlabs.com/blog/all-about-auditing-with-puppet/`

If you just want to audit everything about a resource, use `all`:

```
file { '/etc/passwd':
  audit => all,
}
```

See also

- The *Noop - the don't change anything option* recipe in *Chapter 10*, *Monitoring, Reporting, and Troubleshooting*

Temporarily disabling resources

Sometimes you want to disable a resource for the time being so that it doesn't interfere with other work. For example, you might want to tweak a configuration file on the server until you have the exact settings you want, before checking it into Puppet. You don't want Puppet to overwrite it with an old version in the meantime, so you can set the `noop` metaparameter on the resource:

```
noop => true,
```

How to do it...

This example shows you how to use the `noop` metaparameter:

1. Modify your `site.pp` file as follows:

```
node 'cookbook' {
  file { '/etc/resolv.conf':
    content => "nameserver 127.0.0.1\n",
    noop    => true,
  }
}
```

2. Run Puppet:

```
[root@cookbook ~]# puppet agent -t
Info: Caching catalog for cookbook.example.com
Info: Applying configuration version '1413789438'
Notice: /Stage[main]/Main/Node[cookbook]/File[/etc/resolv.conf]/
content:
--- /etc/resolv.conf  2014-10-20 00:27:43.095999975 -0400
+++ /tmp/puppet-file20141020-8439-11huy1y-0   2014-10-20
03:17:18.969999979 -0400
@@ -1,3 +1 @@
-; generated by /sbin/dhclient-script
-search example.com
-nameserver 192.168.122.1
+nameserver 127.0.0.1

Notice: /Stage[main]/Main/Node[cookbook]/File[/etc/resolv.conf]/
content: current_value {md5}4c0d192511df253826d302bc830a371b,
should be {md5}949343428bded6a653a85910f6bdb48e (noop)
Notice: Node[cookbook]: Would have triggered 'refresh' from 1
events
Notice: Class[Main]: Would have triggered 'refresh' from 1 events
Notice: Stage[main]: Would have triggered 'refresh' from 1 events
Notice: Finished catalog run in 0.50 seconds
```

How it works...

The `noop` metaparameter is set to `true`, so for this particular resource, it's as if you had to run Puppet with the `--noop` flag. Puppet noted that the resource would have been applied, but otherwise did nothing.

The nice thing with running the agent in test mode (`-t`) is that Puppet output a diff of what it would have done if the `noop` was not present (you can tell puppet to show the diff's without using `-t` with `--show_diff`; `-t` implies many different settings):

```
--- /etc/resolv.conf  2014-10-20 00:27:43.095999975 -0400
+++ /tmp/puppet-file20141020-8439-1lhuy1y-0    2014-10-20 
03:17:18.969999979 -0400
@@ -1,3 +1 @@
-; generated by /sbin/dhclient-script
-search example.com
-nameserver 192.168.122.1
+nameserver 127.0.0.1
```

This can be very useful when debugging a template; you can work on your changes and then see what they would look like on the node without actually applying them. Using the diff, you can see whether your updated template produces the correct output.

7
Managing Applications

Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?

— Brian W. Kernighan.

In this chapter, we will cover the following recipes:

- Using public modules
- Managing Apache servers
- Creating Apache virtual hosts
- Creating nginx virtual hosts
- Managing MySQL
- Creating databases and users

Introduction

Without applications, a server is just a very expensive space heater. In this chapter, I'll present some recipes to manage some specific software with Puppet: MySQL, Apache, **nginx**, and Ruby. I hope the recipes will be useful to you in themselves. However, the patterns and techniques they use are applicable to almost any software, so you can adapt them to your own purposes without much difficulty. One thing that is common about these applications, they are common. Most Puppet installations will have to deal with a web server, Apache or nginx. Most, if not all, will have databases and some of those will have MySQL. When everyone has to deal with a problem, community solutions are generally better tested and more thorough than homegrown solutions. We'll use modules from the Puppet Forge in this chapter to manage these applications.

When you are writing your own Apache or nginx modules from scratch, you'll have to pay attention to the nuances of the distributions you support. Some distributions call the apache package `httpd`, while others use `apache2`; the same can be said for MySQL. In addition, Debian-based distributions use an enabled folder method to enable custom sites in Apache, which are virtual sites, whereas RPM based distributions do not. For more information on virtual sites, visit `http://httpd.apache.org/docs/2.2/vhosts/`.

Using public modules

When you write a Puppet module to manage some software or service, you don't have to start from scratch. Community-contributed modules are available at the Puppet Forge site for many popular applications. Sometimes, a community module will be exactly what you need and you can download and start using it straight away. In most cases, you will need to make some modifications to suit your particular needs and environment.

Like all community efforts, there are some excellent and some less than excellent modules on the Forge. You should read the README section of the module and decide whether the module is going to work in your installation. At the least, ensure that your distribution is supported. Puppetlabs has introduced a set of modules that are supported, that is, if you are an enterprise customer, they will support your use of the module in your installation. Additionally, most Forge modules deal with multiple operating systems, distributions, and a great number of use cases. In many cases, not using a forge module is like reinventing the wheel. One caveat though is that Forge modules may be more complex than your local modules. You should read the code and get a sense of what the module is doing. Knowing how the module works will help you debug it later.

How to do it...

In this example, we'll use the `puppet module` command to find and install the useful `stdlib` module, which contains many utility functions to help you develop Puppet code. It is one of the aforementioned supported modules by puppetlabs. I'll download the module into my user's home directory and manually install it in the Git repository. To install puppetlabs stdlib module, follow these steps:

1. Run the following command:

```
t@mylaptop ~ $ puppet module search puppetlabs-stdlib
Notice: Searching https://forgeapi.puppetlabs.com ...
NAME                DESCRIPTION                        AUTHOR
KEYWORDS
puppetlabs-stdlib   Puppet Module Standard Library     @puppetlabs
stdlib stages
```

2. We verified that we have the right module, so we'll install it with `module install` now:

```
t@mylaptop ~ $ puppet module install puppetlabs-stdlib
Notice: Preparing to install into /home/thomas/.puppet/modules ...
Notice: Downloading from https://forgeapi.puppetlabs.com ...
Notice: Installing -- do not interrupt ...
/home/thomas/.puppet/modules
└── puppetlabs-stdlib (v4.3.2)
```

3. The module is now ready to use in your manifests; most good modules come with a `README` file to show you how to do this.

How it works...

You can search for modules that match the package or software you're interested in with the `puppet module search` command. To install a specific module, use `puppet module install`. You can add the `-i` option to tell Puppet where to find your module directory.

You can browse the forge to see what's available at `http://forge.puppetlabs.com/`.

More information on supported modules is available at `https://forge.puppetlabs.com/supported`.

The current list of supported modules is available at `https://forge.puppetlabs.com/modules?endorsements=supported`.

There's more...

Modules on the Forge include a `metadata.json` file, which describes the module and which operating systems the module supports. This file also includes a list of modules that are required by the module.

This file was previously named Modulefile and not in JSON format; the old Modulefile format was deprecated in Version 3.6.

As we will see in our next section, when installing a module from the Forge, the required dependencies will automatically be installed as well.

Not all publically available modules are on Puppet Forge. Some other great places to look at on GitHub are:

- `https://github.com/camptocamp`
- `https://github.com/example42`

Though not a collection of modules as such, the Puppet Cookbook website has many useful and illuminating code examples, patterns, and tips, maintained by the admirable Dean Wilson:

`http://www.puppetcookbook.com/`

Managing Apache servers

Apache is the world's favorite web server, so it's highly likely that part of your Puppetly duties will include installing and managing Apache.

How to do it...

We'll install and use the `puppetlabs-apache` module to install and start Apache. This time, when we run `puppet module install`, we'll use the `-i` option to tell Puppet to install the module in our Git repository's module's directory.

1. Install the module using `puppet modules install`:

```
t@mylaptop ~/puppet $ puppet module install -i modules puppetlabs-
apache
Notice: Preparing to install into /home/thomas/puppet/modules ...
Notice: Downloading from https://forgeapi.puppetlabs.com ...
Notice: Installing -- do not interrupt ...
/home/thomas/puppet/modules
└─┬ puppetlabs-apache (v1.1.1)
  ├── puppetlabs-concat (v1.1.1)
  └── puppetlabs-stdlib (v4.3.2)
```

2. Add the modules to your Git repository and push them out:

```
t@mylaptop ~/puppet $ git add modules/apache modules/concat
modules/stdlib
t@mylaptop ~/puppet $ git commit -m "adding puppetlabs-apache
module"
[production 395b079] adding puppetlabs-apache module
 647 files changed, 35017 insertions(+), 13 deletions(-)
 rename modules/{apache => apache.cookbook}/manifests/init.pp
(100%)
 create mode 100644 modules/apache/CHANGELOG.md
 create mode 100644 modules/apache/CONTRIBUTING.md
...
t@mylaptop ~/puppet $ git push origin production
Counting objects: 277, done.
Delta compression using up to 4 threads.
```

```
Compressing objects: 100% (248/248), done.
Writing objects: 100% (266/266), 136.25 KiB | 0 bytes/s, done.
Total 266 (delta 48), reused 0 (delta 0)
remote: To puppet@puppet.example.com:/etc/puppet/environments/
puppet.git
remote:    9faaa16..395b079  production -> production
```

3. Create a web server node definition in `site.pp`:

```
node webserver {
  class {'apache': }
}
```

4. Run Puppet to apply the default Apache module configuration:

```
[root@webserver ~]# puppet agent -t
Info: Caching certificate for webserver.example.com
Notice: /File[/var/lib/puppet/lib/puppet/provider/a2mod]/ensure:
created
...
Info: Caching catalog for webserver.example.com
...
Info: Class[Apache::Service]: Scheduling refresh of Service[httpd]
Notice: /Stage[main]/Apache::Service/Service[httpd]: Triggered
'refresh' from 51 events
Notice: Finished catalog run in 11.73 seconds
```

5. Verify that you can reach `webserver.example.com`:

```
[root@webserver ~]# curl http://webserver.example.com
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<html>
 <head>
  <title>Index of /</title>
 </head>
 <body>
<h1>Index of /</h1>
<table><tr><th><img src="/icons/blank.gif" alt="[ICO]"></
th><th><a href="?C=N;O=D">Name</a></th><th><a href="?C=M;O=A">Last
modified</a></th><th><a href="?C=S;O=A">Size</a></
th><th><a href="?C=D;O=A">Description</a></th></tr><tr><th
colspan="5"><hr></th></tr>
<tr><th colspan="5"><hr></th></tr>
</table>
</body></html>
```

How it works...

Installing the puppetlabs-Apache module from the Forge causes both puppetlabs-concat and puppetlabs-stdlib to be installed into our modules directory. The concat module is used to stitch snippets of files together in a specific order. It is used by the Apache module to create the main Apache configuration files.

We then defined a web server node and applied the Apache class to that node. We used all the default values and let the Apache module configure our server to be an Apache web server.

The Apache module then went and rewrote all our Apache configurations. By default, the module purges all the configuration files from the Apache directory (`/etc/apache2` or `/etc/httpd` depending on the distribution). The module can configure many different distributions and handle the nuances of each distribution. As a user of the module, you don't need to know how your distribution deals with the Apache module configuration.

After purging and rewriting the configuration files, the module ensures that the apache2 service is running (`httpd` on Enterprise Linux (EL)).

We then tested the webserver using curl. There was nothing returned but an empty index page. This is the expected behavior. Normally, when we install Apache on a server, there are some files that display a default page (`welcome.conf` on EL-based systems), since the module purged those configurations, we only see an empty page.

In a production environment, you would modify the defaults applied by the Apache module; the suggested configuration from the README is as follows:

```
class { 'apache':
  default_mods        => false,
  default_confd_files => false,
}
```

Creating Apache virtual hosts

Apache virtual hosts are created with the `apache` module with the defined type `apache::vhost`. We will create a new vhost on our Apache webserver called **navajo**, one of the apache tribes.

How to do it...

Follow these steps to create Apache virtual hosts:

1. Create a navajo `apache::vhost` definition as follows:

```
apache::vhost { 'navajo.example.com':
    port          => '80',
    docroot => '/var/www/navajo',
  }
```

2. Create an index file for the new vhost:

```
file {'/var/www/navajo/index.html':
    content => "<html>\nnavajo.example.com\nhttp://en.wikipedia.
org/wiki/Navajo_people\n</html>\n",
    mode    => '0644',
    require => Apache::Vhost['navajo.example.com']
  }
```

3. Run Puppet to create the new vhost:

```
[root@webserver ~]# puppet agent -t
Info: Caching catalog for webserver.example.com
Info: Applying configuration version '1414475598'
Notice: /Stage[main]/Main/Node[webserver]/Apache::Vhost[navajo.
example.com]/File[/var/www/navajo]/ensure: created
Notice: /Stage[main]/Main/Node[webserver]/Apache::Vhost[navajo.
example.com]/File[25-navajo.example.com.conf]/ensure: created
Info: /Stage[main]/Main/Node[webserver]/Apache::Vhost[navajo.
example.com]/File[25-navajo.example.com.conf]: Scheduling refresh
of Service[httpd]
Notice: /Stage[main]/Main/Node[webserver]/File[/var/www/navajo/
index.html]/ensure: defined content as '{md5}5212fe215f4c0223fb861
02a34319cc6'
Notice: /Stage[main]/Apache::Service/Service[httpd]: Triggered
'refresh' from 1 events
Notice: Finished catalog run in 2.73 seconds
```

4. Verify that you can reach the new virtual host:

```
[root@webserver ~]# curl http://navajo.example.com
<html>
navajo.example.com
http://en.wikipedia.org/wiki/Navajo_people
</html>
```

How it works...

The `apache::vhost` defined type creates a virtual host configuration file for Apache, `25-navajo.example.com.conf`. The file is created with a template; `25` at the beginning of the filename is the "priority level" of this virtual host. When Apache first starts, it reads through its configuration directory and starts executing files in an alphabetical order. Files that begin with numbers are read before files that start with letters. In this way, the Apache module ensures that the virtual hosts are read in a specific order, which can be specified when you define the virtual host. The contents of this file are as follows:

```
# ************************************
# Vhost template in module puppetlabs-apache
# Managed by Puppet
# ************************************

<VirtualHost *:80>
  ServerName navajo.example.com

  ## Vhost docroot
  DocumentRoot "/var/www/navajo"



  ## Directories, there should at least be a declaration for
    /var/www/navajo


  <Directory "/var/www/navajo">
    Options Indexes FollowSymLinks MultiViews
    AllowOverride None
    Order allow,deny
    Allow from all
  </Directory>

  ## Load additional static includes


  ## Logging
  ErrorLog "/var/log/httpd/navajo.example.com_error.log"
  ServerSignature Off
  CustomLog "/var/log/httpd/navajo.example.com_access.log"
    combined


</VirtualHost>
```

As you can see, the default file has created log files, set up directory access permissions and options, in addition to specifying the listen port and DocumentRoot.

The vhost definition creates the DocumentRoot directory, specified as 'root' to the apache::virtual definition. The directory is created before the virtual host configuration file; after that file has been created, a notify trigger is sent to the Apache process to restart.

Our manifest included a file that required the Apache::Vhost['navajo.example.com'] resource; our file was then created after the directory and the virtual host configuration file.

When we run curl on the new website (if you haven't created a hostname alias in DNS, you will have to create one in your local /etc/hosts file for navajo.example.com, or specify the host as curl -H 'Host: navajo.example.com' <ipaddress of navajo.example.com>), we see the contents of the index file we created:

```
file {'/var/www/navajo/index.html':
    content => "<html>\nnavajo.example.com\nhttp://en.wikipedia.org/
wiki/Navajo_people\n</html>\n",
    mode    => '0644',
    require => Apache::Vhost['navajo.example.com']
  }
[root@webserver ~]# curl http://navajo.example.com
<html>
navajo.example.com
http://en.wikipedia.org/wiki/Navajo_people
<\html>
```

There's more...

Both the defined type and the template take into account a multitude of possible configuration scenarios for virtual hosts. It is highly unlikely that you will find a setting that is not covered by this module. You should look at the definition for apache::virtual and the sheer number of possible arguments.

The module also takes care of several settings for you. For instance, if we change the listen port on our navajo virtual host from 80 to 8080, the module will make the following changes in /etc/httpd/conf.d/ports.conf:

```
Listen 80
+Listen 8080
 NameVirtualHost *:80
+NameVirtualHost *:8080
```

And in our virtual host file:

```
-<VirtualHost *:80>
+<VirtualHost *:8080>
```

So that we can now curl on port `8080` and see the same results:

```
[root@webserver ~]# curl http://navajo.example.com:8080
<html>
navajo.example.com
http://en.wikipedia.org/wiki/Navajo_people
</html>
```

And when we try on port `80`:

```
[root@webserver ~]# curl http://navajo.example.com
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<html>
 <head>
  <title>Index of /</title>
 </head>
 <body>
<h1>Index of /</h1>
<table><tr><th><img src="/icons/blank.gif" alt="[ICO]"></
th><th><a href="?C=N;O=D">Name</a></th><th><a href="?C=M;O=A">Last
modified</a></th><th><a href="?C=S;O=A">Size</a></th><th><a
href="?C=D;O=A">Description</a></th></tr><tr><th colspan="5"><hr></th></
tr>
<tr><th colspan="5"><hr></th></tr>
</table>
</body>
</html>
```

As we can see, the virtual host is no longer listening on port `80` and we receive the default empty directory listing we saw in our earlier example.

Creating nginx virtual hosts

Nginx is a fast, lightweight web server that is preferred over Apache in many contexts, especially where high performance is important. Nginx is configured slightly differently than Apache; like Apache though, there is a Forge module that can be used to configure nginx for us. Unlike Apache, however, the module that is suggested for use is not supplied by puppetlabs but by James Fryman. This module uses some interesting tricks to configure itself. Previous versions of this module used R.I. Pienaar's `module_data` package. This package is used to configure hieradata within a module. It's used to supply default values to the nginx module. I wouldn't recommend starting out with this module at this point, but it is a good example of where module configuration may be headed in the future. Giving modules the ability to modify hieradata may prove useful.

How to do it...

In this example, we'll use a Forge module to configure nginx. We'll download the module and use it to configure virtualhosts.

1. Download the `jfryman-nginx` module from the Forge:

```
t@mylaptop ~ $ cd ~/puppet
t@mylaptop ~/puppet $ puppet module install -i modules jfryman-
nginx
Notice: Preparing to install into /home/thomas/puppet/modules ...
Notice: Downloading from https://forgeapi.puppetlabs.com ...
Notice: Installing -- do not interrupt ...
/home/thomas/puppet/modules
└┬ jfryman-nginx (v0.2.1)
  ├── puppetlabs-apt (v1.7.0)
  ├── puppetlabs-concat (v1.1.1)
  └── puppetlabs-stdlib (v4.3.2)
```

2. Replace the definition for webserver with an nginx configuration:

```
node webserver {
  class {'nginx':}
  nginx::resource::vhost { 'mescalero.example.com':
      www_root => '/var/www/mescalero',
  }
  file {'/var/www/mescalero':
    ensure  => 'directory''directory',
    mode    => '0755',
    require => Nginx::Resource::Vhost['mescalero.example.com'],
  }
  file {'/var/www/mescalero/index.html':
    content => "<html>\nmescalero.example.com\nhttp://
en.wikipedia.org/wiki/Mescalero\n</html>\n",
    mode    => 0644,
    require => File['/var/www/mescalero'],
  }
}
```

3. If apache is still running on your webserver, stop it:

```
[root@webserver ~]# puppet resource service httpd ensure=false
Notice: /Service[httpd]/ensure: ensure changed 'running' to
'stopped'
service { 'httpd':
  ensure => 'stopped',
}
```

```
Run puppet agent on your webserver node:
[root@webserver ~]# puppet agent -t
Info: Caching catalog for webserver.example.com
Info: Applying configuration version '1414561483'
Notice: /Stage[main]/Main/Node[webserver]/Nginx::Resource::Vhost[m
escalero.example.com]/Concat[/etc/nginx/sites-available/mescalero.
example.com.conf]/File[/etc/nginx/sites-available/mescalero.
example.com.conf]/ensure: defined content as '{md5}35bb59bfcd0cf5a
549d152aaec284357'
Info: /Stage[main]/Main/Node[webserver]/Nginx::Resource::Vhost[me
scalero.example.com]/Concat[/etc/nginx/sites-available/mescalero.
example.com.conf]/File[/etc/nginx/sites-available/mescalero.
example.com.conf]: Scheduling refresh of Class[Nginx::Service]
Info: Concat[/etc/nginx/sites-available/mescalero.example.com.
conf]: Scheduling refresh of Class[Nginx::Service]
Notice: /Stage[main]/Main/Node[webserver]/Nginx::Resource::Vhost[
mescalero.example.com]/File[mescalero.example.com.conf symlink]/
ensure: created
Info: /Stage[main]/Main/Node[webserver]/Nginx::Resource::Vhost[m
escalero.example.com]/File[mescalero.example.com.conf symlink]:
Scheduling refresh of Service[nginx]
Notice: /Stage[main]/Main/Node[webserver]/File[/var/www/
mescalero]/ensure: created
Notice: /Stage[main]/Main/Node[webserver]/File[/var/www/mescalero/
index.html]/ensure: defined content as '{md5}2bd618c7dc3a3addc9e27
c2f3cfde294'
Notice: /Stage[main]/Nginx::Config/File[/etc/nginx/conf.d/
proxy.conf]/ensure: defined content as '{md5}1919fd65635d4965327
3e14028888617'
Info: Computing checksum on file /etc/nginx/conf.d/example_ssl.
conf
Info: /Stage[main]/Nginx::Config/File[/etc/nginx/conf.d/example_
ssl.conf]: Filebucketed /etc/nginx/conf.d/example_ssl.conf to
puppet with sum 84724f296c7056157d531d6b1215b507
Notice: /Stage[main]/Nginx::Config/File[/etc/nginx/conf.d/example_
ssl.conf]/ensure: removed
Info: Computing checksum on file /etc/nginx/conf.d/default.conf
Info: /Stage[main]/Nginx::Config/File[/etc/nginx/conf.d/default.
conf]: Filebucketed /etc/nginx/conf.d/default.conf to puppet with
sum 4dce452bf8dbb01f278ec0ea9ba6cf40
Notice: /Stage[main]/Nginx::Config/File[/etc/nginx/conf.d/default.
conf]/ensure: removed
Info: Class[Nginx::Config]: Scheduling refresh of
Class[Nginx::Service]
```

```
Info: Class[Nginx::Service]: Scheduling refresh of Service[nginx]
Notice: /Stage[main]/Nginx::Service/Service[nginx]: Triggered 
'refresh' from 2 events
Notice: Finished catalog run in 28.98 seconds
```

4. Verify that you can reach the new virtualhost:

```
[root@webserver ~]# curl mescalero.example.com
<html>
mescalero.example.com
http://en.wikipedia.org/wiki/Mescalero
</html>
```

How it works...

Installing the `jfryman-nginx` module causes the concat, stdlib, and APT modules to be installed. We run Puppet on our master to have the plugins created by these modules added to our running master. The stdlib and concat have facter and Puppet plugins that need to be installed for the nginx module to work properly.

With the plugins synchronized, we can then run puppet agent on our web server. As a precaution, we stop Apache if it was previously started (we can't have nginx and Apache both listening on port `80`). After puppet agent runs, we verified that nginx was running and the virtual host was configured.

There's more...

This nginx module is under active development. There are several interesting solutions employed with the module. Previous releases used the `ripienaar-module_data` module, which allows a module to include default values for its various attributes, via a hiera plugin. Although still in an early stage of development, this system is already usable and represents one of the cutting-edge modules on the Forge.

In the next section, we'll use a supported module to configure and manage MySQL installations.

Managing MySQL

MySQL is a very widely used database server, and it's fairly certain you'll need to install and configure a MySQL server at some point. The `puppetlabs-mysql` module can simplify your MySQL deployments.

How to do it...

Follow these steps to create the example:

1. Install the `puppetlabs-mysql` module:

```
t@mylaptop ~/puppet $ puppet module install -i modules puppetlabs-
mysql
Notice: Preparing to install into /home/thomas/puppet/modules ...
Notice: Downloading from https://forgeapi.puppetlabs.com ...
Notice: Installing -- do not interrupt ...
/home/thomas/puppet/modules
└─┬ puppetlabs-mysql (v2.3.1)
  └── puppetlabs-stdlib (v4.3.2)
```

2. Create a new node definition for your MySQL server:

```
node dbserver {
  class { '::mysql::server':
    root_password    => 'PacktPub',
    override_options => {
      'mysqld' => { 'max_connections' => '1024' }
    }
  }
}
```

3. Run Puppet to install the database server and apply the new root password:

```
[root@dbserver ~]# puppet agent -t
Info: Caching catalog for dbserver.example.com
Info: Applying configuration version '1414566216'
Notice: /Stage[main]/Mysql::Server::Install/Package[mysql-server]/
ensure: created
Notice: /Stage[main]/Mysql::Server::Service/Service[mysqld]/
ensure: ensure changed 'stopped' to 'running'
Info: /Stage[main]/Mysql::Server::Service/Service[mysqld]:
Unscheduling refresh on Service[mysqld]
Notice: /Stage[main]/Mysql::Server::Root_password/Mysql_
user[root@localhost]/password_hash: defined 'password_hash' as
'*6ABB0D4A7D1381BAEE4D078354557D495ACFC059'
Notice: /Stage[main]/Mysql::Server::Root_password/File[/root/.
my.cnf]/ensure: defined content as '{md5}87bc129b137c9d613e9f31c80
ea5426c'
Notice: Finished catalog run in 35.50 seconds
```

4. Verify that you can connect to the database:

```
[root@dbserver ~]# mysql
```

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 11
Server version: 5.1.73 Source distribution

Copyright (c) 2000, 2013, Oracle and/or its affiliates. All rights
reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current
input statement.

mysql>
```

How it works...

The MySQL module installs the MySQL server and ensures that the server is running. It then configures the root password for MySQL. The module does a lot of other things for you as well. It creates a `.my.cnf` file with the root user password. When we run the `mysql` client, the `.my.cnf` file sets all the defaults, so we do not need to supply any arguments.

There's more...

In the next section, we'll show how to create databases and users.

Creating databases and users

Managing a database means more than ensuring that the service is running; a database server is nothing without databases. Databases need users and privileges. Privileges are handled with `GRANT` statements. We will use the `puppetlabs-mysql` package to create a database and a user with access to that database. We'll create a MySQL user Drupal and a database called Drupal. We'll create a table named nodes and place data into that table.

How to do it...

Follow these steps to create databases and users:

1. Create a database definition within your `dbserver` class:

```
mysql::db { 'drupal':
    host     => 'localhost',
    user    => 'drupal',
    password    => 'Cookbook',
    sql     => '/root/drupal.sql',
    require => File['/root/drupal.sql']
  }

  file { '/root/drupal.sql':
    ensure => present,
    source => 'puppet:///modules/mysql/drupal.sql',
  }
```

2. Allow the Drupal user to modify the nodes table:

```
mysql_grant { 'drupal@localhost/drupal.nodes':
    ensure     => 'present',
    options    => ['GRANT'],
    privileges => ['ALL'],
    table      => 'drupal.nodes'nodes',
    user       => 'drupal@localhost',
  }
```

3. Create the `drupal.sql` file with the following contents:

```
CREATE TABLE users (id INT PRIMARY KEY AUTO_INCREMENT, title
VARCHAR(255), body TEXT);
INSERT INTO users (id, title, body) VALUES (1,'First
Node','Contents of first Node');
INSERT INTO users (id, title, body) VALUES (2,'Second
Node','Contents of second Node');
```

4. Run Puppet to have user, database, and `GRANT` created:

```
[root@dbserver ~]# puppet agent -t
Info: Caching catalog for dbserver.example.com
Info: Applying configuration version '1414648818'
Notice: /Stage[main]/Main/Node[dbserver]/File[/root/drupal.sql]/
ensure: defined content as '{md5}780f3946cfc0f373c6d4146394650f6b'
Notice: /Stage[main]/Main/Node[dbserver]/Mysql_grant[drupal@
localhost/drupal.nodes]/ensure: created
Notice: /Stage[main]/Main/Node[dbserver]/Mysql::Db[drupal]/Mysql_
user[drupal@localhost]/ensure: created
Notice: /Stage[main]/Main/Node[dbserver]/Mysql::Db[drupal]/Mysql_
database[drupal]/ensure: created
Info: /Stage[main]/Main/Node[dbserver]/Mysql::Db[drupal]/Mysql_
database[drupal]: Scheduling refresh of Exec[drupal-import]
Notice: /Stage[main]/Main/Node[dbserver]/Mysql::Db[drupal]/Mysql_
grant[drupal@localhost/drupal.*]/ensure: created
```

```
Notice: /Stage[main]/Main/Node[dbserver]/Mysql::Db[drupal]/
Exec[drupal-import]: Triggered 'refresh' from 1 events
Notice: Finished catalog run in 10.06 seconds
```

5. Verify that the database and table have been created:

```
[root@dbserver ~]# mysql drupal
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 34
Server version: 5.1.73 Source distribution

Copyright (c) 2000, 2013, Oracle and/or its affiliates. All rights
reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current
input statement.

mysql> show tables;
+------------------+
| Tables_in_drupal |
+------------------+
| users            |
+------------------+
1 row in set (0.00 sec)
```

6. Now, verify that our default data has been loaded into the table:

```
mysql> select * from users;
+----+-------------+-------------------------+
| id | title       | body                    |
+----+-------------+-------------------------+
|  1 | First Node  | Contents of first Node  |
|  2 | Second Node | Contents of second Node |
+----+-------------+-------------------------+
2 rows in set (0.00 sec)
```

How it works...

We start with the definition of the new drupal database:

```
mysql::db { 'drupal':
  host     => 'localhost',
  user     => 'drupal',
  password     => 'Cookbook',
  sql      => '/root/drupal.sql',
  require => File['/root/drupal.sql']
}
```

We specify that we'll connect from localhost (we could connect to the database from another server) using the drupal user. We give the password for the user and specify a SQL file that will be applied to the database after the database has been created. We require that this file already exist and define the file next:

```
file { '/root/drupal.sql':
    ensure => present,
    source => 'puppet:///modules/mysql/drupal.sql',
  }
```

We then ensure that the user has the appropriate privileges with a `mysql_grant` statement:

```
mysql_grant { 'drupal@localhost/drupal.nodes':
  ensure     => 'present',
  options    => ['GRANT'],
  privileges => ['ALL'],
  table      => 'drupal.nodes',
  user       => 'drupal@localhost',
}
```

There's more...

Using the puppetlabs-MySQL and puppetlabs-Apache module, we can create an entire functioning web server. The puppetlabs-Apache module will install Apache, and we can include the PHP module and MySQL module as well. We can then use the puppetlabs-Mysql module to install the MySQL server, and then create the required drupal databases and seed the database with the data.

Deploying a new drupal installation would be as simple as including a class on a node.

8
Internode Coordination

"Rest is not idleness, and to lie sometimes on the grass under trees on a summer's day, listening to the murmur of the water, or watching the clouds float across the sky, is by no means a waste of time."

— John Lubbock

In this chapter, we will cover the following recipes:

- Managing firewalls with iptables
- Building high-availability services using Heartbeat
- Managing NFS servers and file shares
- Using HAProxy to load-balance multiple web servers
- Managing Docker with Puppet

Introduction

As powerful as Puppet is to manage the configuration of a single server, it's even more useful when coordinating many machines. In this chapter, we'll explore ways to use Puppet to help you create high-availability clusters, share files across your network, set up automated firewalls, and use load-balancing to get more out of the machines you have. We'll use exported resources as the communication between nodes.

Managing firewalls with iptables

In this chapter, we will begin to configure services that require communication between hosts over the network. Most Linux distributions will default to running a host-based firewall, **iptables**. If you want your hosts to communicate with each other, you have two options: turn off iptables or configure iptables to allow the communication.

I prefer to leave iptables turned on and configure access. Keeping iptables is just another layer on your defense across the network. iptables isn't a magic bullet that will make your system secure, but it will block access to services you didn't intend to expose to the network.

Configuring iptables properly is a complicated task, which requires deep knowledge of networking. The example presented here is a simplification. If you are unfamiliar with iptables, I suggest you research iptables before continuing. More information can be found at `http://wiki.centos.org/HowTos/Network/IPTables` or `https://help.ubuntu.com/community/IptablesHowTo`.

Getting ready

In the following examples, we'll be using the Puppet Labs Firewall module to configure iptables. Prepare by installing the module into your Git repository with `puppet module install`:

```
t@mylaptop ~ $ puppet module install -i ~/puppet/modules puppetlabs
-firewall
Notice: Preparing to install into /home/thomas/puppet/modules ...
Notice: Downloading from https://forgeapi.puppetlabs.com ...
/home/thomas/puppet/modules
└── puppetlabs-firewall (v1.2.0)
```

How to do it...

To configure the firewall module, we need to create a set of rules, which will be applied before all other rules. As a simple example, we'll create the following rules:

- Allow all traffic on the loopback (lo) interface
- Allow all ICMP traffic
- Allow all traffic that is part of an established connection (ESTABLISHED, RELATED)
- Allow all TCP traffic to port 22 (ssh)

We will create a `myfw` (my firewall) class to configure the firewall module. We will then apply the `myfw` class to a node to have iptables configured on that node:

1. Create a class to contain these rules and call it `myfw::pre`:

```
class myfw::pre {
  Firewall {
    require => undef,
  }
  firewall { '0000 Allow all traffic on loopback':
    proto => 'all',
    iniface => 'lo',
    action => 'accept',
  }
  firewall { '0001 Allow all ICMP':
    proto => 'icmp',
    action => 'accept',
  }
  firewall { '0002 Allow all established traffic':
    proto => 'all',
    state => ['RELATED', 'ESTABLISHED'],
    action => 'accept',
  }
  firewall { '0022 Allow all TCP on port 22 (ssh)':
    proto => 'tcp',
    port => '22',
    action => 'accept',
  }
}
```

2. When traffic doesn't match any of the previous rules, we want a final rule that will drop the traffic. Create the class `myfw::post` to contain the default drop rule:

```
class myfw::post {
  firewall { '9999 Drop all other traffic':
    proto  => 'all',
    action => 'drop',
    before => undef,
  }
}
```

3. Create a `myfw` class, which will include `myfw::pre` and `myfw::post` to configure the firewall:

```
class myfw {
  include firewall
  # our rulesets
  include myfw::post
  include myfw::pre
```

```
    # clear all the rules
    resources { "firewall":
      purge => true
    }

    # resource defaults
    Firewall {
      before => Class['myfw::post'],
      require => Class['myfw::pre'],
    }
  }
```

4. Attach the `myfw` class to a node definition; I'll do this to my cookbook node:

   ```
   node cookbook {
     include myfw
   }
   ```

5. Run Puppet on cookbook to see whether the firewall rules have been applied:

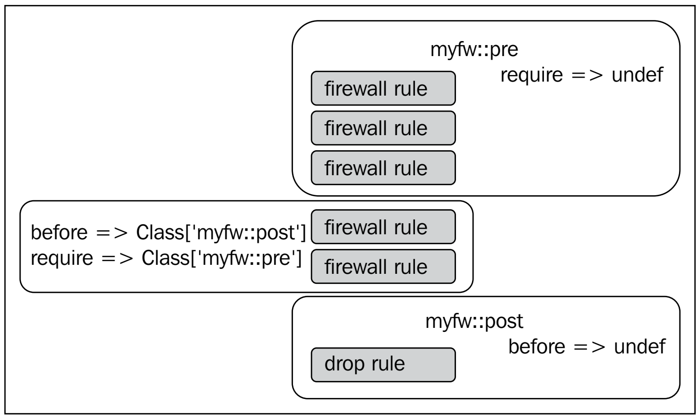
   ```
   [root@cookbook ~]# puppet agent -t
   Info: Retrieving pluginfacts
   Info: Retrieving plugin
   Info: Loading facts
   Info: Caching catalog for cookbook.example.com
   Info: Applying configuration version '1415512948'
   Notice: /Stage[main]/Myfw::Pre/Firewall[000 Allow all traffic on 
   loopback]/ensure: created
   Notice: /File[/etc/sysconfig/iptables]/seluser: seluser changed 
   'unconfined_u' to 'system_u'
   Notice: /Stage[main]/Myfw::Pre/Firewall[0001 Allow all ICMP]/
   ensure: created
   Notice: /Stage[main]/Myfw::Pre/Firewall[0022 Allow all TCP on port 
   22 (ssh)]/ensure: created
   Notice: /Stage[main]/Myfw::Pre/Firewall[0002 Allow all established 
   traffic]/ensure: created
   Notice: /Stage[main]/Myfw::Post/Firewall[9999 Drop all other 
   traffic]/ensure: created
   Notice: /Stage[main]/Myfw/Firewall[9003 
   49bcd611c61bdd18b235cea46ef04fae]/ensure: removed
   Notice: Finished catalog run in 15.65 seconds
   ```

6. Verify the new rules with `iptables-save`:

```
# Generated by iptables-save v1.4.7 on Sun Nov  9 01:18:30 2014
*filter
:INPUT ACCEPT [0:0]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [74:35767]
-A INPUT -i lo -m comment --comment "0000 Allow all traffic on
loopback" -j ACCEPT
-A INPUT -p icmp -m comment --comment "0001 Allow all ICMP" -j
ACCEPT
-A INPUT -m comment --comment "0002 Allow all established traffic"
-m state --state RELATED,ESTABLISHED -j ACCEPT
-A INPUT -p tcp -m multiport --ports 22 -m comment --comment "022
Allow all TCP on port 22 (ssh)" -j ACCEPT
-A INPUT -m comment --comment "9999 Drop all other traffic" -j
DROP
COMMIT
# Completed on Sun Nov  9 01:18:30 2014
```

How it works...

This is a great example of how to use metaparameters to achieve a complex ordering with little effort. Our `myfw` module achieves the following configuration:



All the rules in the `myfw::pre` class are guaranteed to come before any other firewall rules we define. The rules in `myfw::post` are guaranteed to come after any other firewall rules. So, we have the rules in `myfw::pre` first, then any other rules, followed by the rules in `myfw::post`.

Our definition for the `myfw` class sets up this dependency with resource defaults:

```
# resource defaults
Firewall {
  before => Class['myfw::post'],
  require => Class['myfw::pre'],
}
```

These defaults first tell Puppet that any firewall resource should be executed before anything in the `myfw::post` class. Second, they tell Puppet that any firewall resource should require that the resources in `myfw::pre` already be executed.

When we defined the `myfw::pre` class, we removed the require statement in a resource default for Firewall resources. This ensures that the resources within the myfw::pre-class don't require themselves before executing (Puppet will complain that we created a cyclic dependency otherwise):

```
Firewall {
    require => undef,
  }
```

We use the same trick in our `myfw::post` definition. In this case, we only have a single rule in the post class, so we simply remove the `before` requirement:

```
firewall { '9999 Drop all other traffic':
    proto  => 'all',
    action => 'drop',
    before => undef,
  }
```

Finally, we include a rule to purge all the existing iptables rules on the system. We do this to ensure we have a consistent set of rules; only rules defined in Puppet will persist:

```
# clear all the rules
resources { "firewall":
  purge => true
}
```

There's more...

As we hinted, we can now define firewall resources in our manifests and have them applied to the iptables configuration after the initialization rules (`myfw::pre`) but before the final drop (`myfw::post`). For example, to allow http traffic on our cookbook machine, modify the node definition as follows:

```
include myfw
firewall {'0080 Allow HTTP':
  proto  => 'tcp',
  action => 'accept',
  port   => 80,
}
```

Run Puppet on cookbook:

```
[root@cookbook ~]# puppet agent -t
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Loading facts
Info: Caching catalog for cookbook.example.com
Info: Applying configuration version '1415515392'
Notice: /File[/etc/sysconfig/iptables]/seluser: seluser changed
'unconfined_u' to 'system_u'
Notice: /Stage[main]/Main/Node[cookbook]/Firewall[0080 Allow HTTP]/
ensure: created
Notice: Finished catalog run in 2.74 seconds
```

Verify that the new rule has been added after the last myfw::pre rule (port 22, ssh):

```
[root@cookbook ~]# iptables-save
# Generated by iptables-save v1.4.7 on Sun Nov  9 01:46:38 2014
*filter
:INPUT ACCEPT [0:0]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [41:26340]
-A INPUT -i lo -m comment --comment "0000 Allow all traffic on loopback"
-j ACCEPT
-A INPUT -p icmp -m comment --comment "0001 Allow all ICMP" -j ACCEPT
-A INPUT -m comment --comment "0002 Allow all established traffic" -m
state --state RELATED,ESTABLISHED -j ACCEPT
```

```
-A INPUT -p tcp -m multiport --ports 22 -m comment --comment "0022 Allow
all TCP on port 22 (ssh)" -j ACCEPT
-A INPUT -p tcp -m multiport --ports 80 -m comment --comment "0080 Allow
HTTP" -j ACCEPT
-A INPUT -m comment --comment "9999 Drop all other traffic" -j DROP
COMMIT
# Completed on Sun Nov  9 01:46:38 2014
```

[The Puppet Labs Firewall module has a built-in notion of order, all our firewall resource titles begin with a number. This is a requirement. The module attempts to order resources based on the title. You should keep this in mind when naming your firewall resources.]

In the next section, we'll use our firewall module to ensure that two nodes can communicate as required.

Building high-availability services using Heartbeat

High-availability services are those that can survive the failure of an individual machine or network connection. The primary technique for high availability is redundancy, otherwise known as throwing hardware at the problem. Although the eventual failure of an individual server is certain, the simultaneous failure of two servers is unlikely enough that this provides a good level of redundancy for most applications.

One of the simplest ways to build a redundant pair of servers is to have them share an IP address using Heartbeat. Heartbeat is a daemon that runs on both machines and exchanges regular messages—heartbeats—between the two. One server is the primary one, and normally has the resource; in this case, an IP address (known as a virtual IP, or VIP). If the secondary server fails to detect a heartbeat from the primary server, it can take over the address, ensuring continuity of service. In real-world scenarios, you may want more machines involved in the VIP, but for this example, two machines works well enough.

In this recipe, we'll set up two machines in this configuration using Puppet, and I'll explain how to use it to provide a high-availability service.

Getting ready

You'll need two machines, of course, and an extra IP address to use as the VIP. You can usually request this from your ISP, if necessary. In this example, I'll be using machines named `cookbook` and `cookbook2`, with `cookbook` being the primary. We'll add the hosts to the heartbeat configuration.

How to do it...

Follow these steps to build the example:

1. Create the file `modules/heartbeat/manifests/init.pp` with the following contents:

```
# Manage Heartbeat
class heartbeat {
  package { 'heartbeat':
    ensure => installed,
  }

  service { 'heartbeat':
    ensure  => running,
    enable  => true,
    require => Package['heartbeat'],
  }

  file { '/etc/ha.d/authkeys':
    content => "auth 1\n1 sha1 TopSecret",
    mode    => '0600',
    require => Package['heartbeat'],
    notify  => Service['heartbeat'],
  }
  include myfw
  firewall {'0694 Allow UDP ha-cluster':
    proto  => 'udp',
    port   => 694,
    action => 'accept',
  }
}
```

2. Create the file `modules/heartbeat/manifests/vip.pp` with the following contents:

```
# Manage a specific VIP with Heartbeat
class
  heartbeat::vip($node1,$node2,$ip1,$ip2,$vip,$interface='eth0:1')
{
  include heartbeat

  file { '/etc/ha.d/haresources':
    content => "${node1} IPaddr::${vip}/${interface}\n",
    require => Package['heartbeat'],
    notify  => Service['heartbeat'],
  }

  file { '/etc/ha.d/ha.cf':
    content => template('heartbeat/vip.ha.cf.erb'),
    require => Package['heartbeat'],
    notify  => Service['heartbeat'],
  }
}
```

3. Create the file `modules/heartbeat/templates/vip.ha.cf.erb` with the following contents:

```
use_logd yes
udpport 694
autojoin none
ucast eth0 <%= @ip1 %>
ucast eth0 <%= @ip2 %>
keepalive 1
deadtime 10
warntime 5
auto_failback off
node <%= @node1 %>
node <%= @node2 %>
```

4. Modify your `site.pp` file as follows. Replace the `ip1` and `ip2` addresses with the primary IP addresses of your two nodes, `vip` with the virtual IP address you'll be using, and `node1` and `node2` with the hostnames of the two nodes. (Heartbeat uses the fully-qualified domain name of a node to determine whether it's a member of the cluster, so the values for `node1` and `node2` should match what's given by `facter fqdn` on each machine.):

```
node cookbook,cookbook2 {
  class { 'heartbeat::vip':
    ip1   => '192.168.122.132',
    ip2   => '192.168.122.133',
    node1 => 'cookbook.example.com',
    node2 => 'cookbook2.example.com',
    vip   => '192.168.122.200/24',
  }
}
```

5. Run Puppet on each of the two servers:

```
[root@cookbook2 ~]# puppet agent -t
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Loading facts
Info: Caching catalog for cookbook2.example.com
Info: Applying configuration version '1415517914'
Notice: /Stage[main]/Heartbeat/Package[heartbeat]/ensure: created
Notice: /Stage[main]/Myfw::Pre/Firewall[0000 Allow all traffic on
loopback]/ensure: created
Notice: /Stage[main]/Myfw::Pre/Firewall[0001 Allow all ICMP]/
ensure: created
Notice: /File[/etc/sysconfig/iptables]/seluser: seluser changed
'unconfined_u' to 'system_u'
Notice: /Stage[main]/Myfw::Pre/Firewall[0022 Allow all TCP on port
22 (ssh)]/ensure: created
Notice: /Stage[main]/Heartbeat::Vip/File[/etc/ha.d/haresources]/
ensure: defined content as '{md5}fb9f5d9d2b26e3bddf681676d8b2129c'
Info: /Stage[main]/Heartbeat::Vip/File[/etc/ha.d/haresources]:
Scheduling refresh of Service[heartbeat]
Notice: /Stage[main]/Heartbeat::Vip/File[/etc/ha.d/ha.cf]/ensure:
defined content as '{md5}84da22f7ac1a3629f69dcf29ccfd8592'
Info: /Stage[main]/Heartbeat::Vip/File[/etc/ha.d/ha.cf]:
Scheduling refresh of Service[heartbeat]
Notice: /Stage[main]/Heartbeat/Service[heartbeat]/ensure: ensure
changed 'stopped' to 'running'
Info: /Stage[main]/Heartbeat/Service[heartbeat]: Unscheduling
refresh on Service[heartbeat]
Notice: /Stage[main]/Myfw::Pre/Firewall[0002 Allow all established
traffic]/ensure: created
Notice: /Stage[main]/Myfw::Post/Firewall[9999 Drop all other
traffic]/ensure: created
Notice: /Stage[main]/Heartbeat/Firewall[0694 Allow UDP ha
-cluster]/ensure: created
Notice: Finished catalog run in 12.64 seconds
```

6. Verify that the VIP is running on one of the nodes (it should be on cookbook at this point; note that you will need to use the ip command, ifconfig will not show the address):

```
[root@cookbook ~]# ip addr show dev eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_
fast state UP qlen 1000
    link/ether 52:54:00:c9:d5:63 brd ff:ff:ff:ff:ff:ff
    inet 192.168.122.132/24 brd 192.168.122.255 scope global eth0
    inet 192.168.122.200/24 brd 192.168.122.255 scope global
secondary eth0:1
    inet6 fe80::5054:ff:fec9:d563/64 scope link
       valid_lft forever preferred_lft forever
```

7. As we can see, cookbook has the eth0:1 interface active. If you stop heartbeat on cookbook, cookbook2 will create eth0:1 and take over:

```
[root@cookbook2 ~]# ip a show dev eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_
fast state UP qlen 1000
    link/ether 52:54:00:ee:9c:fa brd ff:ff:ff:ff:ff:ff
    inet 192.168.122.133/24 brd 192.168.122.255 scope global eth0
    inet 192.168.122.200/24 brd 192.168.122.255 scope global
secondary eth0:1
    inet6 fe80::5054:ff:feee:9cfa/64 scope link
       valid_lft forever preferred_lft forever
```

How it works...

We need to install Heartbeat first of all, using the heartbeat class:

```
# Manage Heartbeat
class heartbeat {
  package { 'heartbeat':
    ensure => installed,
  }
  ...
}
```

Next, we use the heartbeat::vip class to manage a specific virtual IP:

```
# Manage a specific VIP with Heartbeat
class
```

```
    heartbeat::vip($node1,$node2,$ip1,$ip2,$vip,$interface='eth0:1') {
    include heartbeat
```

As you can see, the class includes an `interface` parameter; by default, the VIP will be configured on `eth0:1`, but if you need to use a different interface, you can pass it in using this parameter.

Each pair of servers that we configure with a virtual IP will use the `heartbeat::vip` class with the same parameters. These will be used to build the `haresources` file:

```
file { '/etc/ha.d/haresources':
  content => "${node1} IPaddr::${vip}/${interface}\n",
  notify  => Service['heartbeat'],
  require => Package['heartbeat'],
}
```

This tells Heartbeat about the resource it should manage (that's a Heartbeat resource, such as an IP address or a service, not a Puppet resource). The resulting `haresources` file might look as follows:

```
cookbook.example.com IPaddr::192.168.122.200/24/eth0:1
```

The file is interpreted by Heartbeat as follows:

- `cookbook.example.com`: This is the name of the primary node, which should be the default owner of the resource
- `IPaddr`: This is the type of resource to manage; in this case, an IP address
- `192.168.122.200/24`: This is the value for the IP address
- `eth0:1`: This is the virtual interface to configure with the managed IP address

For more information on how heartbeat is configured, please visit the high-availability site at `http://linux-ha.org/wiki/Heartbeat`.

We will also build the `ha.cf` file that tells Heartbeat how to communicate between cluster nodes:

```
file { '/etc/ha.d/ha.cf':
  content => template('heartbeat/vip.ha.cf.erb'),
  notify  => Service['heartbeat'],
  require => Package['heartbeat'],
}
```

To do this, we use the template file:

```
use_logd yes
udpport 694
autojoin none
```

```
ucast eth0 <%= @ip1 %>
ucast eth0 <%= @ip2 %>
keepalive 1
deadtime 10
warntime 5
auto_failback off
node <%= @node1 %>
node <%= @node2 %>
```

The interesting values here are the IP addresses of the two nodes (`ip1` and `ip2`), and the names of the two nodes (`node1` and `node2`).

Finally, we create an instance of `heartbeat::vip` on both machines and pass it an identical set of parameters as follows:

```
class { 'heartbeat::vip':
  ip1   => '192.168.122.132',
  ip2   => '192.168.122.133',
  node1 => 'cookbook.example.com',
  node2 => 'cookbook2.example.com',
  vip   => '192.168.122.200/24',
}
```

There's more...

With Heartbeat set up as described in the example, the virtual IP address will be configured on `cookbook` by default. If something happens to interfere with this (for example, if you halt or reboot `cookbook`, or stop the `heartbeat` service, or the machine loses network connectivity), `cookbook2` will immediately take over the virtual IP.

The `auto_failback` setting in `ha.cf` governs what happens next. If `auto_failback` is set to `on`, when `cookbook` becomes available once more, it will automatically take over the IP address. Without `auto_failback`, the IP will stay where it is until you manually fail it again (by stopping `heartbeart` on `cookbook2`, for example).

One common use for a Heartbeat-managed virtual IP is to provide a highly available website or service. To do this, you need to set the DNS name for the service (for example, `cat-pictures.com`) to point to the virtual IP. Requests for the service will be routed to whichever of the two servers currently has the virtual IP. If this server should go down, requests will go to the other, with no visible interruption in service to users.

Heartbeat works great for the previous example but is not in widespread use in this form. Heartbeat only works in two node clusters; for n-node clusters, the newer pacemaker project should be used. More information on Heartbeat, pacemaker, corosync, and other clustering packages can be found at `http://www.linux-ha.org/wiki/Main_Page`.

Managing cluster configuration is one area where exported resources are useful. Each node in a cluster would export information about itself, which could then be collected by the other members of the cluster. Using the puppetlabs-concat module, you can build up a configuration file using exported concat fragments from all the nodes in the cluster.

Remember to look at the Forge before starting your own module. If nothing else, you'll get some ideas that you can use in your own module. Corosync can be managed with the Puppet labs module at `https://forge.puppetlabs.com/puppetlabs/corosync`.

Managing NFS servers and file shares

NFS (**Network File System**) is a protocol to mount a shared directory from a remote server. For example, a pool of web servers might all mount the same NFS share to serve static assets such as images and stylesheets. Although NFS is generally slower and less secure than local storage or a clustered filesystem, the ease with which it can be used makes it a common choice in the datacenter. We'll use our `myfw` module from before to ensure the local firewall permits `nfs` communication. We'll also use the Puppet labs-concat module to edit the list of exported filesystems on our `nfs` server.

How to do it...

In this example, we'll configure an `nfs` server to share (export) some filesystem via NFS.

1. Create an `nfs` module with the following `nfs::exports` class, which defines a concat resource:

```
class nfs::exports {
  exec {'nfs::exportfs':
    command     => 'exportfs -a',
    refreshonly => true,
    path        => '/usr/bin:/bin:/sbin:/usr/sbin',
  }
  concat {'/etc/exports':
    notify => Exec['nfs::exportfs'],
  }
}
```

2. Create the `nfs::export` defined type, we'll use this definition for any `nfs` exports we create:

```
define nfs::export (
  $where = $title,
  $who = '*',
  $options = 'async,ro',
  $mount_options = 'defaults',
  $tag     = 'nfs'
) {
  # make sure the directory exists
  # export the entry locally, then export a resource to be picked
up later.
  file {"$where":
    ensure => 'directory',
  }
  include nfs::exports
  concat::fragment { "nfs::export::$where":
    content => "${where} ${who}(${options})\n",
    target  => '/etc/exports'
  }
  @@mount { "nfs::export::${where}::${::ipaddress}":
    name    => "$where",
    ensure  => 'mounted',
    fstype  => 'nfs',
    options => "$mount_options",
    device  => "${::ipaddress}:${where}",
    tag     => "$tag",
  }
}
```

3. Now create the `nfs::server` class, which will include the OS-specific configuration for the server:

```
class nfs::server {
  # ensure nfs server is running
  # firewall should allow nfs communication
  include nfs::exports
  case $::osfamily {
    'RedHat': { include nfs::server::redhat }
    'Debian': { include nfs::server::debian }
  }
  include myfw
  firewall {'2049 NFS TCP communication':
    proto  => 'tcp',
    port   => '2049',
    action => 'accept',
  }
  firewall {'2049 UDP NFS communication':
    proto  => 'udp',
    port   => '2049',
    action => 'accept',
  }
  firewall {'0111 TCP PORTMAP':
    proto  => 'tcp',
    port   => '111',
    action => 'accept',
  }
  firewall {'0111 UDP PORTMAP':
    proto  => 'udp',
    port   => '111',
    action => 'accept',
  }
  firewall {'4000 TCP STAT':
    proto  => 'tcp',
    port   => '4000-4010',
    action => 'accept',
  }
  firewall {'4000 UDP STAT':
    proto  => 'udp',
    port   => '4000-4010',
    action => 'accept',
  }
}
```

4. Next, create the `nfs::server::redhat` class:

```
class nfs::server::redhat {
  package {'nfs-utils':
    ensure => 'installed',
  }
  service {'nfs':
    ensure => 'running',
    enable => true
  }
  file {'/etc/sysconfig/nfs':
    source => 'puppet:///modules/nfs/nfs',
    mode   => 0644,
    notify => Service['nfs'],
  }
}
```

5. Create the `/etc/sysconfig/nfs` support file for RedHat systems in the files directory of our `nfs` repo (`modules/nfs/files/nfs`):

```
STATD_PORT=4000
STATD_OUTGOING_PORT=4001
RQUOTAD_PORT=4002
```

```
LOCKD_TCPPORT=4003
LOCKD_UDPPORT=4003
MOUNTD_PORT=4004
```

6. Now create the support class for Debian systems, `nfs::server::debian`:

```
class nfs::server::debian {
  # install the package
  package {'nfs':
    name   => 'nfs-kernel-server',
    ensure => 'installed',
  }
  # config
  file {'/etc/default/nfs-common':
    source => 'puppet:///modules/nfs/nfs-common',
    mode   => 0644,
    notify => Service['nfs-common']
  }
  # services
  service {'nfs-common':
    ensure => 'running',
    enable => true,
  }
  service {'nfs':
    name   => 'nfs-kernel-server',
    ensure => 'running',
    enable => true,
    require => Package['nfs-kernel-server']
  }
}
```

7. Create the nfs-common configuration for Debian (which will be placed in `modules/nfs/files/nfs-common`):

```
STATDOPTS="--port 4000 --outgoing-port 4001"
```

8. Apply the `nfs::server` class to a node and then create an export on that node:

```
node debian {
  include nfs::server
  nfs::export {'/srv/home':
    tag => "srv_home" }
}
```

9. Create a collector for the exported resource created by the `nfs::server` class in the preceding code snippet:

```
node cookbook {
  Mount <<| tag == "srv_home" |>> {
    name    => '/mnt',
  }
}
```

10. Finally, run Puppet on the node Debian to create the exported resource. Then, run Puppet on the cookbook node to mount that resource:

```
root@debian:~# puppet agent -t
Info: Caching catalog for debian.example.com
Info: Applying configuration version '1415602532'
Notice: Finished catalog run in 0.78 seconds
[root@cookbook ~]# puppet agent -t
Info: Caching catalog for cookbook.example.com
Info: Applying configuration version '1415603580'
Notice: /Stage[main]/Main/Node[cookbook]/Mount[nfs::export::/srv/
home::192.168.122.148]/ensure: ensure changed 'ghost' to 'mounted'
Info: Computing checksum on file /etc/fstab
Info: /Stage[main]/Main/Node[cookbook]/Mount[nfs::export::/srv/
home::192.168.122.148]: Scheduling refresh of Mount[nfs::export::/
srv/home::192.168.122.148]
Info: Mount[nfs::export::/srv/home::192.168.122.148]
(provider=parsed): Remounting
Notice: /Stage[main]/Main/Node[cookbook]/Mount[nfs::export::/srv/
home::192.168.122.148]: Triggered 'refresh' from 1 events
Info: /Stage[main]/Main/Node[cookbook]/Mount[nfs::export::/srv/
home::192.168.122.148]: Scheduling refresh of Mount[nfs::export::/
srv/home::192.168.122.148]
Notice: Finished catalog run in 0.34 seconds
```

11. Verify the mount with `mount`:

```
[root@cookbook ~]# mount -t nfs
192.168.122.148:/srv/home on /mnt type nfs (rw)
```

How it works...

The `nfs::exports` class defines an exec, which runs `'exportfs -a'`, to export all filesystems defined in `/etc/exports`. Next, we define a concat resource to contain `concat::fragments`, which we will define next in our `nfs::export` class. Concat resources specify the file that the fragments are to be placed into; `/etc/exports` in this case. Our `concat` resource has a notify for the previous exec. This has the effect that whenever `/etc/exports` is updated, we run `'exportfs -a'` again to export the new entries:

```
class nfs::exports {
  exec {'nfs::exportfs':
    command     => 'exportfs -a',
    refreshonly => true,
    path        => '/usr/bin:/bin:/sbin:/usr/sbin',
  }
  concat {'/etc/exports':
    notify => Exec['nfs::exportfs'],
  }
}
```

We then created an `nfs::export` defined type, which does all the work. The defined type adds an entry to `/etc/exports` via a `concat::fragment` resource:

```
define nfs::export (
  $where = $title,
  $who = '*',
  $options = 'async,ro',
  $mount_options = 'defaults',
  $tag     = 'nfs'
) {
  # make sure the directory exists
  # export the entry locally, then export a resource to be picked up
later.
  file {"$where":
    ensure => 'directory',
  }
  include nfs::exports
  concat::fragment { "nfs::export::$where":
    content => "${where} ${who}(${options})\n",
    target  => '/etc/exports'
  }
```

In the definition, we use the attribute `$where` to define what filesystem we are exporting. We use `$who` to specify who can mount the filesystem. The attribute `$options` contains the exporting options such as **rw** (**read-write**), **ro** (**read-only**). Next, we have the options that will be placed in `/etc/fstab` on the client machine, the mount options, stored in `$mount_options`. The `nfs::exports` class is included here so that `concat::fragment` has a concat target defined.

Next, the exported mount resource is created; this is done on the server, so the `${::ipaddress}` variable holds the IP address of the server. We use this to define the device for the mount. The device is the IP address of the server, a colon, and then the filesystem being exported. In this example, it is `'192.168.122.148:/srv/home'`:

```
@@mount { "nfs::export::${where}::${::ipaddress}":
    name     => "$where",
    ensure   => 'mounted',
    fstype   => 'nfs',
    options  => "$mount_options",
    device   => "${::ipaddress}:${where}",
    tag      => "$tag",
  }
```

We reuse our `myfw` module and include it in the `nfs::server` class. This class illustrates one of the things to consider when writing your modules. Not all Linux distributions are created equal. Debian and RedHat deal with NFS server configuration quite differently. The `nfs::server` module deals with this by including OS-specific subclasses:

```
class nfs::server {
  # ensure nfs server is running
  # firewall should allow nfs communication
  include nfs::exports
  case $::osfamily {
    'RedHat': { include nfs::server::redhat }
    'Debian': { include nfs::server::debian }
  }
  include myfw
  firewall {'2049 NFS TCP communication':
    proto  => 'tcp',
    port   => '2049',
    action => 'accept',
  }
  firewall {'2049 UDP NFS communication':
    proto  => 'udp',
    port   => '2049',
    action => 'accept',
  }
  firewall {'0111 TCP PORTMAP':
    proto  => 'tcp',
    port   => '111',
    action => 'accept',
  }
```

```
      firewall {'0111 UDP PORTMAP':
        proto  => 'udp',
        port   => '111',
        action => 'accept',
      }
      firewall {'4000 TCP STAT':
        proto  => 'tcp',
        port   => '4000-4010',
        action => 'accept',
      }
      firewall {'4000 UDP STAT':
        proto  => 'udp',
        port   => '4000-4010',
        action => 'accept',
      }
    }
```

The `nfs::server` module opens several firewall ports for NFS communication. NFS traffic is always carried over port 2049 but ancillary systems, such as locking, quota, and file status daemons, use ephemeral ports chosen by the portmapper, by default. The portmapper itself uses port 111. So our module needs to allow 2049, 111, and a few other ports. We attempt to configure the ancillary services to use ports 4000 through 4010.

In the `nfs::server::redhat` class, we modify `/etc/sysconfig/nfs` to use the ports specified. Also, we install the nfs-utils package and start the nfs service:

```
    class nfs::server::redhat {
      package {'nfs-utils':
        ensure => 'installed',
      }
      service {'nfs':
        ensure => 'running',
        enable => true
      }
      file {'/etc/sysconfig/nfs':
        source => 'puppet:///modules/nfs/nfs',
        mode   => 0644,
        notify => Service['nfs'],
      }
    }
```

We do the same for Debian-based systems in the `nfs::server::debian` class. The packages and services have different names but overall the process is similar:

```
    class nfs::server::debian {
      # install the package
      package {'nfs':
        name   => 'nfs-kernel-server',
        ensure => 'installed',
      }
      # config
      file {'/etc/default/nfs-common':
        source => 'puppet:///modules/nfs/nfs-common',
        mode   => 0644,
        notify => Service['nfs-common']
      }
      # services
      service {'nfs-common':
        ensure => 'running',
        enable => true,
      }
      service {'nfs':
        name   => 'nfs-kernel-server',
        ensure => 'running',
        enable => true,
      }
    }
```

With everything in place, we include the server class to configure the NFS server and then define an export:

```
    include nfs::server
    nfs::export {'/srv/home':
      tag => "srv_home" }
```

What's important here is that we defined the `tag` attribute, which will be used in the exported resource we collect in the following code snippet:

```
  Mount <<| tag == "srv_home" |>> {
    name   => '/mnt',
  }
```

We use the spaceship syntax (`<<|  |>>`) to collect all the exported mount resources that have the tag we defined earlier (`srv_home`). We then use a syntax called "override on collect" to modify the name attribute of the mount to specify where to mount the filesystem.

Using this design pattern with exported resources, we can change the server exporting the filesystem and have any nodes that mount the resource updated automatically. We can have many different nodes collecting the exported mount resource.

Using HAProxy to load-balance multiple web servers

Load balancers are used to spread a load among a number of servers. Hardware load balancers are still somewhat expensive, whereas software balancers can achieve most of the benefits of a hardware solution.

HAProxy is the software load balancer of choice for most people: fast, powerful, and highly configurable.

How to do it...

In this recipe, I'll show you how to build an HAProxy server to load-balance web requests across web servers. We'll use exported resources to build the haproxy configuration file just like we did for the NFS example.

1. Create the file modules/haproxy/manifests/master.pp with the following contents:

```
class haproxy::master ($app = 'myapp') {
  # The HAProxy master server
  # will collect haproxy::slave resources and add to its balancer
  package { 'haproxy': ensure => installed }
  service { 'haproxy':
    ensure  => running,
    enable  => true,
    require => Package['haproxy'],
  }

  include haproxy::config

  concat::fragment { 'haproxy.cfg header':
    target  => 'haproxy.cfg',
    source  => 'puppet:///modules/haproxy/haproxy.cfg',
    order   => '001',
    require => Package['haproxy'],
    notify  => Service['haproxy'],
  }

  # pull in the exported entries
  Concat::Fragment <<| tag == "$app" |>> {
    target => 'haproxy.cfg',
    notify => Service['haproxy'],
  }
}
```

2. Create the file `modules/haproxy/files/haproxy.cfg` with the following contents:

```
global
        daemon
        user haproxy
        group haproxy
        pidfile /var/run/haproxy.pid

defaults
        log     global
        stats   enable
        mode    http
        option  httplog
        option  dontlognull
        option  dontlog-normal
        retries 3
        option  redispatch
        timeout connect 4000
        timeout client 60000
        timeout server 30000

listen  stats :8080
        mode http
        stats uri /
        stats auth haproxy:topsecret

listen  myapp 0.0.0.0:80
        balance leastconn
```

3. Modify your `manifests/nodes.pp` file as follows:

```
node 'cookbook' {
  include haproxy
}
```

4. Create the slave server configuration in the `haproxy::slave` class:

```
class haproxy::slave ($app = "myapp", $localport = 8000) {
  # haproxy slave, export haproxy.cfg fragment
  # configure simple web server on different port
  @@concat::fragment { "haproxy.cfg $::fqdn":
    content => "\t\tserver ${::hostname}
${::ipaddress}:${localport}   check maxconn 100\n",
    order   => '0010',
    tag     => "$app",
  }
  include myfw
  firewall {"${localport} Allow HTTP to haproxy::slave":
    proto   => 'tcp',
    port    => $localport,
    action => 'accept',
  }

  class {'apache': }
  apache::vhost { 'haproxy.example.com':
    port           => '8000',
    docroot => '/var/www/haproxy',
  }
  file {'/var/www/haproxy':
    ensure  => 'directory',
    mode    => 0755,
    require => Class['apache'],
  }
  file {'/var/www/haproxy/index.html':
    mode    => '0644',
    content => "<html><body><h1>${::fqdn} haproxy::slave\n</
body></html>\n",
    require => File['/var/www/haproxy'],
  }
}
```

5. Create the `concat` container resource in the `haproxy::config` class as follows:

```
class haproxy::config {
  concat {'haproxy.cfg':
    path  => '/etc/haproxy/haproxy.cfg',
    order => 'numeric',
    mode  => '0644',
  }
}
```

6. Modify `site.pp` to define the master and slave nodes:

```
node master {
  class {'haproxy::master':
    app => 'cookbook'
  }
}
node slave1,slave2 {
  class {'haproxy::slave':
    app => 'cookbook'
  }
}
```

7. Run Puppet on each of the slave servers:

```
root@slave1:~# puppet agent -t
Info: Caching catalog for slave1
Info: Applying configuration version '1415646194'
Notice: /Stage[main]/Haproxy::Slave/Apache::Vhost[haproxy.example.
com]/File[25-haproxy.example.com.conf]/ensure: created
Info: /Stage[main]/Haproxy::Slave/Apache::Vhost[haproxy.example.
com]/File[25-haproxy.example.com.conf]: Scheduling refresh of
Service[httpd]
Notice: /Stage[main]/Haproxy::Slave/Apache::Vhost[haproxy.example.
com]/File[25-haproxy.example.com.conf symlink]/ensure: created
Info: /Stage[main]/Haproxy::Slave/Apache::Vhost[haproxy.example.
com]/File[25-haproxy.example.com.conf symlink]: Scheduling refresh
of Service[httpd]
Notice: /Stage[main]/Apache::Service/Service[httpd]/ensure: ensure
changed 'stopped' to 'running'
Info: /Stage[main]/Apache::Service/Service[httpd]: Unscheduling
refresh on Service[httpd]
Notice: Finished catalog run in 1.71 seconds
```

8. Run Puppet on the master node to configure and run `haproxy`:

```
[root@master ~]# puppet agent -t
Info: Caching catalog for master.example.com
Info: Applying configuration version '1415647075'
Notice: /Stage[main]/Haproxy::Master/Package[haproxy]/ensure:
created
Notice: /Stage[main]/Myfw::Pre/Firewall[0000 Allow all traffic on
loopback]/ensure: created
Notice: /Stage[main]/Myfw::Pre/Firewall[0001 Allow all ICMP]/
ensure: created
Notice: /Stage[main]/Haproxy::Master/Firewall[8080 haproxy
statistics]/ensure: created
Notice: /File[/etc/sysconfig/iptables]/seluser: seluser changed
'unconfined_u' to 'system_u'
Notice: /Stage[main]/Myfw::Pre/Firewall[0022 Allow all TCP on port
22 (ssh)]/ensure: created
Notice: /Stage[main]/Haproxy::Master/Firewall[0080 http haproxy]/
ensure: created
```

```
Notice: /Stage[main]/Myfw::Pre/Firewall[0002 Allow all established
traffic]/ensure: created

Notice: /Stage[main]/Myfw::Post/Firewall[9999 Drop all other
traffic]/ensure: created

Notice: /Stage[main]/Haproxy::Config/Concat[haproxy.cfg]/
File[haproxy.cfg]/content:

...

+listen  myapp 0.0.0.0:80

+        balance leastconn

+    server slave1 192.168.122.148:8000   check maxconn 100

+    server slave2 192.168.122.133:8000   check maxconn 100


Info: Computing checksum on file /etc/haproxy/haproxy.cfg

Info: /Stage[main]/Haproxy::Config/Concat[haproxy.cfg]/
File[haproxy.cfg]: Filebucketed /etc/haproxy/haproxy.cfg to puppet
with sum 1f337186b0e1ba5ee82760cb437fb810

Notice: /Stage[main]/Haproxy::Config/Concat[haproxy.cfg]/
File[haproxy.cfg]/content: content changed '{md5}1f337186b0e1ba5ee
82760cb437fb810' to '{md5}b070f076e1e691e053d6853f7d966394'

Notice: /Stage[main]/Haproxy::Master/Service[haproxy]/ensure:
ensure changed 'stopped' to 'running'

Info: /Stage[main]/Haproxy::Master/Service[haproxy]: Unscheduling
refresh on Service[haproxy]

Notice: Finished catalog run in 33.48 seconds
```

9. Check the HAProxy stats interface on master port `8080` in your web browser (`http://master.example.com:8080`) to make sure everything is okay (The username and password are in `haproxy.cfg`, `haproxy`, and `topsecret`). Try going to the proxied service as well. Notice that the page changes on each reload as the service is redirected from slave1 to slave2 (`http://master.example.com`).

How it works...

We built a complex configuration from various components of the previous sections. This type of deployment becomes easier the more you do it. At a top level, we configured the master to collect exported resources from slaves. The slaves exported their configuration information to allow haproxy to use them in the load balancer. As slaves are added to the system, they can export their resources and be added to the balancer automatically.

We used our `myfw` module to configure the firewall on the slaves and the master to allow communication.

We used the Forge Apache module to configure the listening web server on the slaves. We were able to generate a fully functioning website with five lines of code (10 more to place `index.html` on the website).

There are several things going on here. We have the firewall configuration and the Apache configuration in addition to the `haproxy` configuration. We'll focus on how the exported resources and the `haproxy` configuration fit together.

In the `haproxy::config` class, we created the concat container for the `haproxy` configuration:

```
class haproxy::config {
  concat {'haproxy.cfg':
    path  => '/etc/haproxy/haproxy.cfg',
    order => 'numeric',
    mode  => 0644,
  }
}
```

We reference this in `haproxy::slave`:

```
class haproxy::slave ($app = "myapp", $localport = 8000) {
  # haproxy slave, export haproxy.cfg fragment
  # configure simple web server on different port
  @@concat::fragment { "haproxy.cfg $::fqdn":
    content => "\t\tserver ${::hostname} ${::ipaddress}:${localport}
check maxconn 100\n",
    order   => '0010',
    tag     => "$app",
  }
```

We are doing a little trick here with concat; we don't define the target in the exported resource. If we did, the slaves would try and create a `/etc/haproxy/haproxy.cfg` file, but the slaves do not have `haproxy` installed so we would get catalog failures. What we do is modify the resource when we collect it in `haproxy::master`:

```
# pull in the exported entries
  Concat::Fragment <<| tag == "$app" |>> {
    target => 'haproxy.cfg',
    notify => Service['haproxy'],
  }
```

In addition to adding the target when we collect the resource, we also add a notify so that the `haproxy` service is restarted when we add a new host to the configuration. Another important point here is that we set the order attribute of the slave configurations to 0010, when we define the header for the `haproxy.cfg` file; we use an order value of 0001 to ensure that the header is placed at the beginning of the file:

```
  concat::fragment { 'haproxy.cfg header':
    target  => 'haproxy.cfg',
    source  => 'puppet:///modules/haproxy/haproxy.cfg',
    order   => '001',
    require => Package['haproxy'],
    notify  => Service['haproxy'],
  }
```

The rest of the `haproxy::master` class is concerned with configuring the firewall as we did in previous examples.

There's more...

HAProxy has a vast range of configuration parameters, which you can explore; see the HAProxy website at `http://haproxy.1wt.eu/#docs`.

Although it's most often used as a web server, HAProxy can proxy a lot more than just HTTP. It can handle any kind of TCP traffic, so you can use it to balance the load of MySQL servers, SMTP, video servers, or anything you like.

You can use the design we showed to attack many problems of coordination of services between multiple servers. This type of interaction is very common; you can apply it to many configurations for load balancing or distributed systems. You can use the same workflow described previously to have nodes export firewall resources (`@@firewall`) to permit their own access.

Managing Docker with Puppet

Docker is a platform for rapid deployment of containers. Containers are like a lightweight virtual machine that might only run a single process. The containers in Docker are called docks and are configured with files called Dockerfiles. Puppet can be used to configure a node to not only run Docker but also configure and start several docks. You can then use Puppet to ensure that your docks are running and are consistently configured.

Getting ready

Download and install the Puppet Docker module from the Forge (`https://forge.puppetlabs.com/garethr/docker`):

```
t@mylaptop ~ $ cd puppet
t@mylaptop ~/puppet $ puppet module install -i modules garethr-docker
Notice: Preparing to install into /home/thomas/puppet/modules ...
Notice: Downloading from https://forgeapi.puppetlabs.com ...
```

```
Notice: Installing -- do not interrupt ...
/home/thomas/puppet/modules
└─┬ garethr-docker (v3.3.0)
  ├── puppetlabs-apt (v1.7.0)
  ├── puppetlabs-stdlib (v4.3.2)
  └── stahnma-epel (v1.0.2)
```

Add these modules to your Puppet repository. The `stahnma-epel` module is required for Enterprise Linux-based distributions; it contains the Extra Packages for Enterprise Linux YUM repository.

How to do it...

Perform the following steps to manage Docker with Puppet:

1. To install Docker on a node, we just need to include the `docker` class. We'll do more than install Docker; we'll also download an image and start an application on our test node. In this example, we'll create a new machine called `shipyard`. Add the following node definition to `site.pp`:

   ```
      node shipyard {
     class {'docker': }
     docker::image {'phusion/baseimage': }
     docker::run {'cookbook':
       image   => 'phusion/baseimage',
       expose  => '8080',
       ports   => '8080',
       command => 'nc -k -l 8080',
     }
   }
   ```

2. Run Puppet on your shipyard node to install Docker. This will also download the `phusion/baseimage docker` image:

   ```
   [root@shipyard ~]# puppet agent -t
   Info: Retrieving pluginfacts
   Info: Retrieving plugin
   Info: Loading facts
   Info: Caching catalog for shipyard
   Info: Applying configuration version '1421049252'
   ```

```
Notice: /Stage[main]/Epel/File[/etc/pki/rpm-gpg/RPM
-GPG-KEY-EPEL-6]/ensure: defined content as '{md5}
d865e6b948a74cb03bc3401c0b01b785'

Notice: /Stage[main]/Epel/Epel::Rpm_gpg_key[EPEL-6]/Exec[import
-EPEL-6]/returns: executed successfully

...

Notice: /Stage[main]/Docker::Install/Package[docker]/
ensure: created

...

Notice: /Stage[main]/Main/Node[shipyard]/Docker::Run[cookbook]/
File[/etc/init.d/docker-cookbook]/ensure: created

Info: /Stage[main]/Main/Node[shipyard]/Docker::Run[cookbook]/
File[/etc/init.d/docker-cookbook]: Scheduling refresh of
Service[docker-cookbook]

Notice: /Stage[main]/Main/Node[shipyard]/Docker::Run[cookbook]/
Service[docker-cookbook]: Triggered 'refresh' from 1 events
```

3. Verify that your container is running on shipyard using `docker ps`:

```
[root@shipyard ~]# docker ps

CONTAINER ID        IMAGE                        COMMAND
CREATED              STATUS               PORTS
NAMES

f6f5b799a598        phusion/baseimage:0.9.15   "/bin/nc -l 8080"
About a minute ago   Up About a minute   0.0.0.0:49157->8080/tcp
suspicious_hawking
```

4. Verify that the dock is running netcat on port 8080 by connecting to the port listed previously (`49157`):

```
[root@shipyard ~]# nc -v localhost 49157

Connection to localhost 49157 port [tcp/*] succeeded!
```

How it works...

We began by installing the docker module from the Forge. This module installs the `docker-io` package on our node, along with any required dependencies.

We then defined a `docker::image` resource. This instructs Puppet to ensure that the named image is downloaded and available to docker. On our first run, Puppet will make docker download the image. We used `phusion/baseimage` as our example because it is quite small, well-known, and includes the netcat daemon we used in the example. More information on `baseimage` can be found at `http://phusion.github.io/baseimage-docker/`.

We then went on to define a `docker::run` resource. This example isn't terribly useful; it simply starts netcat in listen mode on port 8080. We need to expose that port to our machine, so we define the expose attribute of our `docker::run` resource. There are many other options available for the `docker::run` resource. Refer to the source code for more details.

We then used docker ps to list the running docks on our shipyard machine. We parsed out the listening port on our local machine and verified that netcat was listening.

There's more...

Docker is a great tool for rapid deployment and development. You can spin as many docks as you need on even the most modest hardware. One great use for docker is having docks act as test nodes for your modules. You can create a docker image, which includes Puppet, and then have Puppet run within the dock. For more information on docker, visit `http://www.docker.com/`.

9
External Tools and the Puppet Ecosystem

"By all means leave the road when you wish. That is precisely the use of a road: to reach individually chosen points of departure."

– Robert Bringhurst, The Elements of Typographic Style

In this chapter, we will cover the following recipes:

- Creating custom facts
- Adding external facts
- Setting facts as environment variables
- Generating manifests with the Puppet resource command
- Generating manifests with other tools
- Using an external node classifier
- Creating your own resource types
- Creating your own providers
- Creating custom functions
- Testing your Puppet manifests with rspec-puppet
- Using librarian-puppet
- Using r10k

Introduction

Puppet is a useful tool by itself, but you can get much greater benefits by using Puppet in combination with other tools and frameworks. We'll look at some ways of getting data into Puppet, including custom Facter facts, external facts, and tools to generate Puppet manifests automatically from the existing configuration.

You'll also learn how to extend Puppet by creating your own custom functions, resource types, and providers; how to use an external node classifier script to integrate Puppet with other parts of your infrastructure; and how to test your code with rspec-puppet.

Creating custom facts

While Facter's built-in facts are useful, it's actually quite easy to add your own facts. For example, if you have machines in different data centers or hosting providers, you could add a custom fact for this so that Puppet can determine whether any local settings need to be applied (for example, local DNS servers or network routes).

How to do it...

Here's an example of a simple custom fact:

1. Create the directory `modules/facts/lib/facter` and then create the file `modules/facts/lib/facter/hello.rb` with the following contents:

   ```
   Facter.add(:hello) do
     setcode do
       "Hello, world"
     end
   end
   ```

2. Modify your `site.pp` file as follows:

   ```
   node 'cookbook' {
     notify { $::hello: }
   }
   ```

3. Run Puppet:

   ```
   [root@cookbook ~]# puppet agent -t
   Notice: /File[/var/lib/puppet/lib/facter/hello.rb]/ensure: defined
   content as '{md5}f66d5e290459388c5ffb3694dd22388b'
   Info: Loading facts
   Info: Caching catalog for cookbook.example.com
   Info: Applying configuration version '1416205745'
   ```

```
Notice: Hello, world
Notice: /Stage[main]/Main/Node[cookbook]/Notify[Hello, world]/
message: defined 'message' as 'Hello, world'
Notice: Finished catalog run in 0.53 seconds
```

How it works...

Facter facts are defined in Ruby files that are distributed with facter. Puppet can add additional facts to facter by creating files within the `lib/facter` subdirectory of a module. These files are then transferred to client nodes as we saw earlier with the `puppetlabs-stdlib` module. To have the command-line facter use these `puppet` facts, append the `-p` option to facter as shown in the following command line:

```
[root@cookbook ~]# facter hello


[root@cookbook ~]# facter -p hello
Hello, world
```

If you are using an older version of Puppet (older than 3.0), you will need to enable `pluginsync` in your `puppet.conf` file as shown in the following command line:

```
[main]
pluginsync = true
```

Facts can contain any Ruby code, and the last value evaluated inside the `setcode do ... end` block will be the value returned by the fact. For example, you could make a more useful fact that returns the number of users currently logged in to the system:

```
Facter.add(:users) do
  setcode do
    %x{/usr/bin/who |wc -l}.chomp
  end
end
```

To reference the fact in your manifests, just use its name like a built-in fact:

```
notify { "${::users} users logged in": }
Notice:  2 users logged in
```

You can add custom facts to any Puppet module. When creating facts that will be used by multiple modules, it may make sense to place them in a facts module. In most cases, the custom fact is related to a specific module and should be placed in that module.

There's more...

The name of the Ruby file that holds the fact definition is irrelevant. You can name this file whatever you wish; the name of the fact comes from the `Facter.add()` function call. You may also call this function several times within a single Ruby file to define multiple facts as necessary. For instance, you could `grep` the `/proc/meminfo` file and return several facts based on memory information as shown in the `meminfo.rb` file in the following code snippet:

```
File.open('/proc/meminfo') do |f|
  f.each_line { |line|
  if (line[/^Active:/])
    Facter.add(:memory_active) do
      setcode do line.split(':')[1].to_i
      end
    end
  end
  if (line[/^Inactive:/])
    Facter.add(:memory_inactive) do
      setcode do line.split(':')[1].to_i
      end
    end
  end
  }
end
```

After synchronizing this file to a node, the `memory_active` and `memory_inactive` facts would be available as follows:

```
[root@cookbook ~]# facter -p |grep memory_
memory_active => 63780
memory_inactive => 58188
```

You can extend the use of facts to build a completely nodeless Puppet configuration; in other words, Puppet can decide what resources to apply to a machine, based solely on the results of facts. Jordan Sissel has written about this approach at `http://www.semicomplete.com/blog/geekery/puppet-nodeless-configuration.html`.

You can find out more about custom facts, including how to make sure that OS-specific facts work only on the relevant systems, and how to weigh facts so that they're evaluated in a specific order at the puppetlabs website:

`http://docs.puppetlabs.com/guides/custom_facts.html`

See also

- The *Importing dynamic information* recipe in *Chapter 3*, *Writing Better Manifests*
- The *Configuring Hiera* recipe in *Chapter 2*, *Puppet Infrastructure*

Adding external facts

The *Creating custom facts* recipe describes how to add extra facts written in Ruby. You can also create facts from simple text files or scripts with external facts instead.

External facts live in the `/etc/facter/facts.d` directory and have a simple `key=value` format like this:

```
message="Hello, world"
```

Getting ready

Here's what you need to do to prepare your system to add external facts:

1. You'll need Facter Version 1.7 or higher to use external facts, so look up the value of `facterversion` or use `facter -v`:

   ```
   [root@cookbook ~]# facter facterversion
   2.3.0
   [root@cookbook ~]# facter -v
   2.3.0
   ```

2. You'll also need to create the external facts directory, using the following command:

   ```
   [root@cookbook ~]# mkdir -p /etc/facter/facts.d
   ```

How to do it...

In this example, we'll create a simple external fact that returns a message, as shown in the *Creating custom facts* recipe:

1. Create the file `/etc/facter/facts.d/local.txt` with the following contents:

   ```
   model=ED-209
   ```

2. Run the following command:

   ```
   [root@cookbook ~]# facter model
   ED-209
   ```

Well, that was easy! You can add more facts to the same file, or other files, of course, as follows:

```
model=ED-209
builder=OCP
directives=4
```

However, what if you need to compute a fact in some way, for example, the number of logged-in users? You can create executable facts to do this.

3. Create the file `/etc/facter/facts.d/users.sh` with the following contents:

   ```
   #!/bin/sh
   echo users=`who |wc -l`
   ```

4. Make this file executable with the following command:

   ```
   [root@cookbook ~]# chmod a+x /etc/facter/facts.d/users.sh
   ```

5. Now check the `users` value with the following command:

   ```
   [root@cookbook ~]# facter users
   2
   ```

How it works...

In this example, we'll create an external fact by creating files on the node. We'll also show how to override a previously defined fact.

1. Current versions of Facter will look into `/etc/facter/facts.d` for files of type `.txt`, `.json`, or `.yaml`. If facter finds a text file, it will parse the file for `key=value` pairs and add the key as a new fact:

   ```
   [root@cookbook ~]# facter model
   ED-209
   ```

2. If the file is a YAML or JSON file, then facter will parse the file for `key=value` pairs in the respective format. For YAML, for instance:

   ```
   ---
   registry: NCC-68814
   class: Andromeda
   shipname: USS Prokofiev
   ```

3. The resulting output will be as follows:

   ```
   [root@cookbook ~]# facter registry class shipname
   class => Andromeda
   registry => NCC-68814
   shipname => USS Prokofiev
   ```

4. In the case of executable files, Facter will assume that their output is a list of `key=value` pairs. It will execute all the files in the `facts.d` directory and add their output to the internal fact hash.

   In Windows, batch files or PowerShell scripts may be used in the same way that executable scripts are used in Linux.

5. In the `users` example, Facter will execute the `users.sh` script, which results in the following output:

   ```
   users=2
   ```

6. It will then search this output for `users` and return the matching value:

   ```
   [root@cookbook ~]# facter users
   2
   ```

7. If there are multiple matches for the key you specified, Facter determines which fact to return based on a weight property. In my version of facter, the weight of external facts is 10,000 (defined in `facter/util/directory_loader.rb` as `EXTERNAL_FACT_WEIGHT`). This high value is to ensure that the facts you define can override the supplied facts. For example:

   ```
   [root@cookbook ~]# facter architecture
   x86_64
   [root@cookbook ~]# echo "architecture=ppc64">>/etc/facter/facts.d/
   myfacts.txt
   [root@cookbook ~]# facter architecture
   ppc64
   ```

There's more...

Since all external facts have a weight of 10,000, the order in which they are parsed within the `/etc/facter/facts.d` directory sets their precedence (with the last one encountered having the highest precedence). To create a fact that will be favored over another, you'll need to have it created in a file that comes last alphabetically:

```
[root@cookbook ~]# facter architecture
ppc64
[root@cookbook ~]# echo "architecture=r10000" >>/etc/facter/facts.d/z-
architecture.txt
[root@cookbook ~]# facter architecture
r10000
```

Debugging external facts

If you're having trouble getting Facter to recognize your external facts, run Facter in debug mode to see what's happening:

```
ubuntu@cookbook:~/puppet$ facter -d robin
Fact file /etc/facter/facts.d/myfacts.json was parsed but returned an
empty data set
```

The `X` JSON file was parsed but returned an empty data set error, which means Facter didn't find any `key=value` pairs in the file or (in the case of an executable fact) in its output.

Note that if you have external facts present, Facter parses or runs all the facts in the `/etc/facter/facts.d` directory every time you query Facter. If some of these scripts take a long time to run, that can significantly slow down anything that uses Facter (run Facter with the `--iming` switch to troubleshoot this). Unless a particular fact needs to be recomputed every time it's queried, consider replacing it with a cron job that computes it every so often and writes the result to a text file in the Facter directory.

Using external facts in Puppet

Any external facts you create will be available to both Facter and Puppet. To reference external facts in your Puppet manifests, just use the fact name in the same way you would for a built-in or custom fact:

```
notify { "There are $::users people logged in right now.": }
```

Unless you are specifically attempting to override a defined fact, you should avoid using the name of a predefined fact.

See also

- The *Importing dynamic information* recipe in *Chapter 3*, *Writing Better Manifests*
- The *Configuring Hiera* recipe in *Chapter 2*, *Puppet Infrastructure*
- The *Creating custom facts* recipe in this chapter

Setting facts as environment variables

Another handy way to get information into Puppet and Facter is to pass it using environment variables. Any environment variable whose name starts with `FACTER_` will be interpreted as a fact. For example, ask facter the value of hello using the following command:

```
[root@cookbook ~]# facter -p hello
Hello, world
```

Now override the value with an environment variable and ask again:

```
[root@cookbook ~]# FACTER_hello='Howdy!' facter -p hello
Howdy!
```

It works just as well with Puppet, so let's run through an example.

How to do it...

In this example we'll set a fact using an environment variable:

1. Keep the node definition for cookbook the same as our last example:

   ```
   node cookbook {
     notify {"$::hello": }
   }
   ```

2. Run the following command:

   ```
   [root@cookbook ~]# FACTER_hello="Hallo Welt" puppet agent -t
   Info: Caching catalog for cookbook.example.com
   Info: Applying configuration version '1416212026'
   Notice: Hallo Welt
   Notice: /Stage[main]/Main/Node[cookbook]/Notify[Hallo Welt]/
   message: defined 'message' as 'Hallo Welt'
   Notice: Finished catalog run in 0.27 seconds
   ```

Generating manifests with the Puppet resource command

If you have a server that is already configured as it needs to be, or nearly so, you can capture that configuration as a Puppet manifest. The Puppet resource command generates Puppet manifests from the existing configuration of a system. For example, you can have `puppet resource` generate a manifest that creates all the users found on the system. This is very useful to take a snapshot of a working system and get its configuration quickly into Puppet.

How to do it...

Here are some examples of using `puppet resource` to get data from a running system:

1. To generate the manifest for a particular user, run the following command:

   ```
   [root@cookbook ~]# puppet resource user thomas
   user { 'thomas':
   ```

```
  ensure           => 'present',
  comment          => 'thomas Admin User',
  gid              => '1001',
  groups           => ['bin', 'wheel'],
  home             => '/home/thomas',
  password         => '!!',
  password_max_age => '99999',
  password_min_age => '0',
  shell            => '/bin/bash',
  uid              => '1001',
}
```

2. For a particular service, run the following command:

```
[root@cookbook ~]# puppet resource service sshd
service { 'sshd':
  ensure => 'running',
  enable => 'true',
}
```

3. For a package, run the following command:

```
[root@cookbook ~]# puppet resource package kernel
package { 'kernel':
  ensure => '2.6.32-431.23.3.el6',
}
```

There's more...

You can use `puppet resource` to examine each of the resource types available in Puppet. In the preceding examples, we generated a manifest for a specific instance of the resource type, but you can also use `puppet resource` to dump all instances of the resource:

```
[root@cookbook ~]# puppet resource service
service { 'abrt-ccpp':
  ensure => 'running',
  enable => 'true',
}
service { 'abrt-oops':
  ensure => 'running',
  enable => 'true',
}
```

```
        service { 'abrtd':
          ensure => 'running',
          enable => 'true',
        }
        service { 'acpid':
          ensure => 'running',
          enable => 'true',
        }
        service { 'atd':
          ensure => 'running',
          enable => 'true',
        }
        service { 'auditd':
          ensure => 'running',
          enable => 'true',
        }
```

This will output the state of each service on the system; this is because each service is an enumerable resource. When you try the same command with a resource that is not enumerable, you get an error message:

```
[root@cookbook ~]# puppet resource file
Error: Could not run: Listing all file instances is not supported.
Please specify a file or directory, e.g. puppet resource file /etc
```

Asking Puppet to describe each file on the system will not work; that's something best left to an audit tool such as `tripwire` (a system designed to look for changes on every file on the system, `http://www.tripwire.com`).

Generating manifests with other tools

If you want to quickly capture the complete configuration of a running system as a Puppet manifest, there are a couple of tools available to help. In this example, we'll look at Blueprint, which is designed to examine a machine and dump its state as Puppet code.

Getting ready

Here's what you need to do to prepare your system to use Blueprint.

Run the following command to install Blueprint; we'll use `puppet resource` here to change the state of the `python-pip` package:

```
[root@cookbook ~]# puppet resource package python-pip ensure=installed
Notice: /Package[python-pip]/ensure: created
```

```
package { 'python-pip':
  ensure => '1.3.1-4.el6',
}
[root@cookbook ~]# pip install blueprint
Downloading/unpacking blueprint
  Downloading blueprint-3.4.2.tar.gz (59kB): 59kB downloaded
  Running setup.py egg_info for package blueprint
Installing collected packages: blueprint
  Running setup.py install for blueprint
    changing mode of build/scripts-2.6/blueprint from 644 to 755
...
Successfully installed blueprint
Cleaning up...
```

You may need to install Git on your cookbook node if it is not already installed.

How to do it...

These steps will show you how to run Blueprint:

1. Run the following commands:

```
[root@cookbook ~]# mkdir blueprint && cd blueprint
[root@cookbook blueprint]# blueprint create -P blueprint_test
# [blueprint] searching for APT packages to exclude
# [blueprint] searching for Yum packages to exclude
# [blueprint] caching excluded Yum packages
# [blueprint] parsing blueprintignore(5) rules
# [blueprint] searching for npm packages
# [blueprint] searching for configuration files
# [blueprint] searching for APT packages
# [blueprint] searching for PEAR/PECL packages
# [blueprint] searching for Python packages
# [blueprint] searching for Ruby gems
# [blueprint] searching for software built from source
# [blueprint] searching for Yum packages
# [blueprint] searching for service dependencies
blueprint_test/manifests/init.pp
```

2. Read the `blueprint_test/manifests/init.pp` file to see the generated code:

```
#
# Automatically generated by blueprint(7).  Edit at your own risk.
#
class blueprint_test {
  Exec {
    path => '/usr/lib64/qt-3.3/bin:/usr/local/sbin:/usr/local/
bin:/sbin:/bin:/usr/sbin:/usr/bin:/root/bin',
  }
  Class['sources'] -> Class['files'] -> Class['packages']
    class files {
      file {
        '/etc':
          ensure => directory;
        '/etc/aliases.db':
content => template('blueprint_test/etc/aliases.db'),
          ensure   => file,
group    => root,
          mode     => 0644,
          owner    => root;
'/etc/audit':
          ensure => directory;
'/etc/audit/audit.rules':
          content => template('blueprint_test/etc/audit/audit.
rules'),
          ensure   => file,
          group    => root,
          mode     => 0640,
          owner    => root;
        '/etc/blkid':
          ensure => directory;
'/etc/cron.hourly':
          ensure => directory;
'/etc/cron.hourly/run-backup':
          content => template('blueprint_test/etc/cron.hourly/run-
backup'),
          ensure   => file,
          group    => root,
          mode     => 0755,
owner    => root;
'/etc/crypttab':
          content => template('blueprint_test/etc/crypttab'),
          ensure   => file,
          group    => root,
          mode     => 0644,
          owner    => root;
```

There's more...

Blueprint just takes a snapshot of the system as it stands; it makes no intelligent decisions, and Blueprint captures all the files on the system and all the packages. It will generate a configuration much larger than you may actually require. For instance, when configuring a server, you may specify that you want the Apache package installed. The dependencies for the Apache package will be installed automatically and you need to specify them. When generating the configuration with a tool such as Blueprint, you will capture all those dependencies and lock the versions that are installed on your system currently. Looking at our generated Blueprint code, we can see that this is the case:

```
class yum {
  package {
    'GeoIP':
      ensure => '1.5.1-5.el6.x86_64';
    'PyXML':
      ensure => '0.8.4-19.el6.x86_64';
    'SDL':
      ensure => '1.2.14-3.el6.x86_64';
    'apr':
      ensure => '1.3.9-5.el6_2.x86_64';
    'apr-util':
      ensure => '1.3.9-3.el6_0.1.x86_64';
```

If you were creating this manifest yourself, you would likely specify `ensure => installed` instead of a specific version.

Packages install default versions of files. Blueprint has no notion of this and will add all the files to the manifest, even those that have not changed. By default, Blueprint will indiscriminately capture all the files in `/etc` as file resources.

Blueprint and similar tools have a very small use case generally, but may help you to get familiar with the Puppet syntax and give you some ideas on how to specify your own manifests. I would not recommend blindly using this tool to create a system, however.

There's no shortcut to good configuration management, those who hope to save time and effort by cutting and pasting someone else's code as a whole (as with public modules) are likely to find that it saves neither.

Using an external node classifier

When Puppet runs on a node, it needs to know which classes should be applied to that node. For example, if it is a web server node, it might need to include an `apache` class. The normal way to map nodes to classes is in the Puppet manifest itself, for example, in your `site.pp` file:

```
node 'web1' {
  include apache
}
```

Alternatively, you can use an **External Node Classifier (ENC)** to do this job. An ENC is any executable program that can accept the fully-qualified domain name (FQDN) as the first command-line argument (`$1`). The script is expected to return a list of classes, parameters, and an optional environment to apply to the node. The output is expected to be in the standard YAML format. When using an ENC, you should keep in mind that the classes applied through the standard `site.pp` manifest are merged with those provided by the ENC.

[Parameters returned by the ENC are available as top-scope variables to the node.]

An ENC could be a simple shell script, for example, or a wrapper around a more complicated program or API that can decide how to map nodes to classes. The ENC provided by Puppet enterprise and The Foreman (`http://theforeman.org/`) are both simple scripts, which connect to the web API of their respective systems.

In this example, we'll build the most simple of ENCs, a shell script that simply prints a list of classes to include. We'll start by including an `enc` class, which defines `notify` that will print a top-scope variable `$enc`.

Getting ready

We'll start by creating our `enc` class to include with the `enc` script:

1. Run the following command:

```
t@mylaptop ~/puppet $ mkdir -p modules/enc/manifests
```

2. Create the file `modules/enc/manifests/init.pp` with the following contents:

```
class enc {
  notify {"We defined this from $enc": }
}
```

How to do it...

Here's how to build a simple external node classifier. We'll perform all these steps on our Puppet master server. If you are running masterless, then do these steps on a node:

1. Create the file `/etc/puppet/cookbook.sh` with the following contents:

```
#!/bin/bash
cat <<EOF
---
classes:
enc:
parameters:
  enc: $0
EOF
```

2. Run the following command:

```
root@puppet:/etc/puppet# chmod a+x cookbook.sh
```

3. Modify your `/etc/puppet/puppet.conf` file as follows:

```
[main]
  node_terminus = exec
  external_nodes = /etc/puppet/cookbook.sh
```

4. Restart Apache (restart the master) to make the change effective.
5. Ensure your `site.pp` file has the following empty definition for the default node:

```
node default {}
```

6. Run Puppet:

```
[root@cookbook ~]# puppet agent -t
Info: Caching catalog for cookbook.example.com
Info: Applying configuration version '1416376937'
Notice: We defined this from /etc/puppet/cookbook.sh
Notice: /Stage[main]/Enc/Notify[We defined this from /etc/puppet/
cookbook.sh]/message: defined 'message' as 'We defined this from /
etc/puppet/cookbook.sh'
Notice: Finished catalog run in 0.17 seconds
```

How it works...

When an ENC is set in `puppet.conf`, Puppet will call the specified program with the node's fqdn (technically, the certname variable) as the first command-line argument. In our example script, this argument is ignored, and it just outputs a fixed list of classes (actually, just one class).

Obviously this script is not terribly useful; a more sophisticated script might check a database to find the class list, or look up the node in a hash, or an external text file or database (often an organization's configuration management database, **CMDB**). Hopefully, this example is enough to get you started with writing your own external node classifier. Remember that you can write your script in any language you prefer.

There's more...

An ENC can supply a whole list of classes to be included in the node, in the following (YAML) format:

```
---
classes:
  CLASS1:
  CLASS2:
  CLASS3:
```

For classes that take parameters, you can use this format:

```
---
classes:
  mysql:
    package: percona-server-server-5.5
    socket:  /var/run/mysqld/mysqld.sock
    port:    3306
```

You can also produce top-scope variables using an ENC with this format:

```
---
parameters:
  message: 'Anyone home MyFly?'
```

Variables that you set in this way will be available in your manifest using the normal syntax for a top-scope variable, for example `$::message`.

See also

- See the puppetlabs ENC page for more information on writing and using ENCs: `http://docs.puppetlabs.com/guides/external_nodes.html`

Creating your own resource types

As you know, Puppet has a bunch of useful built-in resource types: packages, files, users, and so on. Usually, you can do everything you need to do by using either combinations of these built-in resources, or `define`, which you can use more or less in the same way as a resource (see *Chapter 3*, *Writing Better Manifests* for information on definitions).

In the early days of Puppet, creating your own resource type was more common as the list of core resources was shorter than it is today. Before you consider creating your own resource type, I suggest searching the Forge for alternative solutions. Even if you can find a project that only partially solves your problem, you will be better served by extending and helping out that project, rather than trying to create your own. However, if you need to create your own resource type, Puppet makes it quite easy. The native types are written in Ruby, and you will need a basic familiarity with Ruby in order to create your own.

Let's refresh our memory on the distinction between types and providers. A type describes a resource and the parameters it can have (for example, the `package` type). A provider tells Puppet how to implement a resource type for a particular platform or situation (for example, the `apt/dpkg` providers implement the `package` type for Debian-like systems).

A single type (`package`) can have many providers (APT, YUM, Fink, and so on). If you don't specify a provider when declaring a resource, Puppet will choose the most appropriate one given the environment.

We'll use Ruby in this section; if you are not familiar with Ruby try visiting `http://www.ruby-doc.org/docs/Tutorial/` or `http://www.codecademy.com/tracks/ruby/`.

How to do it...

In this section, we'll see how to create a custom type that we can use to manage Git repositories, and in the next section, we'll write a provider to implement this type.

Create the file `modules/cookbook/lib/puppet/type/gitrepo.rb` with the following contents:

```
Puppet::Type.newtype(:gitrepo) do
  ensurable

  newparam(:source) do
    isnamevar
  end

  newparam(:path)
end
```

How it works...

Custom types can live in any module, in a `lib/puppet/type` subdirectory and in a file named for the type (in our example, that's `modules/cookbook/lib/puppet/type/gitrepo.rb`).

The first line of `gitrepo.rb` tells Puppet to register a new type named `gitrepo`:

```
Puppet::Type.newtype(:gitrepo) do
```

The `ensurable` line automatically gives the type an `ensure` property, such as Puppet's built-in resources:

```
ensurable
```

We'll now give the type some parameters. For the moment, all we need is a `source` parameter for the Git source URL, and a `path` parameter to tell Puppet where the repo should be created in the filesystem:

```
newparam(:source) do
  isnamevar
end
```

The `isnamevar` declaration tells Puppet that the `source` parameter is the type's namevar. So when you declare an instance of this resource, whatever name you give, it will be the value of `source`, for example:

```
gitrepo { 'git://github.com/puppetlabs/puppet.git':
  path => '/home/ubuntu/dev/puppet',
}
```

Finally, we tell Puppet that the type accepts the `path` parameter:

```
newparam(:path)
```

There's more...

When deciding whether or not you should create a custom type, you should ask a few questions about the resource you are trying to describe such as:

- Is the resource enumerable? Can you easily obtain a list of all the instances of the resource on the system?
- Is the resource atomic? Can you ensure that only one copy of the resource exists on the system (this is particularly important when you want to use `ensure=>absent` on the resource)?
- Is there any other resource that describes this resource? In such a case, a defined type based on the existing resource would, in most cases, be a simpler solution.

Documentation

Our example is deliberately simple, but when you move on to developing real custom types for your production environment, you should add documentation strings to describe what the type and its parameters do, for example:

```
Puppet::Type.newtype(:gitrepo) do
  @doc = "Manages Git repos"

  ensurable

  newparam(:source) do
    desc "Git source URL for the repo"
    isnamevar
  end

  newparam(:path) do
    desc "Path where the repo should be created"
  end
end
```

Validation

You can use parameter validation to generate useful error messages when someone tries to pass bad values to the resource. For example, you could validate that the directory where the repo is to be created actually exists:

```
newparam(:path) do
  validate do |value|
    basepath = File.dirname(value)
    unless File.directory?(basepath)
      raise ArgumentError , "The path %s doesn't exist" % basepath
    end
  end
end
```

You can also specify the list of allowed values that the parameter can take:

```
newparam(:breakfast) do
  newvalues(:bacon, :eggs, :sausages)
end
```

Creating your own providers

In the previous section, we created a new custom type called `gitrepo` and told Puppet that it takes two parameters, `source` and `path`. However, so far, we haven't told Puppet how to actually check out the repo; in other words, how to create a specific instance of this type. That's where the provider comes in.

We saw that a type will often have several possible providers. In our example, there is only one sensible way to instantiate a Git repo, so we'll only supply one provider: `git`. If you were to generalize this type—to just repo, say—it's not hard to imagine creating several different providers depending on the type of repo, for example, `git`, `svn`, `cvs`, and so on.

How to do it...

We'll add the `git` provider, and create an instance of a `gitrepo` resource to check that it all works. You'll need Git installed for this to work, but if you're using the Git-based manifest management setup described in *Chapter 2*, *Puppet Infrastructure*, we can safely assume that Git is available.

1. Create the file `modules/cookbook/lib/puppet/provider/gitrepo/git.rb` with the following contents:

```
require 'fileutils'

Puppet::Type.type(:gitrepo).provide(:git) do
  commands :git => "git"

  def create
    git "clone", resource[:source], resource[:path]
  end

  def exists?
    File.directory? resource[:path]
  end
end
```

2. Modify your `site.pp` file as follows:

```
node 'cookbook' {
  gitrepo { 'https://github.com/puppetlabs/puppetlabs-git':
    ensure => present,
    path   => '/tmp/puppet',
  }
}
```

3. Run Puppet:

```
[root@cookbook ~]# puppet agent -t
Notice: /File[/var/lib/puppet/lib/puppet/type/gitrepo.rb]/ensure:
defined content as '{md5}6471793fe2b4372d40289ad4b614fe0b'
Notice: /File[/var/lib/puppet/lib/puppet/provider/gitrepo]/ensure:
created
Notice: /File[/var/lib/puppet/lib/puppet/provider/gitrepo/git.rb]/
ensure: defined content as '{md5}f860388234d3d0bdb3b3ec98bbf5115b'
Info: Caching catalog for cookbook.example.com
Info: Applying configuration version '1416378876'
Notice: /Stage[main]/Main/Node[cookbook]/Gitrepo[https://github.
com/puppetlabs/puppetlabs-git]/ensure: created
Notice: Finished catalog run in 2.59 seconds
```

How it works...

Custom providers can live in any module, in a `lib/puppet/provider/TYPE_NAME` subdirectory in a file named after the provider. (The provider is the actual program that is run on the system; in our example, the program is Git and the provider is in `modules/cookbook/lib/puppet/provider/gitrepo/git.rb`. Note that the name of the module is irrelevant.)

After an ntitial require line in `git.rb`, we tell Puppet to register a new provider for the `gitrepo` type with the following line:

```
Puppet::Type.type(:gitrepo).provide(:git) do
```

When you declare an instance of the `gitrepo` type in your manifest, Puppet will first of all check whether the instance already exists, by calling the `exists?` method on the provider. So we need to supply this method, complete with code to check whether an instance of the `gitrepo` type already exists:

```
def exists?
  File.directory? resource[:path]
end
```

This is not the most sophisticated implementation; it simply returns `true` if a directory exists matching the `path` parameter of the instance. A better implementation of `exists?` might check, for example, whether there is a `.git` subdirectory and that it contains valid Git metadata. But this will do for now.

If `exists?` returns `true`, then Puppet will take no further action because the specified resource exists (as far as Puppet knows). If it returns `false`, Puppet assumes the resource doesn't yet exist, and will try to create it by calling the provider's `create` method.

Accordingly, we supply some code for the `create` method that calls the `git clone` command to create the repo:

```
def create
  git "clone", resource[:source], resource[:path]
end
```

The method has access to the instance's parameters, which we need to know where to check out the repo from, and which directory to create it in. We get this by looking at `resource[:source]` and `resource[:path]`.

There's more...

You can see that custom types and providers in Puppet are very powerful. In fact, they can do anything—at least, anything that Ruby can do. If you are managing some parts of your infrastructure with complicated `define` statements and `exec` resources, you may want to consider replacing these with a custom type. However, as stated previously, it's worth looking around to see if someone else has already done this before implementing your own.

Our example was very simple, and there is much more to learn about writing your own types. If you're going to distribute your code for others to use, or even if you aren't, it's a good idea to include tests with it. puppetlabs has a useful page on the interface between custom types and providers:

`http://docs.puppetlabs.com/guides/custom_types.html`

on implementing providers:

`http://docs.puppetlabs.com/guides/provider_development.html`

and a complete worked example of developing a custom type and provider, a little more advanced than that presented in this book:

`http://docs.puppetlabs.com/guides/complete_resource_example.html`

Creating custom functions

If you've read the recipe *Using GnuPG to encrypt secrets* in *Chapter 4*, *Working with Files and Packages*, then you've already seen an example of a custom function (in that example, we created a `secret` function, which shelled out to GnuPG). Let's look at `custom` functions in a little more detail now and build an example.

How to do it...

If you've read the recipe *Distributing cron jobs efficiently* in *Chapter 6*, *Managing Resources and Files*, you might remember that we used the `inline_template` function to set a random time for cron jobs to run, based on the hostname of the node. In this example, we'll take that idea and turn it into a custom function called `random_minute`:

1. Create the file `modules/cookbook/lib/puppet/parser/functions/random_minute.rb` with the following contents:

```
module Puppet::Parser::Functions
  newfunction(:random_minute, :type => :rvalue) do |args|
    lookupvar('hostname').sum % 60
  end
end
```

2. Modify your `site.pp` file as follows:

```
node 'cookbook' {
  cron { 'randomised cron job':
    command => '/bin/echo Hello, world >>/tmp/hello.txt',
    hour    => '*',
    minute  => random_minute(),
  }
}
```

3. Run Puppet:

```
[root@cookbook ~]# puppet agent -t
Info: Retrieving pluginfacts
Info: Retrieving plugin
Notice: /File[/var/lib/puppet/lib/puppet/parser/functions/
random_minute.rb]/ensure: defined content as '{md5}
e6ff40165e74677e5837027bb5610744'
Info: Loading facts
Info: Caching catalog for cookbook.example.com
Info: Applying configuration version '1416379652'
Notice: /Stage[main]/Main/Node[cookbook]/Cron[custom fuction
example job]/ensure: created
Notice: Finished catalog run in 0.41 seconds
```

4. Check `crontab` with the following command:

```
[root@cookbook ~]# crontab -l
# HEADER: This file was autogenerated at Wed Nov 19 01:48:11 -0500
2014 by puppet.
# HEADER: While it can still be managed manually, it is definitely
not recommended.
# HEADER: Note particularly that the comments starting with
'Puppet Name' should
# HEADER: not be deleted, as doing so could cause duplicate cron
jobs.
# Puppet Name: run-backup
0 15 * * * /usr/local/bin/backup
# Puppet Name: custom fuction example job
15 * * * * /bin/echo Hallo, welt >>/tmp/hallo.txt
```

How it works...

Custom functions can live in any module, in the `lib/puppet/parser/functions` subdirectory in a file named after the function (in our example, `random_minute.rb`).

The function code goes inside a `module ... end` block like this:

```
module Puppet::Parser::Functions
  ...
end
```

We then call `newfunction` to declare our new function, passing the name (`:random_minute`) and the type of function (`:rvalue`):

```
newfunction(:random_minute, :type => :rvalue) do |args|
```

The `:rvalue` bit simply means that this function returns a value.

Finally, the function code itself is as follows:

```
    lookupvar('hostname').sum % 60
```

The `lookupvar` function lets you access facts and variables by name; in this case, `hostname` to get the name of the node we're running on. We use the Ruby `sum` method to get the numeric sum of the characters in this string, and then perform integer division modulo 60 to make sure the result is in the range `0..59`.

There's more...

You can, of course, do a lot more with custom functions. In fact, anything you can do in Ruby, you can do in a custom function. You also have access to all the facts and variables that are in scope at the point in the Puppet manifest where the function is called, by calling `lookupvar` as shown in the example. You can also work on arguments, for example, a general purpose hashing function that takes two arguments: the size of the hash table and optionally the thing to hash. Create `modules/cookbook/lib/puppet/parser/functions/hashtable.rb` with the following contents:

```
module Puppet::Parser::Functions
  newfunction(:hashtable, :type => :rvalue) do |args|
    if args.length == 2
      hashtable=lookupvar(args[1]).sum
    else
      hashtable=lookupvar('hostname').sum
    end

    if args.length > 0
      size = args[0].to_i
    else
      size = 60
    end
    unless size == 0
      hashtable % size
    else
      0
    end
  end
end
```

Now we'll create a test for our `hashtable` function and alter `site.pp` as follows:

```
node cookbook {
  $hours = hashtable(24)
  $minutes = hashtable()
  $days = hashtable(30)
  $days_fqdn = hashtable(30,'fqdn')
  $days_ipaddress = hashtable(30,'ipaddress')
  notify {"\n hours=${hours}\n minutes=${minutes}\n days=${days}\n
    days_fqdn=${days_fqdn}\n days_ipaddress=${days_ipaddress}\n":}
}
```

Now, run Puppet and observe the values that are returned:

```
Notice:  hours=15
 minutes=15
 days=15
 days_fqdn=4
 days_ipaddress=2
```

Our simple definition quickly grew when we added the ability to add arguments. As with all programming, care should be taken when working with arguments to ensure that you do not have any error conditions. In the preceding code, we specifically looked for the situation where the size variable was 0, to avoid a divide by zero error.

To find out more about what you can do with custom functions, see the puppetlabs website:

`http://docs.puppetlabs.com/guides/custom_functions.html`

Testing your puppet manifests with rspec-puppet

It would be great if we could verify that our Puppet manifests satisfy certain expectations without even having to run Puppet. The `rspec-puppet` tool is a nifty tool to do this. Based on RSpec, a testing framework for Ruby programs, `rspec-puppet` lets you write test cases for your Puppet manifests that are especially useful to catch regressions (bugs introduced when fixing another bug), and refactoring problems (bugs introduced when reorganizing your code).

Getting ready

Here's what you'll need to do to install `rspec-puppet`.

Run the following commands:

```
t@mylaptop~ $ sudo puppet resource package rspec-puppet ensure=installed
provider=gem
Notice: /Package[rspec-puppet]/ensure: created
package { 'rspec-puppet':
  ensure => ['1.0.1'],
}
t@mylaptop ~ $ sudo puppet resource package puppetlabs_spec_helper
ensure=installed provider=gem
Notice: /Package[puppetlabs_spec_helper]/ensure: created
package { 'puppetlabs_spec_helper':
  ensure => ['0.8.2'],
}
```

How to do it...

Let's create an example class, `thing`, and write some tests for it.

1. Define the `thing` class:

```
class thing {
  service {'thing':
    ensure  => 'running',
    enable  => true,
    require => Package['thing'],
  }
  package {'thing':
    ensure => 'installed'
  }
  file {'/etc/thing.conf':
    content => 'fubar\n',
    mode    => 0644,
    require => Package['thing'],
    notify  => Service['thing'],
  }
}
```

2. Run the following commands:

```
t@mylaptop ~/puppet]$cd modules/thing
t@mylaptop~/puppet/modules/thing $ rspec-puppet-init
 + spec/
 + spec/classes/
 + spec/defines/
 + spec/functions/
 + spec/hosts/
 + spec/fixtures/
 + spec/fixtures/manifests/
 + spec/fixtures/modules/
 + spec/fixtures/modules/heartbeat/
 + spec/fixtures/manifests/site.pp
 + spec/fixtures/modules/heartbeat/manifests
 + spec/fixtures/modules/heartbeat/templates
 + spec/spec_helper.rb
 + Rakefile
```

3. Create the file `spec/classes/thing_spec.rb` with the following contents:

```
require 'spec_helper'

describe 'thing' do
  it { should create_class('thing') }
  it { should contain_package('thing') }
  it { should contain_service('thing').with(
    'ensure' => 'running'
  ) }
  it { should contain_file('/etc/things.conf') }
end
```

4. Run the following commands:

```
t@mylaptop ~/.puppet/modules/thing $ rspec
...F

Failures:

  1) thing should contain File[/etc/things.conf]
     Failure/Error: it { should contain_file('/etc/things.conf') }
       expected that the catalogue would contain File[/etc/things.
conf]
     # ./spec/classes/thing_spec.rb:9:in `block (2 levels) in <top
(required)>'

Finished in 1.66 seconds
4 examples, 1 failure

Failed examples:

rspec ./spec/classes/thing_spec.rb:9 # thing should contain File[/
etc/things.conf]
```

How it works...

The `rspec-puppet-init` command creates a framework of directories for you to put your specs (test programs) in. At the moment, we're just interested in the `spec/classes` directory. This is where you'll put your class specs, one per class, named after the class it tests, for example, `thing_spec.rb`.

The `spec` code itself begins with the following statement, which sets up the RSpec environment to run the specs:

```
require 'spec_helper'
```

Then, a `describe` block follows:

```
describe 'thing' do
  ..
end
```

The `describe` identifies the class we're going to test (`thing`) and wraps the list of assertions about the class inside a `do .. end` block.

Assertions are our stated expectations of the `thing` class. For example, the first assertion is the following:

```
    it { should create_class('thing') }
```

The `create_class` assertion is used to ensure that the named class is actually created. The next line:

```
    it { should contain_package('thing') }
```

The `contain_package` assertion means what it says: the class should contain a package resource named `thing`.

Next, we test for the existence of the `thing` service:

```
it { should contain_service('thing').with(
  'ensure' => 'running'
) }
```

The preceding code actually contains two assertions. First, that the class contains a `thing` service:

```
contain_service('thing')
```

Second, that the service has an `ensure` attribute with the value `running`:

```
with(
  'ensure' => 'running'
)
```

You can specify any attributes and values you want using the `with` method, as a comma-separated list. For example, the following code asserts several attributes of a `file` resource:

```
it { should contain_file('/tmp/hello.txt').with(
  'content' => "Hello, world\n",
  'owner'   => 'ubuntu',
  'group'   => 'ubuntu',
  'mode'    => '0644'
) }
```

In our `thing` example, we need to only test that the file `thing.conf` is present, using the following code:

```
    it { should contain_file('/etc/thing.conf') }
```

When you run the `rake spec` command, `rspec-puppet` will compile the relevant Puppet classes, run all the specs it finds, and display the results:

```
...F
Failures:
  1) thing should contain File[/etc/things.conf]
     Failure/Error: it { should contain_file('/etc/things.conf') }
       expected that the catalogue would contain File[/etc/things.conf]
     # ./spec/classes/thing_spec.rb:9:in `block (2 levels) in <top
(required)>'
Finished in 1.66 seconds
4 examples, 1 failure
```

As you can see, we defined the file in our test as `/etc/things.conf` but the file in the manifests is `/etc/thing.conf`, so the test fails. Edit `thing_spec.rb` and change `/etc/things.conf to /etc/thing.conf`:

```
      it { should contain_file('/etc/thing.conf') }
```

Now run rspec again:

```
t@mylaptop ~/.puppet/modules/thing $ rspec
....
Finished in 1.6 seconds
4 examples, 0 failures
```

There's more...

There are many conditions you can verify with rspec. Any resource type can be verified with `contain_<resource type>`(title). In addition to verifying your classes will apply correctly, you can also test functions and definitions by using the appropriate subdirectories within the spec directory (classes, defines, or functions).

You can find more information about `rspec-puppet`, including complete documentation for the assertions available and a tutorial, at `http://rspec-puppet.com/`.

When you want to start testing how your code applies to nodes, you'll need to look at another tool, beaker. Beaker works with various virtualization platforms to create temporary virtual machines to which Puppet code is applied. The results are then used for acceptance testing of the Puppet code. This method of testing and developing at the same time is known as **Test-driven development** (**TDD**). More information about beaker is available on the GitHub site at `https://github.com/puppetlabs/beaker`.

See also

- The *Checking your manifests with puppet-lint* recipe in *Chapter 1*, *Puppet Language and Style*

Using librarian-puppet

When you begin to include modules from the forge in your Puppet infrastructure, keeping track of which versions you installed and ensuring consistency between all your testing areas can become a bit of a problem. Luckily, the tools we will discuss in the next two sections can bring order to your system. We will first begin with librarian-puppet, which uses a special configuration file named Puppetfile to specify the source location of your various modules.

Getting ready

We'll install librarian-puppet to work through the example.

Install `librarian-puppet` on your Puppet master, using Puppet of course:

```
root@puppet:~# puppet resource package librarian-puppet ensure=installed
provider=gem
Notice: /Package[librarian-puppet]/ensure: created
package { 'librarian-puppet':
  ensure => ['2.0.0'],
}
```

If you are working in a masterless environment, install `librarian-puppet` on the machine from which you will be managing your code. Your gem install may fail if the Ruby development packages are not available on your master; install the `ruby-dev` package to fix this issue (use Puppet to do it).

How to do it...

We'll use librarian-puppet to download and install a module in this example:

1. Create a working directory for yourself; librarian-puppet will overwrite your modules directory by default, so we'll work in a temporary location for now:

   ```
   root@puppet:~# mkdir librarian
   root@puppet:~# cd librarian
   ```

2. Create a new Puppetfile with the following contents:

```
#!/usr/bin/env ruby
#^syntax detection

forge "https://forgeapi.puppetlabs.com"

# A module from the Puppet Forge
mod 'puppetlabs-stdlib'
```

> Alternatively, you can use `librarian-puppet init` to create an example Puppetfile and edit it to match our example:
>
> ```
> root@puppet:~/librarian# librarian-puppet init
>       create  Puppetfile
> ```

3. Now, run librarian-puppet to download and install the `puppetlabs-stdlib` module in the `modules` directory:

```
root@puppet:~/librarian# librarian-puppet install
root@puppet:~/librarian # ls
modules  Puppetfile  Puppetfile.lock
root@puppet:~/librarian # ls modules
stdlib
```

How it works...

The first line of the `Puppetfile` makes the `Puppetfile` appear to be a Ruby source file. These are completely optional but coerces editors into treating the file as though it was written in Ruby (which it is):

```
#!/usr/bin/env ruby
```

We next define where the Puppet Forge is located; you may specify an internal Forge here if you have a local mirror:

```
forge "https://forgeapi.puppetlabs.com"
```

Now, we added a line to include the `puppetlabs-stdlib` module:

```
mod 'puppetlabs-stdlib'
```

With the Puppetfile in place, we ran librarian-puppet and it downloaded the module from the URL given in the Forge line. As the module was downloaded, librarian-puppet created a Puppetfile.lock file, which includes the location used as source and the version number for the downloaded module:

```
FORGE
  remote: https://forgeapi.puppetlabs.com
  specs:
    puppetlabs-stdlib (4.4.0)

DEPENDENCIES
  puppetlabs-stdlib (>= 0)
```

There's more...

The Puppetfile allows you to pull in modules from sources other than the forge. You may use a local Git url or even a GitHub url to download modules that are not on the Forge. More information on librarian-puppet can be found on the GitHub website at https://github.com/rodjek/librarian-puppet.

Note that librarian-puppet will create the modules directory and remove any modules you placed in there by default. Most installations using librarian-puppet opt to place their local modules in a /local subdirectory (/dist or /companyname are also used).

In the next section, we'll talk about r10k, which goes one step further than librarian and manages your entire environment directory.

Using r10k

The Puppetfile is a very good format to describe which modules you wish to include in your environment. Building upon the Puppetfile is another tool, **r10k**. r10k is a total environment management tool. You can use r10k to clone a local Git repository into your environmentpath and then place the modules specified in your Puppetfile into that directory. The local Git repository is known as the master repository; it is where r10k expects to find your Puppetfile. r10k also understands Puppet environments and will clone Git branches into subdirectories of your environmentpath, simplifying the deployment of multiple environments. What makes r10k particularly useful is its use of a local cache directory to speed up deployments. Using a configuration file, r10k.yaml, you can specify where to store this cache and also where your master repository is held.

Getting ready

We'll install r10k on our controlling machine (usually the master). This is where we will control all the modules downloaded and installed.

1. Install r10k on your puppet master, or on whichever machine you wish to manage your `environmentpath` directory:

```
root@puppet:~# puppet resource package r10k ensure=installed
provider=gem
Notice: /Package[r10k]/ensure: created
package { 'r10k':
  ensure => ['1.3.5'],
}
```

2. Make a new copy of your Git repository (optional, do this on your Git server):

```
[git@git repos]$ git clone --bare puppet.git puppet-r10k.git
Initialized empty Git repository in /home/git/repos/puppet-r10k.
git/
```

3. Check out the new Git repository (on your local machine) and move the existing modules directory to a new location. We'll use `/local` in this example:

```
t@mylaptop ~ $ git clone git@git.example.com:repos/puppet-r10k.git
Cloning into 'puppet-r10k'...
remote: Counting objects: 2660, done.
remote: Compressing objects: 100% (2136/2136), done.
remote: Total 2660 (delta 913), reused 1049 (delta 238)
Receiving objects: 100% (2660/2660), 738.20 KiB | 0 bytes/s, done.
Resolving deltas: 100% (913/913), done.
Checking connectivity... done.
t@mylaptop ~ $ cd puppet-r10k/
t@mylaptop ~/puppet-r10k $ git checkout production
Branch production set up to track remote branch production from
origin.
Switched to a new branch 'production'
t@mylaptop ~/puppet-r10k $ git mv modules local
t@mylaptop ~/puppet-r10k $ git commit -m "moving modules in
preparation for r10k"
[master c96d0dc] moving modules in preparation for r10k
 9 files changed, 0 insertions(+), 0 deletions(-)
 rename {modules => local}/base (100%)
 rename {modules => local}/puppet/files/papply.sh (100%)
 rename {modules => local}/puppet/files/pull-updates.sh (100%)
 rename {modules => local}/puppet/manifests/init.pp (100%)
```

How to do it...

We'll create a Puppetfile to control r10k and install modules on our master.

1. Create a `Puppetfile` into the new Git repository with the following contents:

   ```
   forge "http://forge.puppetlabs.com"
   mod 'puppetlabs/puppetdb', '3.0.0'
   mod 'puppetlabs/stdlib', '3.2.0'
   mod 'puppetlabs/concat'
   mod 'puppetlabs/firewall'
   ```

2. Add the `Puppetfile` to your new repository:

   ```
   t@mylaptop ~/puppet-r10k $ git add Puppetfile
   t@mylaptop ~/puppet-r10k $ git commit -m "adding Puppetfile"
   [production d42481f] adding Puppetfile
    1 file changed, 7 insertions(+)
    create mode 100644 Puppetfile
   t@mylaptop ~/puppet-r10k $ git push
   Counting objects: 7, done.
   Delta compression using up to 4 threads.
   Compressing objects: 100% (5/5), done.
   Writing objects: 100% (5/5), 589 bytes | 0 bytes/s, done.
   Total 5 (delta 2), reused 0 (delta 0)
   To git@git.example.com:repos/puppet-r10k.git
      cf8dfb9..d42481f  production -> production
   ```

3. Back to your master, create `/etc/r10k.yaml` with the following contents:

   ```
   ---
   :cachedir: '/var/cache/r10k'
   :sources:
    :plops:
     remote: 'git@git.example.com:repos/puppet-r10k.git'
     basedir: '/etc/puppet/environments'
   ```

4. Run r10k to have the `/etc/puppet/environments` directory populated (hint: create a backup of your `/etc/puppet/environments` directory first):

   ```
   root@puppet:~# r10k deploy environment -p
   ```

5. Verify that your `/etc/puppet/environments` directory has a production subdirectory. Within that directory, the `/local` directory will exist and the modules directory will have all the modules listed in the `Puppetfile`:

```
root@puppet:/etc/puppet/environments# tree -L 2
.
├── master
│   ├── manifests
│   ├── modules
│   └── README
└── production
    ├── environment.conf
    ├── local
    ├── manifests
    ├── modules
    ├── Puppetfile
    └── README
```

How it works...

We started by creating a copy of our Git repository; this was only done to preserve the earlier work and is not required. The important thing to remember with r10k and librarian-puppet is that they both assume they are in control of the `/modules` subdirectory. We need to move our modules out of the way and create a new location for the modules.

In the `r10k.yaml` file, we specified the location of our new repository. When we ran r10k, it first downloaded this repository into its local cache. Once the Git repository is downloaded locally, r10k will go through each branch and look for a `Puppetfile` within the branch. For each `branch/Puppetfile` combination, the modules specified within are downloaded first to the local cache directory (`cachedir`) and then into the `basedir`, which was given in `r10k.yaml`.

There's more...

You can automate the deployment of your environments using `r10k`. The command we used to run `r10k` and populate our environments directory can be easily placed inside a Git hook to automatically update your environment. There is also a **marionette collective** (**mcollective**) plugin (`https://github.com/acidprime/r10k`), which can be used to have `r10k` run on an arbitrary set of servers.

Using either of these tools will help keep your site consistent, even if you are not taking advantage of the various modules available on the Forge.

10 Monitoring, Reporting, and Troubleshooting

"Show me a completely smooth operation and I'll show you someone who's covering mistakes. Real boats rock."

—Frank Herbert, Chapterhouse: Dune

In this chapter, we will cover the following recipes:

- Noop: the don't change anything option
- Logging command output
- Logging debug messages
- Generating reports
- Producing automatic HTML documentation
- Drawing dependency graphs
- Understanding Puppet errors
- Inspecting configuration settings

Introduction

We've all had the experience of sitting in an exciting presentation about some new technology and rushing home to play with it. Of course, once you start experimenting with it, you immediately run into problems. What's going wrong? Why doesn't it work? How can I see what's happening under the hood? This chapter will help you answer some of these questions, and give you the tools to solve common Puppet problems.

We'll also see how to generate useful reports on your Puppet infrastructure and how Puppet can help you monitor and troubleshoot your network as a whole.

Noop – the don't change anything option

Sometimes your Puppet manifest doesn't do exactly what you expected, or perhaps someone else has checked in changes you didn't know about. Either way, it's good to know exactly what Puppet is going to do before it does it.

When you are retrofitting Puppet into an existing infrastructure you might not know whether Puppet is going to update a `config` file or restart a production service. Any such change could result in unplanned downtime. Also, sometimes manual configuration changes are made on a server that Puppet would overwrite.

To avoid these problems, you can use Puppet's noop mode, which means no operation or do nothing. When run with the noop option, Puppet only reports what it would do but doesn't actually do anything. One caveat here is that even during a noop run, pluginsync still runs and any `lib` directories in modules will be synced to nodes. This will update external fact definitions and possibly Puppet's types and providers.

How to do it...

You may run noop mode when running `puppet agent` or `puppet apply` by appending the `--noop` switch to the command. You may also create a `noop=true` line in your `puppet.conf` file within the `[agent]` or `[main]` sections.

1. Create a `noop.pp` manifest that creates a file as follows:

```
file {'/tmp/noop':
  content => 'nothing',
  mode    => 0644,
}
```

2. Now run puppet agent with the `noop` switch:

```
t@mylaptop ~/puppet/manifests $ puppet apply noop.pp --noop
Notice: Compiled catalog for mylaptop in environment production in 0.41 seconds
Notice: /Stage[main]/Main/File[/tmp/noop]/ensure: current_value absent, should be file (noop)
Notice: Class[Main]: Would have triggered 'refresh' from 1 events
Notice: Stage[main]: Would have triggered 'refresh' from 1 events
Notice: Finished catalog run in 0.02 seconds
```

3. Now run without the `noop` option to see that the file is created:

```
t@mylaptop ~/puppet/manifests $ puppet apply noop.pp
Notice: Compiled catalog for mylaptop in environment production in
0.37 seconds
Notice: /Stage[main]/Main/File[/tmp/noop]/ensure: defined content
as '{md5}3e47b75000b0924b6c9ba5759a7cf15d'
```

How it works...

In the `noop` mode, Puppet does everything it would normally, with the exception of actually making any changes to the machine (the `exec` resources, for example, won't run). It tells you what it would have done, and you can compare this with what you expected to happen. If there are any differences, double-check the manifest or the current state of the machine.

Note that when we ran with `--noop`, Puppet warned us that it would have created the `/tmp/noop` file. This may or may not be what we want, but it's useful to know in advance. If you are making changes to the code applied to your production servers, it's useful to run puppet agent with the `--noop` option to ensure that your changes will not affect the production services.

There's more...

You can also use noop mode as a simple auditing tool. It will tell you whether any changes have been made to the machine since Puppet last applied its manifest. Some organizations require all config changes to be made with Puppet, which is one way of implementing a change control process. Unauthorized changes to the resources managed by Puppet can be detected using Puppet in noop mode and you can then decide whether to merge the changes back into the Puppet manifest or undo them.

You can also use the `--debug` switch when running puppet agent to see the details of every change Puppet makes during an agent run. This can be helpful when trying to figure out how Puppet is applying certain exec resources or to see in what order things are happening.

If you are running a master, you can compile the catalog for a node on the master with the `--trace` option in addition to `--debug`. If the catalog is failing to compile, this method will also fail to compile the catalog (if you have an old definition for the cookbook node that is failing, try commenting it out before running this test). This produces a lot of debugging output. For example, to compile the catalog for our cookbook host on our master and place the results into `/tmp/cookbook.log`:

```
root@puppet: ~#puppet master --compile cookbook.example.com --debug
--trace --logdest /tmp/cookbook.log
Debug: Executing '/etc/puppet/cookbook.sh cookbook.example.com'
Debug: Using cached facts for cookbook.example.com
Info: Caching node for cookbook.example.com
Debug: importing '/etc/puppet/environments/production/modules/enc/
manifests/init.pp' in environment production
Debug: Automatically imported enc from enc into production
Notice: Compiled catalog for cookbook.example.com in environment
production in 0.09 seconds
Info: Caching catalog for cookbook.example.com
Debug: Configuring PuppetDB terminuses with config file /etc/puppet/
puppetdb.conf
Debug: Using cached certificate for ca
Debug: Using cached certificate for puppet
Debug: Using cached certificate_revocation_list for ca
Info: 'replace catalog' command for cookbook.example.com submitted to
PuppetDB with UUIDe2a655ca-bd81-4428-b70a-a3a76c5f15d1
{
  "metadata": {
    "api_version": 1
  },
  "data": {
    "edges": [
      {
        "target": "Class[main]",
        "source": "Stage[main]"
...
```

After compiling the catalog, Puppet will print out the catalog to the command line. The log file (`/tmp/cookbook.log`) will have a lot of information on how the catalog was compiled.

See also

- The *Auditing resources* recipe in *Chapter 6*, *Managing Resources and Files*
- The *Automatic syntax checking with Git hooks* recipe in *Chapter 2*, *Puppet Infrastructure*
- The *Generating reports* recipe in this chapter
- The *Testing your Puppet manifests with rspec-puppet* recipe in *Chapter 9*, *External Tools and the Puppet Ecosystem*

Logging command output

When you use the `exec` resources to run commands on the node, Puppet will give you an error message such as the following if a command returns a non-zero exit status:

```
Notice: /Stage[main]/Main/Exec[/bin/cat /tmp/missing]/returns: /bin/cat:
/tmp/missing: No such file or directory
Error: /bin/cat /tmp/missing returned 1 instead of one of [0]
Error: /Stage[main]/Main/Exec[/bin/cat /tmp/missing]/returns: change from
notrun to 0 failed: /bin/cat /tmp/missing returned 1 instead of one of
[0]
```

As you can see, Puppet not only reports that the command failed, but shows its output:

```
/bin/cat: /tmp/missing: No such file or directory
```

This is useful to figure out why the command didn't work, but sometimes the command actually succeeds (in that it returns a zero exit status) but still doesn't do what we wanted. In that case, how can you see the command output? You can use the `logoutput` attribute.

How to do it...

Follow these steps in order to log command output:

1. Define an `exec` resource with the `logoutput` parameter as shown in the following code snippet:

```
exec { 'exec with output':
  command   => '/bin/cat /etc/hostname',
logoutput => true,
}
```

2. Run Puppet:

```
t@mylaptop ~/puppet/manifests $ puppet apply exec.pp
Notice: Compiled catalog for mylaptop in environment production in
0.46 seconds
Notice: /Stage[main]/Main/Exec[exec with outout]/returns: mylaptop
Notice: /Stage[main]/Main/Exec[exec with outout]/returns: executed
successfully
Notice: Finished catalog run in 0.06 seconds
```

3. As you can see, even though the command succeeds, Puppet prints the output:

```
mylaptop
```

How it works...

The `logoutput` attribute has three possible settings:

- `false`: This never prints the command output
- `on_failure`: This only prints the output if the command fails (the default setting)
- `true`: This always prints the output, whether the command succeeds or fails

There's more...

You can set the default value of `logoutput` to always display command output for all `exec` resources by defining the following in your `site.pp` file:

```
Exec {
logoutput => true,
```

Resource defaults: What's this Exec syntax? It looks like an `exec` resource, but it's not. When you use `Exec` with a capital E, you're setting the resource default for exec. You may set the resource default for any resource by capitalizing the first letter of the resource type. Anywhere that Puppet see's that resource within the current scope or a nested subscope, it will apply the defaults you define.

If you never want to see the command output, whether it succeeds or fails, use:

```
logoutput => false,
```

More information is available at `https://docs.puppetlabs.com/references/latest/type.html#exec`.

Logging debug messages

It can be very helpful when debugging problems if you can print out information at a certain point in the manifest. This is a good way to tell, for example, if a variable isn't defined or has an unexpected value. Sometimes it's useful just to know that a particular piece of code has been run. Puppet's `notify` resource lets you print out such messages.

How to do it...

Define a `notify` resource in your manifest at the point you want to investigate:

```
notify { 'Got this far!': }
```

How it works...

When this resource is applied, Puppet will print out the message:

```
notice: Got this far!
```

There's more...

In addition to simple messages, we can output variables within our `notify` statements. Additionally, we can treat the `notify` calls the same as other resources, having them require or be required by other resources.

Printing out variable values

You can refer to variables in the message:

```
    notify { "operatingsystem is ${::operatingsystem}": }
```

Puppet will interpolate the values in the printout:

```
    Notice: operatingsystem is Fedora
```

The double colon (`::`) before the fact name tells Puppet that this is a variable in top scope (accessible to all classes) and not local to the class. For more about how Puppet handles variable scope, see the Puppet Labs article:

`http://docs.puppetlabs.com/guides/scope_and_puppet.html`

Resource ordering

Puppet compiles your manifests into a catalog; the order in which resources are executed on the client (node) may not be the same as the order of the resources within your source files. When you are using a `notify` resource for debugging, you should use resource chaining to ensure that the `notify resource` is executed before or after your failing resource.

For example, if the exec `failing exec` is failing, you can chain a `notify resource` to run directly before the failed exec resource as shown here:

```
notify{"failed exec on ${hostname}": }->
exec {'failing exec':
  command   => "/bin/grep ${hostname} /etc/hosts",
logoutput => true,
}
```

If you don't chain the resource or use a metaparameter such as `before` or `require`, there is no guarantee your `notify` statement will be executed near the other resources you are interested in debugging. More information on resource ordering can be found at `https://docs.puppetlabs.com/puppet/latest/reference/lang_relationships.html`.

For example, to have your `notify resource` run after `'failing exec'` in the preceding code snippet, use:

```
notify { 'Resource X has been applied':
  require => Exec['failing exec'],
}
```

Note, however, that in this case the `notify resource` will fail to execute since the exec failed. When a resource fails, all the resources that depended on that resource are skipped:

```
notify {'failed exec failed':
  require => Exec['failing exec']
}
```

When we run Puppet, we see that the `notify resource` is skipped:

```
t@mylaptop ~/puppet/manifests $ puppet apply fail.pp
...
Error: /bin/grepmylaptop /etc/hosts returned 1 instead of one of [0]
Error: /Stage[main]/Main/Exec[failing exec]/returns: change from notrun
to 0 failed: /bin/grepmylaptop /etc/hosts returned 1 instead of one
of [0]
Notice: /Stage[main]/Main/Notify[failed exec failed]: Dependency
Exec[failing exec] has failures: true
Warning: /Stage[main]/Main/Notify[failed exec failed]: Skipping because
of failed dependencies
Notice: Finished catalog run in 0.06 seconds
```

Generating reports

If you're managing a lot of machines, Puppet's reporting facility can give you some valuable information on what's actually happening out there.

How to do it...

To enable reports, just add this to a client's `puppet.conf`: within the `[main]` or `[agent]` sections:

```
report = true
```

In recent versions (greater than 3.0) of Puppet, `report = true` is the default setting.

How it works...

With reporting enabled, Puppet will generate a report file, containing data such as:

- Date and time of the run
- Total time for the run
- Log messages output during the run
- List of all the resources in the client's manifest
- Whether Puppet changed any resources, and how many
- Whether the run succeeded or failed

By default, these reports are stored on the node at `/var/lib/puppet/reports` in a directory named after the hostname, but you can specify a different destination using the `reportdir` option. You can create your own scripts to process these reports (which are in the standard YAML format). When we run puppet agent on `cookbook.example.com`, the following file is created on the master:

```
/var/lib/puppet/reports/cookbook.example.com/201411230717.yaml
```

There's more...

If you have more than one master server, you can have all your reports sent to the same server by specifying `report_server` in the `[agent]` section of `puppet.conf`.

If you just want one report, or you don't want to enable reporting all the time, you can add the `--report` switch to the command line when you run Puppet agent manually:

```
[root@cookbook ~]# puppet agent -t --report
Notice: Finished catalog run in 0.34 seconds
```

You won't see any additional output, but a report file will be generated in the `report` directory.

You can also see some overall statistics about a Puppet run by supplying the `--summarize` switch:

```
[root@cookbook ~]# puppet agent -t --report --summarize
Notice: Finished catalog run in 0.35 seconds
Changes:
            Total: 2
Events:
            Total: 2
          Success: 2
Resources:
            Total: 10
          Changed: 2
      Out of sync: 2
Time:
Filebucket: 0.00
         Schedule: 0.00
           Notify: 0.00
Config retrieval: 0.94
            Total: 0.95
         Last run: 1416727357
Version:
Config: 1416727291
           Puppet: 3.7.3
```

Other report types

Puppet can generate different types of reports with the reports option in the `[main]` or `[master]` section of `puppet.conf` on your Puppet master servers. There are several built-in report types listed at `https://docs.puppetlabs.com/references/latest/report.html`. In addition to the built-in report types, there are some community developed reports that are quite useful. The Foreman (`http://theforeman.org`), for example, provides a Foreman report type that you can enable to forward your node reports to the Foreman.

See also

- The *Auditing resources* recipe in *Chapter 6*, *Managing Resources and Files*

Producing automatic HTML documentation

As your manifests get bigger and more complex, it can be helpful to create HTML documentation for your nodes and classes using Puppet's automatic documentation tool, `puppet doc`.

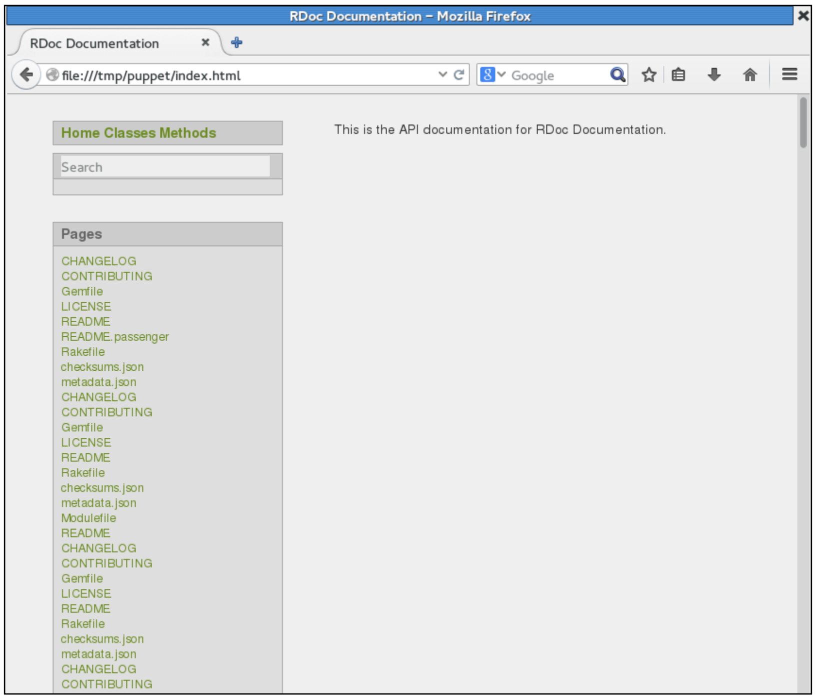
How to do it...

Follow these steps to generate HTML documentation for your manifest:

1. Run the following command:

```
t@mylaptop ~/puppet $ puppet doc --all --outputdir=/tmp/puppet
--mode rdoc --modulepath=modules/
```

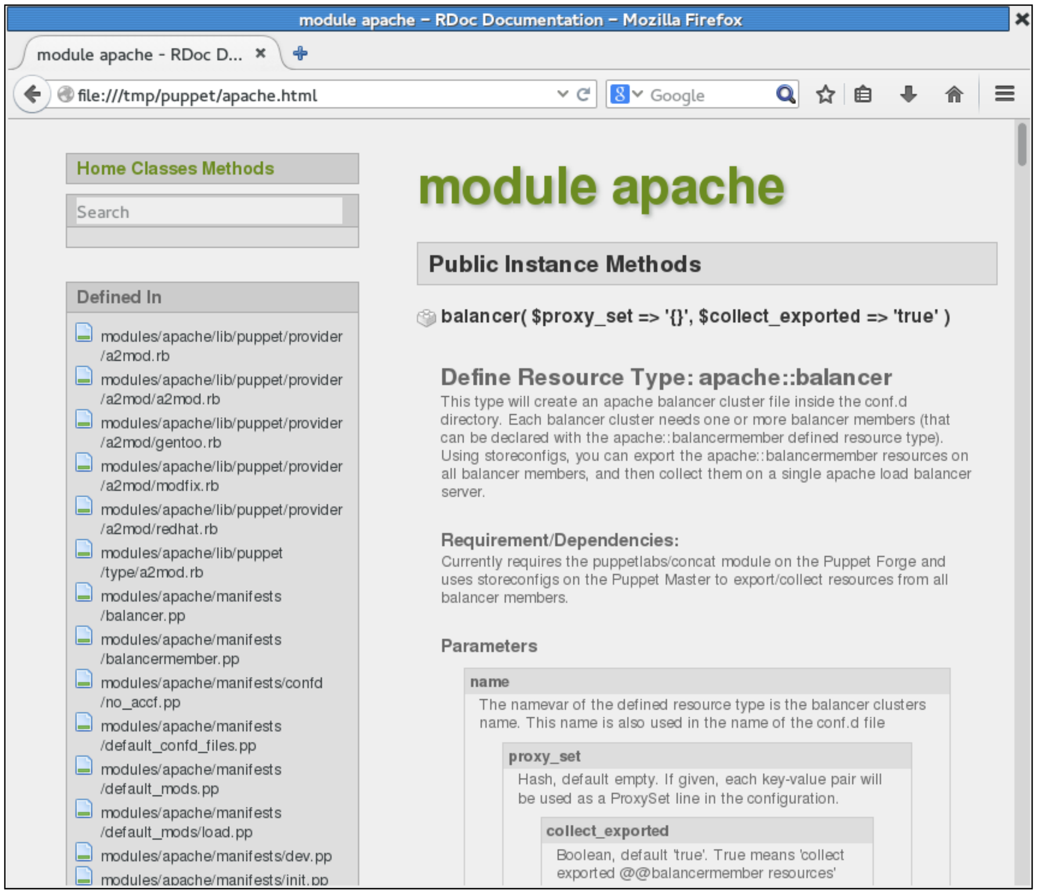
2. This will generate a set of HTML files at `/tmp/puppet`. Open the top-level `index.html` file with your web browser (`file:///tmp/puppet/index.html`), and you'll see something like the following screenshot:


RDoc Documentation – Mozilla Firefox
RDoc Documentation
file:///tmp/puppet/index.html
Google
Home Classes Methods
Search
Pages

This is the API documentation for RDoc Documentation.

CHANGELOG
CONTRIBUTING
Gemfile
LICENSE
README
README.passenger
Rakefile
checksums.json
metadata.json
CHANGELOG
CONTRIBUTING
Gemfile
LICENSE
README
Rakefile
checksums.json
metadata.json
Modulefile
README
CHANGELOG
CONTRIBUTING
Gemfile
LICENSE
README
Rakefile
checksums.json
metadata.json
CHANGELOG
CONTRIBUTING

3. Click the classes link on the left and select the Apache module, something similar to the following will be displayed:



module apache - RDoc Documentation - Mozilla Firefox

module apache - RDoc D...

file:///tmp/puppet/apache.html

Home Classes Methods

Search

Defined In

- modules/apache/lib/puppet/provider/a2mod.rb
- modules/apache/lib/puppet/provider/a2mod/a2mod.rb
- modules/apache/lib/puppet/provider/a2mod/gentoo.rb
- modules/apache/lib/puppet/provider/a2mod/modfix.rb
- modules/apache/lib/puppet/provider/a2mod/redhat.rb
- modules/apache/lib/puppet/type/a2mod.rb
- modules/apache/manifests/balancer.pp
- modules/apache/manifests/balancermember.pp
- modules/apache/manifests/confd/no_accf.pp
- modules/apache/manifests/default_confd_files.pp
- modules/apache/manifests/default_mods.pp
- modules/apache/manifests/default_mods/load.pp
- modules/apache/manifests/dev.pp
- modules/apache/manifests/init.pp

module apache

Public Instance Methods

balancer($proxy_set => '{}', $collect_exported => 'true')

Define Resource Type: apache::balancer

This type will create an apache balancer cluster file inside the conf.d directory. Each balancer cluster needs one or more balancer members (that can be declared with the apache::balancermember defined resource type). Using storeconfigs, you can export the apache::balancermember resources on all balancer members, and then collect them on a single apache load balancer server.

Requirement/Dependencies:

Currently requires the puppetlabs/concat module on the Puppet Forge and uses storeconfigs on the Puppet Master to export/collect resources from all balancer members.

Parameters

name

The namevar of the defined resource type is the balancer clusters name. This name is also used in the name of the conf.d file

proxy_set

Hash, default empty. If given, each key-value pair will be used as a ProxySet line in the configuration.

collect_exported

Boolean, default 'true'. True means 'collect exported @@balancermember resources'

How it works...

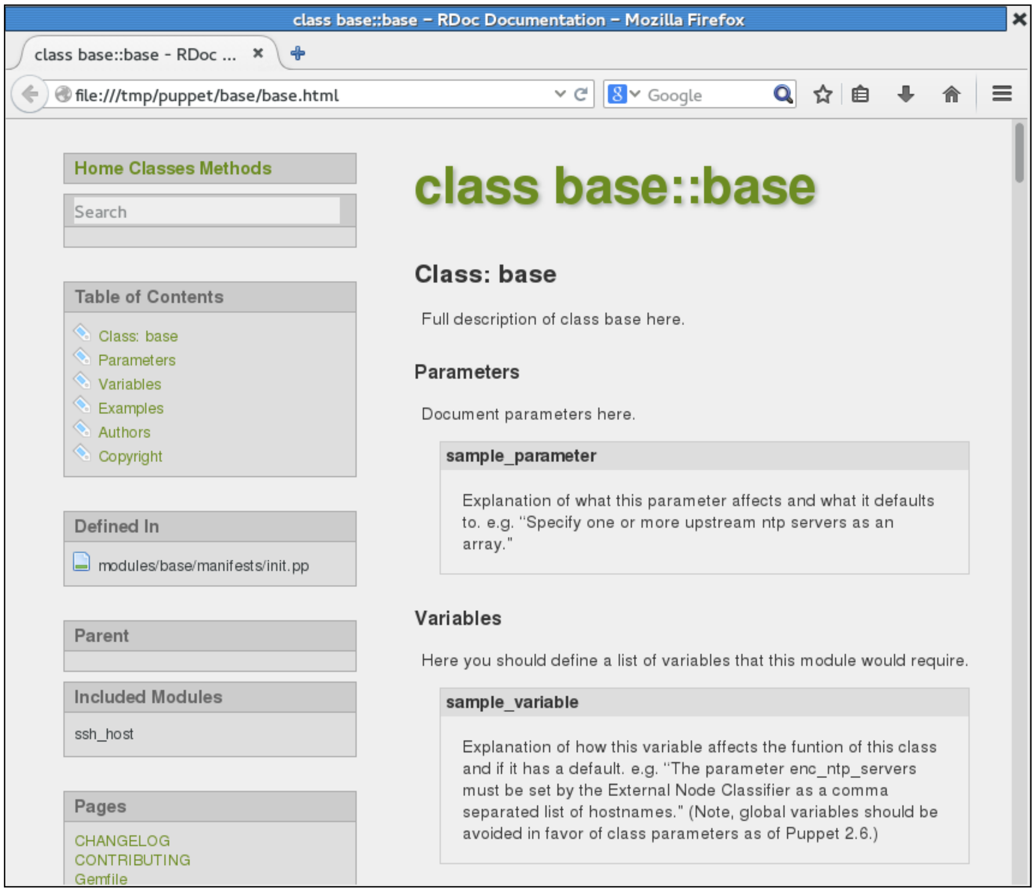
The `puppet doc` command creates a structured HTML documentation tree similar to that produced by **RDoc**, the popular Ruby documentation generator. This makes it easier to understand how different parts of the manifest relate to one another.

There's more...

The `puppet doc` command will generate basic documentation of your manifests as they stand, but you can include more useful information by adding comments to your manifest files, using the standard RDoc syntax. When we created our base class using puppet module generate, these comments were created for us:

```
# == Class: base
#
# Full description of class base here.
#
# === Parameters
#
# Document parameters here.
#
# [*sample_parameter*]
#   Explanation of what this parameter affects and what it defaults to.
#   e.g. "Specify one or more upstream ntp servers as an array."
#
# === Variables
#
# Here you should define a list of variables that this module
would require.
#
# [*sample_variable*]
#   Explanation of how this variable affects the funtion of this class
and if
#   it has a default. e.g. "The parameter enc_ntp_servers must be set
by the
#   External Node Classifier as a comma separated list of hostnames."
(Note,
#   global variables should be avoided in favor of class parameters as
#   of Puppet 2.6.)
#
# === Examples
#
#  class { base:
#    servers => [ 'pool.ntp.org', 'ntp.local.company.com' ],
#  }
#
# === Authors
#
# Author Name <author@domain.com>
#
# === Copyright
#
# Copyright 2014 Your name here, unless otherwise noted.
#
class base {
```

After generating the HTML documentation, we can see the result for the base module as shown in the following screenshot:



class base::base – RDoc Documentation – Mozilla Firefox
class base::base - RDoc ...
file:///tmp/puppet/base/base.html
Google
Home Classes Methods
Search

Table of Contents

- Class: base
- Parameters
- Variables
- Examples
- Authors
- Copyright

Defined In

modules/base/manifests/init.pp

Parent

Included Modules

ssh_host

Pages

CHANGELOG
CONTRIBUTING
Gemfile

class base::base

Class: base

Full description of class base here.

Parameters

Document parameters here.

sample_parameter

Explanation of what this parameter affects and what it defaults to. e.g. "Specify one or more upstream ntp servers as an array."

Variables

Here you should define a list of variables that this module would require.

sample_variable

Explanation of how this variable affects the funtion of this class and if it has a default. e.g. "The parameter enc_ntp_servers must be set by the External Node Classifier as a comma separated list of hostnames." (Note, global variables should be avoided in favor of class parameters as of Puppet 2.6.)

Drawing dependency graphs

Dependencies can get complicated quickly, and it's easy to end up with a circular dependency (where A depends on B, which depends on A) that will cause Puppet to complain and stop working. Fortunately, Puppet's `--graph` option makes it easy to generate a diagram of your resources and the dependencies between them, which can be a big help in fixing such problems.

Getting ready

Install the `graphviz` package to view the diagram files:

```
t@mylaptop ~ $ sudo puppet resource package graphviz ensure=installed
Notice: /Package[graphviz]/ensure: created
package { 'graphviz':
  ensure => '2.34.0-9.fc20',
}
```

How to do it...

Follow these steps to generate a dependency graph for your manifest:

1. Create the directories for a new `trifecta` module:

   ```
   ubuntu@cookbook:~/puppet$ mkdir modules/trifecta
   ubuntu@cookbook:~/puppet$ mkdir modules/trifecta/manifests
   ubuntu@cookbook:~/puppet$ mkdir modules/trifecta/files
   ```

2. Create the file `modules/trifecta/manifests/init.pp` with the following code containing a deliberate circular dependency (can you spot it?):

   ```
   class trifecta {
     package { 'ntp':
       ensure  => installed,
       require => File['/etc/ntp.conf'],
     }

     service { 'ntp':
       ensure  => running,
       require => Package['ntp'],
     }

     file { '/etc/ntp.conf':
       source  => 'puppet:///modules/trifecta/ntp.conf',
       notify  => Service['ntp'],
       require => Package['ntp'],
     }
   }
   ```

3. Create a simple `ntp.conf` file:

```
t@mylaptop~/puppet $ cd modules/trifecta/files
t@mylaptop~/puppet/modules/trifecta/files $ echo "server
127.0.0.1" >ntp.conf
```

4. Since we'll be working locally on this problem, create a `trifecta.pp` manifest that includes the broken trifecta class:

```
include trifecta
```

5. Run Puppet:

```
t@mylaptop ~/puppet/manifests $ puppet apply trifecta.pp
Notice: Compiled catalog for mylaptop in environment production in
1.32 seconds
Error: Could not apply complete catalog: Found 1 dependency cycle:
(File[/etc/ntp.conf] => Package[ntp] => File[/etc/ntp.conf])
Try the '--graph' option and opening the resulting '.dot' file in
OmniGraffle or GraphViz
```

6. Run Puppet with the `--graph` option as suggested:

```
t@mylaptop ~/puppet/manifests $ puppet apply trifecta.pp --graph
Notice: Compiled catalog for mylaptop in environment production in
1.26 seconds
Error: Could not apply complete catalog: Found 1 dependency cycle:
(File[/etc/ntp.conf] => Package[ntp] => File[/etc/ntp.conf])
Cycle graph written to /home/tuphill/.puppet/var/state/graphs/
cycles.dot.
Notice: Finished catalog run in 0.03 seconds
```

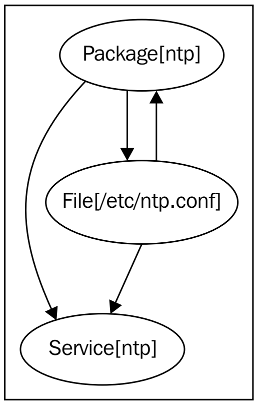
7. Check whether the graph files have been created:

```
t@mylaptop ~/puppet/manifests $ cd ~/.puppet/var/state/graphs
t@mylaptop ~/.puppet/var/state/graphs $ ls -l
total 16
-rw-rw-r--. 1 thomasthomas  121 Nov 23 23:11 cycles.dot
-rw-rw-r--. 1 thomasthomas 2885 Nov 23 23:11 expanded_
relationships.dot
-rw-rw-r--. 1 thomasthomas 1557 Nov 23 23:11 relationships.dot
-rw-rw-r--. 1 thomasthomas 1680 Nov 23 23:11 resources.dot
```

8. Create a graphic using the `dot` command as follows:

```
ubuntu@cookbook:~/puppet$ dot -Tpng -o relationships.png /var/lib/
puppet/state/graphs/relationships.dot
```

9. The graphic will look something like the this:



How it works…

When you run `puppet agent --graph` (or enable the `graph` option in `puppet.conf`), Puppet will generate three graphs in the DOT format (a graphics language):

- `resources.dot`: This shows the hierarchical structure of your classes and resources, but without dependencies
- `relationships.dot`: This shows the dependencies between resources as arrows, as shown in the preceding image
- `expanded_relationships.dot`: This is a more detailed version of the relationships graph

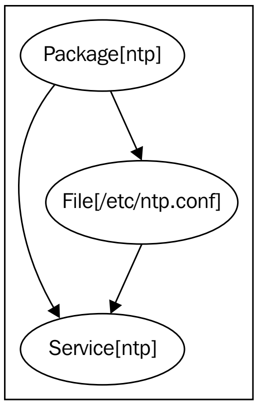
The `dot` tool (part of the `graphviz` package) will convert these to an image format such as PNG for viewing.

In the relationships graph, each resource in your manifest is shown as a balloon (known as a vertex), with arrowed lines connecting them to indicate the dependencies. You can see that in our example, the dependencies between `File['/etc/ntp.conf']` and `Package['ntp']` are bidirectional. When Puppet tries to decide where to begin applying these resources, it can start at `File['/etc/ntp.conf']` and look for what depends on `File['/etc/ntp.conf']` and end up at `Package['ntp']`. When Puppet looks for the dependencies of `Package['ntp']`, it will end up back at `File['/etc/ntp.conf']`, forming a circular path. This type of problem is known as a circular dependency problem; Puppet can't decide where to start because the two resources depend on each other.

To fix the circular dependency problem, all you need to do is remove one of the dependency lines and break the circle. The following code fixes the problem:

```
class trifecta {
  package { 'ntp':
    ensure  => installed,
  }

  service { 'ntp':
    ensure  => running,
    require => Package['ntp'],
  }

  file { '/etc/ntp.conf':
    source  => 'puppet:///modules/trifecta/ntp.conf',
    notify  => Service['ntp'],
    require => Package['ntp'],
  }
}
```

Now when we run `puppet apply` or `agent` with the `--graph` option, the resulting graph does not have any circular paths (cycles):



In this graph it is easy to see that **Package[ntp]** is the first resource to be applied, then **File[/etc/ntp.conf]**, and finally **Service[ntp]**.

[A graph such as that shown previously is known as a Directed Acyclic Graph (DAG). Reducing the resources to a DAG ensures that Puppet can calculate the shortest path of all the vertices (resources) in linear time. For more information on DAGs, look at `http://en.wikipedia.org/wiki/Directed_acyclic_graph`.]

There's more...

Resource and relationship graphs can be useful even when you don't have a bug to find. If you have a very complex network of classes and resources, for example, studying the resources graph can help you see where to simplify things. Similarly, when dependencies become too complicated to understand from reading the manifest, the graphs can be a useful form of documentation. For instance, a graph will make it readily apparent which resources have the most dependencies and which resources are required by the most other resources. Resources that are required by a large number of other resources will have numerous arrows pointing at them.

See also

- The *Using run stages* recipe in *Chapter 3*, *Writing Better Manifests*

Understanding Puppet errors

Puppet's error messages can sometimes be a little confusing. Updated and increasingly helpful error messages are one reason to upgrade your Puppet installation if you are running any version prior to Version 3.

Here are some of the most common errors you might encounter, and what to do about them.

How to do it...

Often the first step is simply to search the Web for the error message text and see what explanations you can find for the error, along with any helpful advice about fixing it. Here are some of the most common puzzling errors, with possible explanations:

`Could not retrieve file metadata for XXX: getaddrinfo: Name or service not known`

Where `XXX` is a file resource, you may have accidentally typed `puppet://modules...` in a file source instead of `puppet:///modules...` (note the triple slash):

`Could not evaluate: Could not retrieve information from environment production source(s) XXX`

The source file may not be present or may not be in the right location in the Puppet repo:

`Error: Could not set 'file' on ensure: No such file or directory XXX`

The file path may specify a parent directory (or directories) that doesn't exist. You can use separate file resources in Puppet to create these:

`change from absent to file failed: Could not set 'file on ensure: No such file or directory`

This is often caused by Puppet trying to write a file to a directory that doesn't exist. Check that the directory either exists already or is defined in Puppet, and that the file resource requires the directory (so that the directory is always created first):

`undefined method 'closed?' for nil:NilClass`

This unhelpful error message is roughly translated as *something went wrong*. It tends to be a catch-all error caused by many different problems, but you may be able to determine what is wrong from the name of the resource, the class, or the module. One trick is to add the `--debug` switch, to get more useful information:

```
[root@cookbook ~]# puppet agent -t --debug
```

If you check your Git history to see what was touched in the most recent change, this may be another way to identify what's upsetting Puppet:

`Could not parse for environment --- "--- production": Syntax error at end of file at line 1`

This can be caused by mistyping command line options, for example, if you type `puppet -verbose` instead of `puppet --verbose`. This kind of error can be hard to see:

`Duplicate definition: X is already defined in [file] at line Y; cannot redefine at [file] line Y`

This one has caused me a bit of puzzlement in the past. Puppet's complaining about a duplicate definition, and normally if you have two resources with the same name, Puppet will helpfully tell you where they are both defined. But in this case, it's indicating the same file and line number for both. How can one resource be a duplicate of itself?

The answer is, if it's a defined type (a resource created with the `define` keyword). If you create two instances of a defined type you'll also have two instances of all the resources contained within the definition, and they need to have distinct names. For example:

```
define check_process() {
  exec { 'is-process-running?':
    command => "/bin/ps ax |/bin/grep ${name} >/tmp/pslist.${name}.
txt",
  }
}

check_process { 'exim': }
check_process { 'nagios': }
```

When we run Puppet, the same error is printed twice:

```
t@mylaptop ~$ puppet apply duplicate.pp
Error: Duplicate declaration: Exec[is-process-running?] is already
declared in file duplicate.pp:4; cannot redeclare at duplicate.pp:4 on
node cookbook.example.com
Error: Duplicate declaration: Exec[is-process-running?] is already
declared in file duplicate.pp:4; cannot redeclare at duplicate.pp:4 on
node cookbook.example.com
```

Because the `exec` resource is named `is-process-running?`, if you try to create more than one instance of the definition, Puppet will refuse because the result would be two `exec` resources with the same name. The solution is to include the name of the instance (or some other unique value) in the title of each resource:

```
  exec { "is-process-${name}-running?":
    command => "/bin/ps ax |/bin/grep ${name} >/tmp/pslist.${name}.txt",
  }
```

Every resource must have a unique name, and a good way to ensure this with a definition is to interpolate the `${name}` variable in its title. Note that we switched from using single to double quotes in the resource title:

```
  "is-process-${name}-running?"
```

The double quotes are required when you want Puppet to interpolate the value of a variable into a string.

See also

- The *Generating reports* recipe in this chapter
- The *Noop: the don't change anything option* recipe in this chapter
- The *Logging debug messages* recipe in this chapter

Inspecting configuration settings

You probably know that Puppet's configuration settings are stored in `puppet.conf`, but there are many parameters, and those that aren't listed in `puppet.conf` will take a default value. How can you see the value of any configuration parameter, regardless of whether or not it's explicitly set in `puppet.conf`? The answer is to use the `puppet config print` command.

How to do it...

Run the following command. This will produce a lot of output (it may be helpful to pipe it through `less` if you'd like to browse the available configuration settings):

```
[root@cookbook ~]# puppet config print |head -25
report_serialization_format = pson
hostcsr = /var/lib/puppet/ssl/csr_cookbook.example.com.pem
filetimeout = 15
masterhttplog = /var/log/puppet/masterhttp.log
pluginsignore = .svn CVS .git
ldapclassattrs = puppetclass
certdir = /var/lib/puppet/ssl/certs
ignoreschedules = false
disable_per_environment_manifest = false
archive_files = false
hiera_config = /etc/puppet/hiera.yaml
req_bits = 4096
clientyamldir = /var/lib/puppet/client_yaml
evaltrace = false
module_working_dir = /var/lib/puppet/puppet-module
tags =
cacrl = /var/lib/puppet/ssl/ca/ca_crl.pem
manifest = /etc/puppet/manifests/site.pp
inventory_port = 8140
ignoreimport = false
dbuser = puppet
postrun_command =
document_all = false
splaylimit = 1800
certificate_expire_warning = 5184000
```

How it works...

Running `puppet config print` will output every configuration parameter and its current value (and there are lots of them).

To see the value for a specific parameter, add it as an argument to `puppet config print` command:

```
[root@cookbook ~]# puppet config print modulepath
/etc/puppet/modules:/usr/share/puppet/modules
```

See also

- The *Generating reports* recipe in this chapter

Module 2

Docker Cookbook

Over 60 hands-on recipes to efficiently work with your Docker 1.0+ environment

1
Introduction and Installation

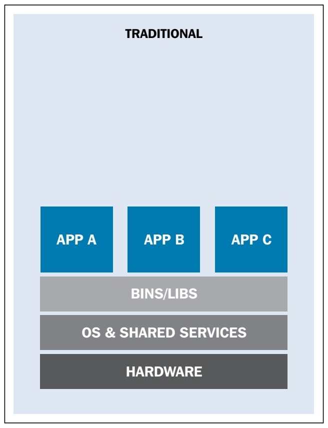
In this chapter, we will cover the following recipes:

- Verifying the requirements for Docker installation
- Installing Docker
- Pulling an image and running a container
- Adding a nonroot user to administer Docker
- Setting up the Docker host with Docker Machine
- Finding help with the Docker command line

Introduction

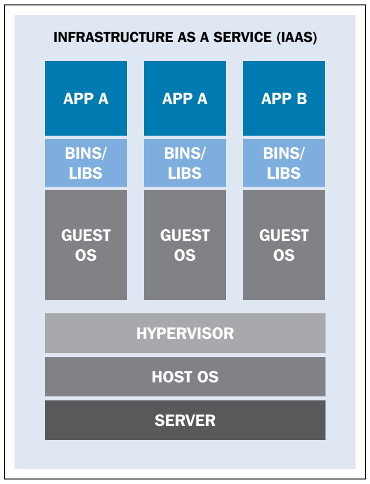
At the very start of the IT revolution, most applications were deployed directly on physical hardware, over the host OS. Because of that single user space, runtime was shared between applications. The deployment was stable, hardware-centric, and had a long maintenance cycle. It was mostly managed by an IT department and gave a lot less flexibility to developers. In such cases, hardware resources were regularly underutilized.

The following diagram depicts such a setup:



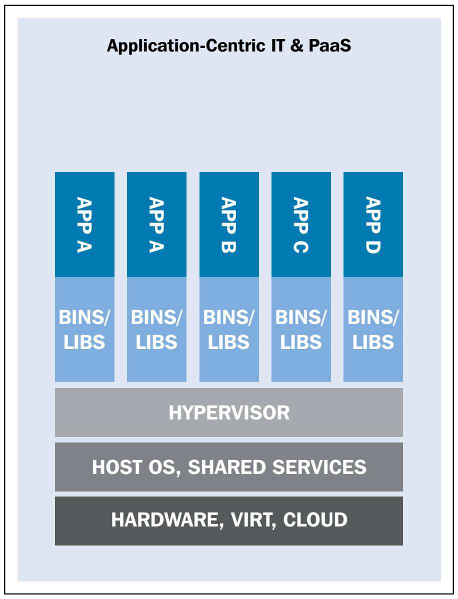
Traditional application deployment (`https://rhsummit.files.wordpress.com/2014/04/rhsummit2014-application-centric_packaging_with_docker_and_linux_containers-20140412riek7.pdf`)

To overcome the limitations set by traditional deployment, virtualization was invented. With hypervisors such as KVM, XEN, ESX, Hyper-V, and so on, we emulated the hardware for virtual machines (VMs) and deployed a guest OS on each virtual machine. VMs can have a different OS than their host; that means we are responsible for managing the patches, security, and performance of that VM. With virtualization, applications are isolated at VM level and defined by the life cycle of VMs. This gives better return on investment and higher flexibility at the cost of increased complexity and redundancy. The following diagram depicts a typical virtualized environment:



Application deployment in a virtualized environment (`https://rhsummit.files.wordpress.com/2014/04/rhsummit2014-application-centric_packaging_with_docker_and_linux_containers-20140412riek7.pdf`)

After virtualization, we are now moving towards more application-centric IT. We have removed the hypervisor layer to reduce hardware emulation and complexity. The applications are packaged with their runtime environment and are deployed using containers. OpenVZ, Solaris Zones, and LXC are a few examples of container technology. Containers are less flexible compared to VMs; for example, we cannot run Microsoft Windows on a Linux OS. Containers are also considered less secure than VMs, because with containers, everything runs on the host OS. If a container gets compromised, then it might be possible to get full access to the host OS. It can be a bit too complex to set up, manage, and automate. These are a few reasons why we have not seen the mass adoption of containers in the last few years, even though we had the technology.

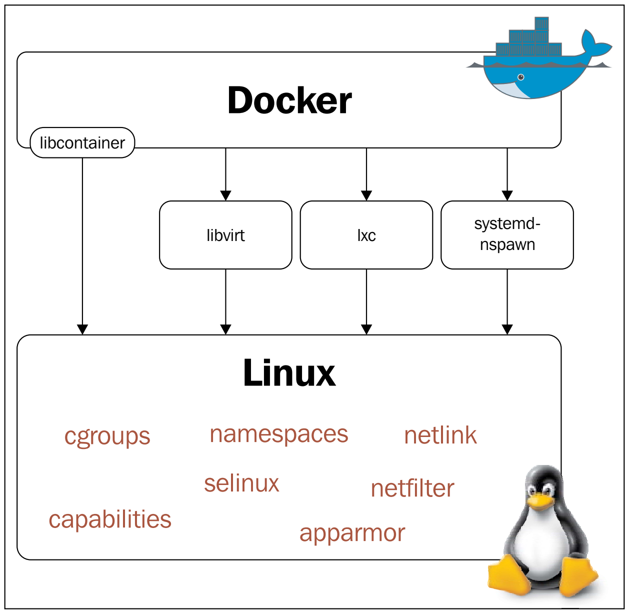


Application deployment with containers (`https://rhsummit.files.wordpress.com/2014/04/rhsummit2014-application-centric_packaging_with_docker_and_linux_containers-20140412riek7.pdf`)

With Docker, containers suddenly became first-class citizens. All big corporations such as Google, Microsoft, Red Hat, IBM, and others are now working to make containers mainstream.

Docker was started as an internal project by Solomon Hykes, who is the current CTO of Docker, Inc., at dotCloud. It was released as open source in March 2013 under the Apache 2.0 license. With dotCloud's platform as a service experience, the founders and engineers of Docker were aware of the challenges of running containers. So with Docker, they developed a standard way to manage containers.

Docker uses Linux's underlying kernel features which enable containerization. The following diagram depicts the execution drivers and kernel features used by Docker. We'll talk about execution drivers later. Let's look at some of the major kernel features that Docker uses:



The execution drivers and kernel features used by Docker (`http://blog.docker.com/wp-content/uploads/2014/03/docker-execdriver-diagram.png`)

Namespaces

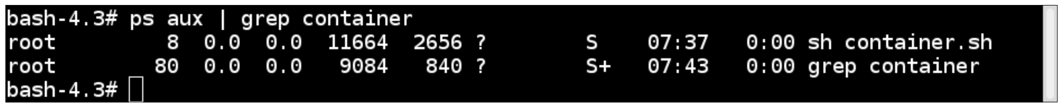
Namespaces are the building blocks of a container. There are different types of namespaces and each one of them isolates applications from each other. They are created using the clone system call. One can also attach to existing namespaces. Some of the namespaces used by Docker have been explained in the following sections.

The pid namespace

The `pid` namespace allows each container to have its own process numbering. Each `pid` forms its own process hierarchy. A parent namespace can see the children namespaces and affect them, but a child can neither see the parent namespace nor affect it.

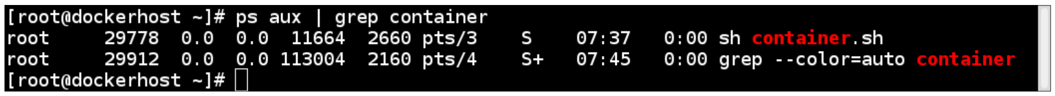
If there are two levels of hierarchy, then at the top level, we would see a process running inside the child namespace with a different PID. So, a process running in a child namespace would have two PIDs: one in the child namespace and the other in the parent namespace. For example, if we run a program on the container (`container.sh`), then we can see the corresponding program on the host as well.

On the container:



```
bash-4.3# ps aux | grep container
root         8  0.0  0.0  11664  2656 ?        S    07:37   0:00 sh container.sh
root        80  0.0  0.0   9084   840 ?        S+   07:43   0:00 grep container
bash-4.3# 
```

On the host:



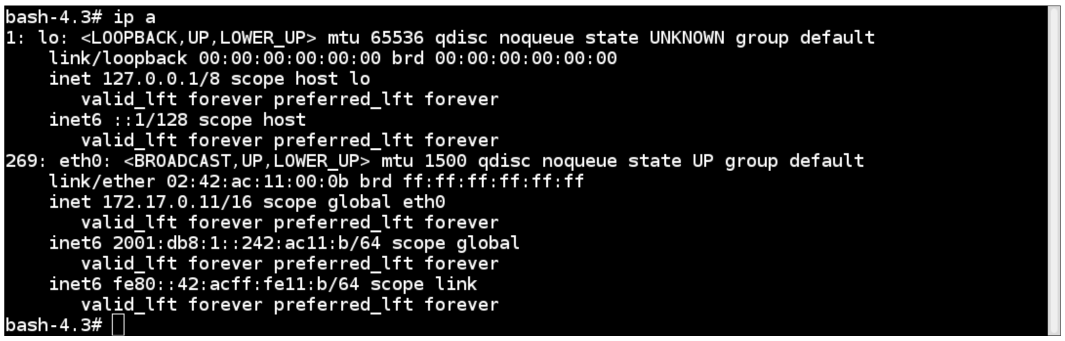
```
[root@dockerhost ~]# ps aux | grep container
root     29778  0.0  0.0  11664  2660 pts/3    S    07:37   0:00 sh container.sh
root     29912  0.0  0.0 113004  2160 pts/4    S+   07:45   0:00 grep --color=auto container
[root@dockerhost ~]# 
```

The net namespace

With the `pid` namespace, we can run the same program multiple times in different isolated environments; for example, we can run different instances of Apache on different containers. But without the `net` namespace, we would not be able to listen on port 80 on each one of them. The `net` namespace allows us to have different network interfaces on each container, which solves the problem I mentioned earlier. Loopback interfaces would be different in each container as well.

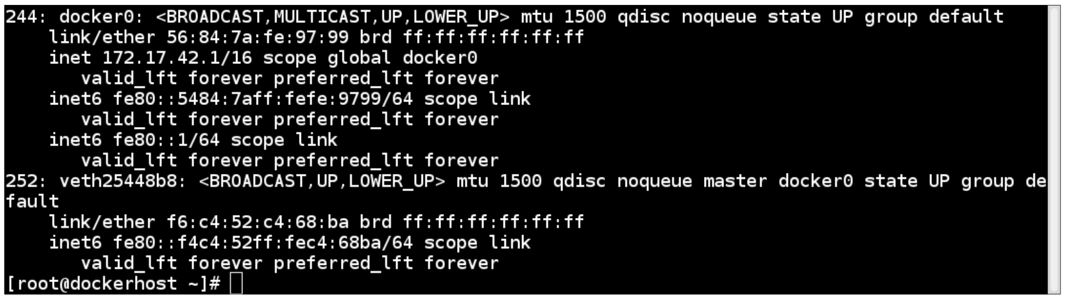
To enable networking in containers, we can create pairs of special interfaces in two different `net` namespaces and allow them to talk to each other. One end of the special interface resides inside the container and the other in the host system. Generally, the interface inside the container is named `eth0`, and in the host system, it is given a random name such as `vethcf1a`. These special interfaces are then linked through a bridge (`docker0`) on the host to enable communication between containers and route packets.

Inside the container, you would see something like the following:



```
bash-4.3# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
269: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:11:00:0b brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.11/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 2001:db8:1::242:ac11:b/64 scope global
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:b/64 scope link
       valid_lft forever preferred_lft forever
bash-4.3# 
```

And in the host, it would look like the following:



```
244: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
    inet 172.17.42.1/16 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::5484:7aff:fefe:9799/64 scope link
       valid_lft forever preferred_lft forever
    inet6 fe80::1/64 scope link
       valid_lft forever preferred_lft forever
252: veth25448b8: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP group de
fault
    link/ether f6:c4:52:c4:68:ba brd ff:ff:ff:ff:ff:ff
    inet6 fe80::f4c4:52ff:fec4:68ba/64 scope link
       valid_lft forever preferred_lft forever
[root@dockerhost ~]#
```

Also, each `net` namespace has its own routing table and firewall rules.

The ipc namespace

Inter Process Communication (**ipc**) provides semaphores, message queues, and shared memory segments. It is not widely used these days but some programs still depend on it.

If the `ipc` resource created by one container is consumed by another container, then the application running on the first container could fail. With the `ipc` namespace, processes running in one namespace cannot access resources from another namespace.

The mnt namespace

With just a chroot, one can inspect the relative paths of the system from a chrooted directory/namespace. The `mnt` namespace takes the idea of a chroot to the next level. With the `mnt` namespace, a container can have its own set of mounted filesystems and root directories. Processes in one `mnt` namespace cannot see the mounted filesystems of another `mnt` namespace.

The uts namespace

With the `uts` namespace, we can have different hostnames for each container.

The user namespace

With `user` namespace support, we can have users who have a nonzero ID on the host but can have a zero ID inside the container. This is because the `user` namespace allows per namespace mappings of users and groups IDs.

There are ways to share namespaces between the host and container and container and container. We'll see how to do that in subsequent chapters.

Cgroups

Control Groups (**cgroups**) provide resource limitations and accounting for containers. From the Linux Kernel documentation:

> *Control Groups provide a mechanism for aggregating/partitioning sets of tasks, and all their future children, into hierarchical groups with specialized behaviour.*

In simple terms, they can be compared to the `ulimit` shell command or the `setrlimit` system call. Instead of setting the resource limit to a single process, cgroups allow the limiting of resources to a group of processes.

Control groups are split into different subsystems, such as CPU, CPU sets, memory block I/O, and so on. Each subsystem can be used independently or can be grouped with others. The features that cgroups provide are:

- **Resource limiting**: For example, one cgroup can be bound to specific CPUs, so all processes in that group would run off given CPUs only
- **Prioritization**: Some groups may get a larger share of CPUs
- **Accounting**: You can measure the resource usage of different subsystems for billing
- **Control**: Freezing and restarting groups

Some of the subsystems that can be managed by cgroups are as follows:

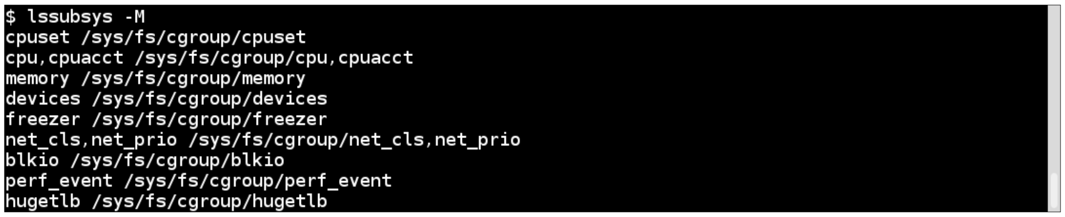
- **blkio**: It sets I/O access to and from block devices such as disk, SSD, and so on
- **Cpu**: It limits access to CPU
- **Cpuacct**: It generates CPU resource utilization
- **Cpuset**: It assigns the CPUs on a multicore system to tasks in a cgroup
- **Devices**: It devises access to a set of tasks in a cgroup
- **Freezer**: It suspends or resumes tasks in a cgroup
- **Memory**: It sets limits on memory use by tasks in a cgroup

There are multiple ways to control work with cgroups. Two of the most popular ones are accessing the cgroup virtual filesystem manually and accessing it with the `libcgroup` library. To use `libcgroup` in fedora, run the following command to install the required packages:

```
$ sudo yum install libcgroup libcgroup-tools
```

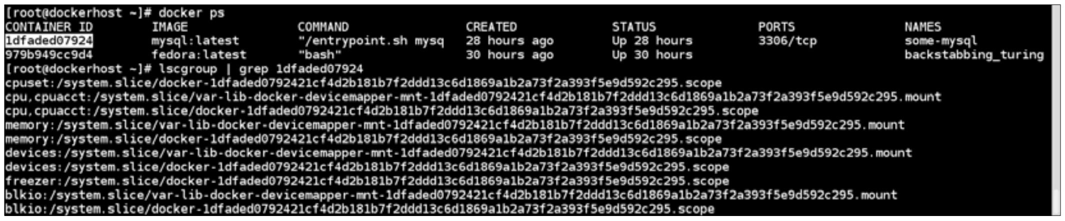
Once installed, you can get the list of subsystems and their mount point in the pseudo filesystem with the following command:

```
$ lssubsys -M
```



```
$ lssubsys -M
cpuset /sys/fs/cgroup/cpuset
cpu,cpuacct /sys/fs/cgroup/cpu,cpuacct
memory /sys/fs/cgroup/memory
devices /sys/fs/cgroup/devices
freezer /sys/fs/cgroup/freezer
net_cls,net_prio /sys/fs/cgroup/net_cls,net_prio
blkio /sys/fs/cgroup/blkio
perf_event /sys/fs/cgroup/perf_event
hugetlb /sys/fs/cgroup/hugetlb
```

Although we haven't looked at the actual commands yet, let's assume that we are running a few containers and want to get the cgroup entries for a container. To get those, we first need to get the container ID and then use the `lscgroup` command to get the cgroup entries of a container, which we can get from the following command:



```
[root@dockerhost ~]# docker ps
CONTAINER ID        IMAGE               COMMAND                CREATED             STATUS              PORTS               NAMES
1dfaded07924        mysql:latest        "/entrypoint.sh mysq   28 hours ago        Up 28 hours         3306/tcp            some-mysql
979b949cc9d4        fedora:latest       "bash"                 30 hours ago        Up 30 hours                             backstabbing_turing
[root@dockerhost ~]# lscgroup | grep 1dfaded07924
cpuset:/system.slice/docker-1dfaded0792421cf4d2b181b7f2ddd13c6d1869a1b2a73f2a393f5e9d592c295.scope
cpu,cpuacct:/system.slice/var-lib-docker-devicemapper-mnt-1dfaded0792421cf4d2b181b7f2ddd13c6d1869a1b2a73f2a393f5e9d592c295.mount
cpu,cpuacct:/system.slice/docker-1dfaded0792421cf4d2b181b7f2ddd13c6d1869a1b2a73f2a393f5e9d592c295.scope
memory:/system.slice/var-lib-docker-devicemapper-mnt-1dfaded0792421cf4d2b181b7f2ddd13c6d1869a1b2a73f2a393f5e9d592c295.mount
memory:/system.slice/docker-1dfaded0792421cf4d2b181b7f2ddd13c6d1869a1b2a73f2a393f5e9d592c295.scope
devices:/system.slice/var-lib-docker-devicemapper-mnt-1dfaded0792421cf4d2b181b7f2ddd13c6d1869a1b2a73f2a393f5e9d592c295.mount
devices:/system.slice/docker-1dfaded0792421cf4d2b181b7f2ddd13c6d1869a1b2a73f2a393f5e9d592c295.scope
freezer:/system.slice/docker-1dfaded0792421cf4d2b181b7f2ddd13c6d1869a1b2a73f2a393f5e9d592c295.scope
blkio:/system.slice/var-lib-docker-devicemapper-mnt-1dfaded0792421cf4d2b181b7f2ddd13c6d1869a1b2a73f2a393f5e9d592c295.mount
blkio:/system.slice/docker-1dfaded0792421cf4d2b181b7f2ddd13c6d1869a1b2a73f2a393f5e9d592c295.scope
```

For more details, visit `https://docs.docker.com/articles/runmetrics/`.

The Union filesystem

The Union filesystem allows the files and directories of separate filesystems, known as layers, to be transparently overlaid to create a new virtual filesystem. While starting a container, Docker overlays all the layers attached to an image and creates a read-only filesystem. On top of that, Docker creates a read/write layer which is used by the container's runtime environment. Look at the *Pulling an image and running a container* recipe of this chapter for more details. Docker can use several Union filesystem variants, including AUFS, Btrfs, vfs, and DeviceMapper.

Docker can work with different execution drivers, such as `libcontainer`, `lxc`, and `libvirt` to manage containers. The default execution driver is `libcontainer`, which comes with Docker out of the box. It can manipulate namespaces, control groups, capabilities, and so on for Docker.

Verifying the requirements for Docker installation

Docker is supported on many Linux platforms, such as RHEL, Ubuntu, Fedora, CentOS, Debian, Arch Linux, and so on. It is also supported on many cloud platforms, such as Amazon EC2, Rackspace Cloud, and Google Compute Engine. With the help of a virtual environment, Boot2Docker, it can also run on OS X and Microsoft Windows. A while back, Microsoft announced that it would add native support to Docker on its next Microsoft Windows release.

In this recipe, let's verify the requirements for Docker installation. We will check on the system with Fedora 21 installation, though the same steps should work on Ubuntu as well.

Getting ready

Log in as root on the system with Fedora 21 installed.

How to do it...

Perform the following steps:

1. Docker is not supported on 32-bit architecture. To check the architecture on your system, run the following command:

   ```
   $ uname -i
   x86_64
   ```

2. Docker is supported on kernel 3.8 or later. It has been back ported on some of the kernel 2.6, such as RHEL 6.5 and above. To check the kernel version, run the following command:

   ```
   $ uname -r
   3.18.7-200.fc21.x86_64
   ```

3. Running kernel should support an appropriate storage backend. Some of these are VFS, DeviceMapper, AUFS, Btrfs, and OverlayFS.

   Mostly, the default storage backend or driver is devicemapper, which uses the device-mapper thin provisioning module to implement layers. It should be installed by default on the majority of Linux platforms. To check for device-mapper, you can run the following command:

   ```
   $ grep device-mapper /proc/devices
   253 device-mapper
   ```

   In most distributions, AUFS would require a modified kernel.

4. Support for cgroups and namespaces are in kernel for sometime and should be enabled by default. To check for their presence, you can look at the corresponding configuration file of the kernel you are running. For example, on Fedora, I can do something like the following:

```
$ grep -i namespaces /boot/config-3.18.7-200.fc21.x86_64
CONFIG_NAMESPACES=y
$ grep -i cgroups /boot/config-3.18.7-200.fc21.x86_64
CONFIG_CGROUPS=y
```

How it works…

With the preceding commands, we verified the requirements for Docker installation.

See also

- Installation document on the Docker website at `https://docs.docker.com/installation/`

Installing Docker

As there are many distributions which support Docker, we'll just look at the installation steps on Fedora 21 in this recipe. For others, you can refer to the installation instructions mentioned in the *See also* section of this recipe. Using Docker Machine, we can set up Docker hosts on local systems, on cloud providers, and other environments very easily. We'll cover that in a different recipe.

Getting ready

Check for the prerequisites mentioned in the previous recipe.

How to do it…

1. Install Docker using yum:

```
$  yum -y install docker
```

How it works…

The preceding command will install Docker and all the packages required by it.

There's more...

The default Docker daemon configuration file is located at `/etc/sysconfig/docker`, which is used while starting the daemon. Here are some basic operations:

- To start the service:
  ```
  $ systemctl start docker
  ```
- To verify the installation:
  ```
  $ docker info
  ```
- To update the package:
  ```
  $ yum -y update docker
  ```
- To enable the service start at boot time:
  ```
  $ systemctl enable docker
  ```
- To stop the service:
  ```
  $ systemctl stop docker
  ```

See also

- The installation document is on the Docker website at `https://docs.docker.com/installation/`

Pulling an image and running a container

I am borrowing this recipe from the next chapter to introduce some concepts. Don't worry if you don't find all the explanation in this recipe. We'll cover all the topics in detail later in this chapter or in the next few chapters. For now, let's pull an image and run it. We'll also get familiar with Docker architecture and its components in this recipe.

Getting ready

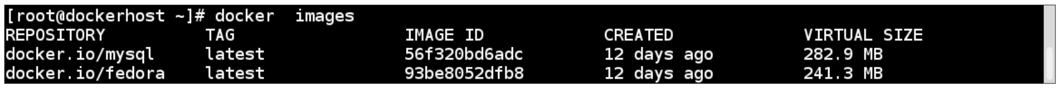
Get access to a system with Docker installed.

How to do it...

1. To pull an image, run the following command:
   ```
   $ docker pull fedora
   ```

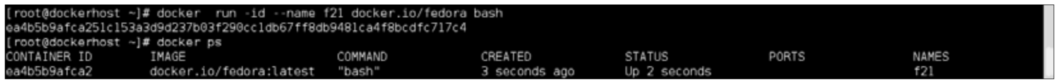
2. List the existing images by using the following command:

```
$ docker images
```



```
[root@dockerhost ~]# docker  images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
docker.io/mysql     latest              56f320bd6adc        12 days ago         282.9 MB
docker.io/fedora    latest              93be8052dfb8        12 days ago         241.3 MB
```

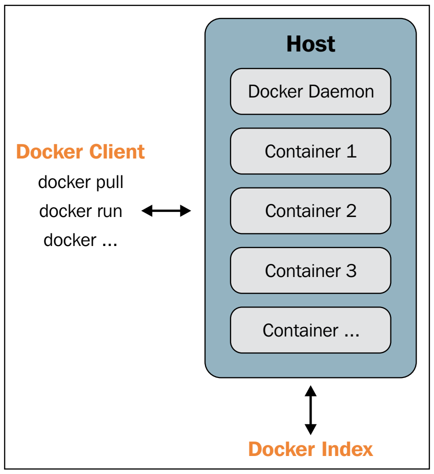
3. Create a container using the pulled image and list the containers as:



```
[root@dockerhost ~]# docker  run -id --name f21 docker.io/fedora bash
ea4b5b9afca251c153a3d9d237b03f290cc1db67ff8db9481ca4f8bcdfc717c4
[root@dockerhost ~]# docker ps
CONTAINER ID        IMAGE                     COMMAND             CREATED             STATUS              PORTS               NAMES
ea4b5b9afca2        docker.io/fedora:latest   "bash"              3 seconds ago       Up 2 seconds                            f21
```

How it works...

Docker has client-server architecture. Its binary consists of the Docker client and server daemon, and it can reside in the same host. The client can communicate via sockets or the RESTful API to either a local or remote Docker daemon. The Docker daemon builds, runs, and distributes containers. As shown in the following diagram, the Docker client sends the command to the Docker daemon running on the host machine. The Docker daemon also connects to either the public or local index to get the images requested by the client:



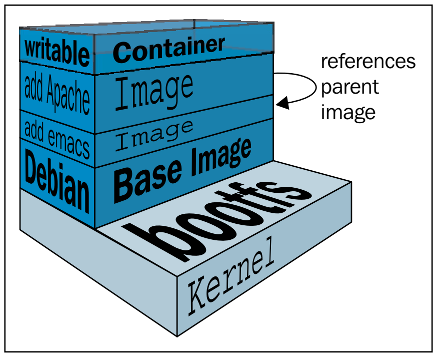
Docker client-server architecture (`https://docs.docker.com/introduction/understanding-docker/`)

So in our case, the Docker client sends a request to the daemon running on the local system, which then connects to the public Docker Index and downloads the image. Once downloaded, we can run it.

There's more...

Let's explore some keywords we encountered earlier in this recipe:

- **Images**: Docker images are read-only templates and they give us containers during runtime. There is the notion of a base image and layers on top of it. For example, we can have a base image of Fedora or Ubuntu and then we can install packages or make modifications over the base image to create a new layer. The base image and new layer can be treated as a new image. For example, in following figure, **Debian** is the base image and **emacs** and **Apache** are the two layers added on top of it. They are highly portable and can be shared easily:



Docker Image layers (`http://docs.docker.com/terms/images/docker-filesystems-multilayer.png`)

Layers are transparently laid on top of the base image to create a single coherent filesystem.

- **Registries**: A registry holds Docker images. It can be public or private from where you can download or upload images. The public Docker registry is called **Docker Hub**, which we will cover later.
- **Index**: An index manages user accounts, permissions, search, tagging, and all that nice stuff that's in the public web interface of the Docker registry.
- **Containers**: Containers are running images that are created by combining the base image and the layers on top of it. They contain everything needed to run an application. As shown in preceding diagram, a temporary layer is also added while starting the container, which would get discarded if not committed after the container is stopped and deleted. If committed, then it would create another layer.
- **Repository**: Different versions of an image can be managed by multiple tags, which are saved with different GUID. A repository is a collection of images tracked by GUIDs.

See also

- The documentation on the Docker website at `http://docs.docker.com/introduction/understanding-docker/`
- With Docker 1.6, the Docker community and Microsoft Windows released a Docker native client for Windows `http://azure.microsoft.com/blog/2015/04/16/docker-client-for-windows-is-now-available`

Adding a nonroot user to administer Docker

For ease of use, we can allow a nonroot user to administer Docker by adding them to a Docker group.

Getting ready

1. Create the Docker group if it is not there already:

   ```
   $ sudo group add docker
   ```

2. Create the user to whom you want to give permission to administer Docker:

   ```
   $ useradd dockertest
   ```

How to do it…

Run the following command to allow the newly created user to administer Docker:

```
$ sudo  gpasswd -a dockertest docker
```

How it works…

The preceding command will add a user to the Docker group. The added user will thus be able to perform all Docker operations. This can be the security risk. Visit *Chapter 9*, *Docker Security* for more details.

Setting up the Docker host with Docker Machine

Earlier this year, Docker released Orchestration tools (`https://blog.docker.com/2015/02/orchestrating-docker-with-machine-swarm-and-compose/`) and Machine, Swarm, and Compose deploy containers seamlessly. In this recipe, we'll cover Docker Machine and look at the others in later chapters. Using the Docker Machine tool (`https://github.com/docker/machine/`), you can set up Docker hosts locally on cloud with one command. It is currently in beta mode and not recommended for production use. It supports environments such as VirtualBox, OpenStack, Google, Digital Ocean, and others. For a complete list, you can visit `https://github.com/docker/machine/tree/master/drivers`. Let's use this tool and set up a host in Google Cloud.

> We will be using Docker Machine just for this recipe. Recipes mentioned in this or other chapters may or may not work on the host set up by Docker Machine.

Getting ready

Docker Machine does not appear with the default installation. You need to download it from its GitHub releases link (`https://github.com/docker/machine/releases`). Please check the latest version and distribution before downloading. As a root user, download the binary and make it executable:

```
$ curl -L
https://github.com/docker/machine/releases/download/v0.2.0/docker-
machine_linux-amd64 > /usr/local/bin/docker-machine
$ chmod a+x  /usr/local/bin/docker-machine
```

If you don't have an account on **Google Compute Engine** (**GCE**), then you can sign up for a free trial (`https://cloud.google.com/compute/docs/signup`) to try this recipe. I am assuming that you have a project on GCE and have the Google Cloud SDK installed on the system on which you downloaded Docker Machine binary. If not, then you can follow these steps:

1. Set up the Google Cloud SDK on your local system:

   ```
   $ curl https://sdk.cloud.google.com | bash
   ```

2. Create a project on GCE (`https://console.developers.google.com/project`) and get its project ID. Please note that the project name and its ID are different.
3. Go to the project home page and under the **APIs & auth** section, select **APIs**, and enable Google **Compute Engine API**.

How to do it...

1. Assign the project ID we collected to a variable, `GCE_PROJECT`:

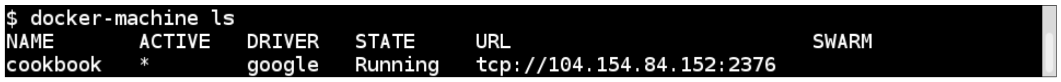
```
$ export  GCE_PROJECT="<Your Project ID>"
```

2. Run the following command and enter the code which is provided on the popped up web browser:

```
$ docker-machine  create -d google --google-
project=$GCE_PROJECT  --google-machine-type=n1-standard-2 --
google-disk-size=50 cookbook

INFO[0000] Opening auth URL in browser.

.......

......

INFO[0015] Saving token in
/home/nkhare/.docker/machine/machines/cookbook/gce_token


INFO[0015] Creating host...

INFO[0015] Generating SSH Key

INFO[0015] Creating instance.

INFO[0016] Creating firewall rule.

INFO[0020] Waiting for Instance...

INFO[0066] Waiting for SSH...

INFO[0066] Uploading SSH Key

INFO[0067] Waiting for SSH Key

INFO[0224] "cookbook" has been created and is now the active
machine.

INFO[0224] To point your Docker client at it, run this in your
shell: eval "$(docker-machine_linux-amd64 env cookbook)"
```

3. List the existing hosts managed by Docker Machine:

```
$ ./docker-machine_linux-amd64 ls
```

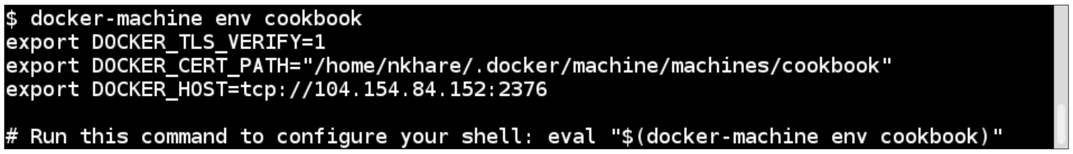


```
$ docker-machine ls
NAME       ACTIVE   DRIVER   STATE     URL                          SWARM
cookbook   *        google   Running   tcp://104.154.84.152:2376
```

You can manage multiple hosts with Docker Machine. The `*` indicates the active one.

4. To display the commands to set up the environment for the Docker client:

   ```
   $  ./docker-machine_linux-amd64 env cookbook
   ```



```
$ docker-machine env cookbook
export DOCKER_TLS_VERIFY=1
export DOCKER_CERT_PATH="/home/nkhare/.docker/machine/machines/cookbook"
export DOCKER_HOST=tcp://104.154.84.152:2376

# Run this command to configure your shell: eval "$(docker-machine env cookbook)"
```

   So, if you point the Docker client with the preceding environment variables, we would connect to the Docker daemon running on the GCE.

5. And to point the Docker client to use our newly created machine, run the following command:

   ```
   $ eval "$(./docker-machine_linux-amd64 env  cookbook)"
   ```

From now on, all the Docker commands will run on the machine we provisioned on GCE, until the preceding environment variables are set.

How it works...

Docker Machine connects to the cloud provider and sets up a Linux VM with Docker Engine. It creates a `.docker/machine/` directory under the current user's home directory to save the configuration.

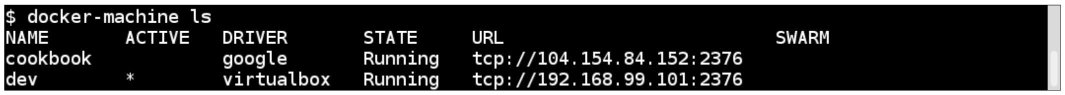
There's more...

Docker Machine provides management commands, such as `create`, `start`, `stop`, `restart`, `kill`, `remove`, `ssh`, and others to manage machines. For detailed options, look for the help option of Docker Machine:

```
$ docker-machine  -h
```

You can use the `--driver/-d` option to create choosing one of the many endpoints available for deployment. For example, to set up the environment with VirtualBox, run the following command:

```
$ docker-machine create --driver virtualbox dev
```



```
$ docker-machine ls
NAME       ACTIVE   DRIVER       STATE     URL                          SWARM
cookbook            google       Running   tcp://104.154.84.152:2376
dev        *        virtualbox   Running   tcp://192.168.99.101:2376
```

Here, `dev` is the machine name. By default, the latest deployed machine becomes primary.

See also

- Documentation on the Docker website at `https://docs.docker.com/machine/`
- Guide to setting up Docker on Google Compute Engine at `https://docs.docker.com/installation/google/`

Finding help with the Docker command line

Docker commands are well documented and can be referred to whenever needed. Lots of documentation is available online as well, but it might differ from the documentation for the Docker version you are running.

Getting ready

Install Docker on your system.

How to do it...

1. On a Linux-based system, you can use the `man` command to find help as follows:

   ```
   $ man docker
   ```

2. Subcommand-specific help can also be found with any of the following commands:

   ```
   $ man docker ps
   $ man docker-ps
   ```

How it works...

The `man` command uses the `man` pages installed by the Docker package to show help.

See also

- Documentation on the Docker website at `http://docs.docker.com/reference/commandline/cli/`

2
Working with Docker Containers

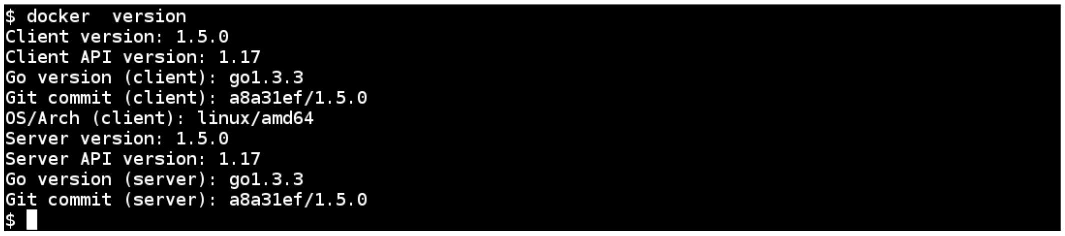
In this chapter, we will cover the following recipes:

- Listing/searching for an image
- Pulling an image
- Listing images
- Starting a container
- Listing containers
- Stopping a container
- Looking at the logs of containers
- Deleting a container
- Setting the restart policy on a container
- Getting privileged access inside a container
- Exposing a port while starting a container
- Accessing the host device inside the container
- Injecting a new process to a running container
- Returning low-level information about a container
- Labeling and filtering containers

Introduction

In the previous chapter, after installing Docker, we pulled an image and created a container from it. Docker's primary objective is running containers. In this chapter, we'll see the different operations we can do with containers such as starting, stopping, listing, deleting, and so on. This will help us to use Docker for different use cases such as testing, CI/CD, setting up PaaS, and so on, which we'll cover in later chapters. Before we start, let's verify the Docker installation by running the following command:

```
$ docker version
```



```
$ docker  version
Client version: 1.5.0
Client API version: 1.17
Go version (client): go1.3.3
Git commit (client): a8a31ef/1.5.0
OS/Arch (client): linux/amd64
Server version: 1.5.0
Server API version: 1.17
Go version (server): go1.3.3
Git commit (server): a8a31ef/1.5.0
$ 
```

This will give the Docker client and server version, as well as other details.

I am using Fedora 20/21 as my primary environment to run the recipes. They should also work with the other environment.

Listing/searching for an image

We need an image to start the container. Let's see how we can search images on the Docker registry. As we have seen in *Chapter 1*, *Introduction and Installation*, a registry holds the Docker images and it can be both public and private. By default, the search will happen on the default public registry, which is called Docker Hub and is located at `https://hub.docker.com/`.

Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client.

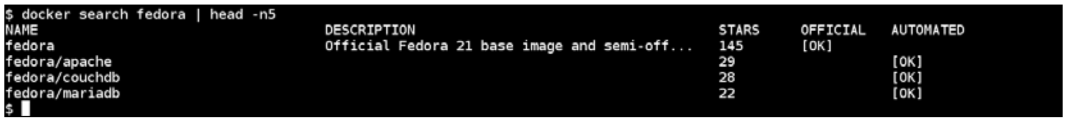
How to do it...

1. To search an image on a Docker registry, run the following command:

   ```
   docker search TERM
   ```

The following is an example to search a Fedora image:

```
$ docker search fedora |  head -n5
```



```
$ docker search fedora | head -n5
NAME                                    DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
fedora                                  Official Fedora 21 base image and semi-off...   145       [OK]
fedora/apache                                                                           29                   [OK]
fedora/couchdb                                                                          28                   [OK]
fedora/mariadb                                                                          22                   [OK]
$ 
```

The preceding screenshot lists the name, description, and number of stars awarded to the image. It also points out whether the image is official and automated or not. `STARS` signifies how many people liked the given image. The `OFFICIAL` column helps us identify whether the image is built from a trusted source or not. The `AUTOMATED` column is a way to tell whether an image is built automatically with push in GitHub or Bitbucket repositories. More details about `AUTOMATED` can be found in the next chapter.

[The convention for image name is `<user>/<name>`, but it can be anything.]

How it works...

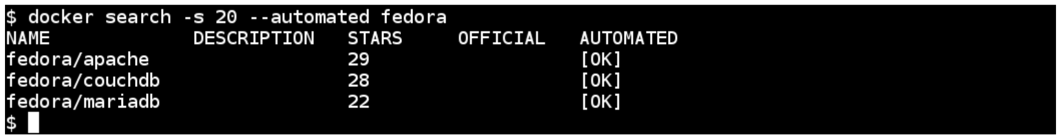
Docker searches for images on the Docker public registry, which has a repository for images at `https://registry.hub.docker.com/`.

We can configure our private index as well, which it can search for.

There's more...

- To list the images that got more than 20 stars and are automated, run the following command:

  ```
  $ docker search -s 20 --automated fedora
  ```



```
$ docker search -s 20 --automated fedora
NAME            DESCRIPTION   STARS     OFFICIAL   AUTOMATED
fedora/apache                 29                   [OK]
fedora/couchdb                28                   [OK]
fedora/mariadb                22                   [OK]
$ 
```

  In *Chapter 3*, *Working with Docker Images*, we will see how to set up automated builds.

- From Docker 1.3 onwards, the `--insecure-registry` option to Docker daemon is provided, which allows us to search/pull/commit images from an insecure registry. For more details, look at `https://docs.docker.com/reference/commandline/cli/#insecure-registries`.

- The Docker package on RHEL 7 and Fedora provides options to add and block the registry with the `--add-registry` and `--block-registry` options respectively, to have better control over the image search path. For more details, look at the following links:
    - `http://rhelblog.redhat.com/2015/04/15/understanding-the-changes-to-docker-search-and-docker-pull-in-red-hat-enterprise-linux-7-1/`
    - `https://github.com/docker/docker/pull/10411`

See also

- For help with the Docker search, run the following command:

  ```
  $ docker search --help
  ```

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#search`

Pulling an image

After searching the image, we can pull it to the system by running the Docker daemon. Let's see how we can do that.

Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client.

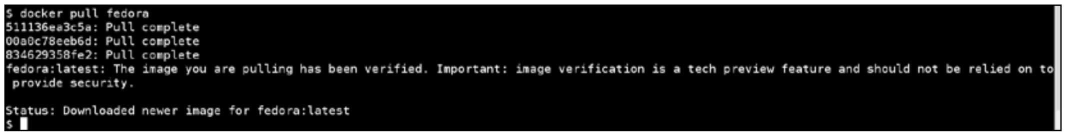
How to do it...

1. To pull an image on the Docker registry, run the following command:

   ```
   docker pull NAME[:TAG]
   ```

The following is an example to pull the Fedora image:

```
$ docker pull fedora
```



```
$ docker pull fedora
511136ea3c5a: Pull complete
00a0c78eeb6d: Pull complete
834629358fe2: Pull complete
fedora:latest: The image you are pulling has been verified. Important: image verification is a tech preview feature and should not be relied on to
 provide security.

Status: Downloaded newer image for fedora:latest
$ 
```

How it works...

The `pull` command downloads all layers from the Docker registry, which are required to create that image locally. We will see details about layers in the next chapter.

There's more...

- Image tags group images of the same type. For example, CentOS can have images with tags such as `centos5`, `centos6`, and so on. For example, to pull an image with the specific tag, run the following command:

  ```
  $ docker pull centos:centos7
  ```

- By default, the image with latest tag gets pulled. To pull all images corresponding to all tags, use the following command:

  ```
  $ docker pull --all-tags centos
  ```

- With Docker 1.6 (`https://blog.docker.com/2015/04/docker-release-1-6/`), we can build and refer to images by a new content-addressable identifier called a `digest`. It is a very useful feature when we want to work with a specific image, rather than tags. To pull an image with a specific digest, we can consider the following syntax:

  ```
  $ docker pull  <image>@sha256:<digest>
  ```

  Here is an example of a command:

  ```
  $ docker pull debian@sha256:cbbf2f9a99b47fc460d422812b6a5adff7dfee
  951d8fa2e4a98caa0382cfbdbf
  ```

  Digest is supported only with the Docker registry v2.

- Once an image gets pulled, it resides on local cache (storage), so subsequent pulls will be very fast. This feature plays a very important role in building Docker layered images.

See also

- Look at the `help` option of Docker `pull`:

  ```
  $ docker pull --help
  ```

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#pull`

Listing images

We can list the images available on the system running the Docker daemon. These images might have been pulled from the registry, imported through the `docker` command, or created through Docker files.

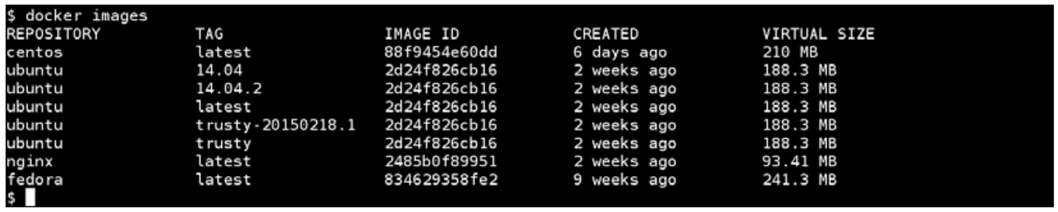
Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client.

How to do it...

1. Run the following command to list the images:

   `$ docker images`



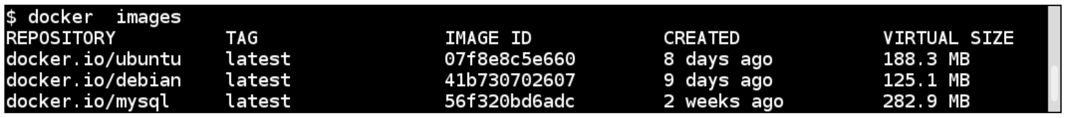
```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
centos              latest              88f9454e60dd        6 days ago          210 MB
ubuntu              14.04               2d24f826cb16        2 weeks ago         188.3 MB
ubuntu              14.04.2             2d24f826cb16        2 weeks ago         188.3 MB
ubuntu              latest              2d24f826cb16        2 weeks ago         188.3 MB
ubuntu              trusty-20150218.1   2d24f826cb16        2 weeks ago         188.3 MB
ubuntu              trusty              2d24f826cb16        2 weeks ago         188.3 MB
nginx               latest              2485b0f89951        2 weeks ago         93.41 MB
fedora              latest              834629358fe2        9 weeks ago         241.3 MB
$ 
```

How it works...

The Docker client talks to the Docker server and gets the list of images at the server end.

There's more...

- All the images with the same name but different tags get downloaded. The interesting thing to note here is that they have the same name but different tags. Also, there are two different tags for the same `IMAGE ID`, which is `2d24f826cb16`.
- You might see a different output for `REPOSITORY`, as shown in the following screenshot, with the latest Docker packages.



```
$ docker  images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
docker.io/ubuntu    latest              07f8e8c5e660        8 days ago          188.3 MB
docker.io/debian    latest              41b730702607        9 days ago          125.1 MB
docker.io/mysql     latest              56f320bd6adc        2 weeks ago         282.9 MB
```

This is because the image listing prints the Docker registry hostname as well. As shown in the preceding screenshot, `docker.io` is the registry hostname.

See also

- Look at the `help` option of `docker images`:

  ```
  $ docker images --help
  ```

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#images`

Starting a container

Once we have images, we can use them to start the containers. In this recipe, we will start a container with the `fedora:latest` image and see what all things happen behind the scene.

Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client.

How to do it...

1. The syntax used to start a container is as follows:

   ```
   docker run [ OPTIONS ]  IMAGE[:TAG]  [COMMAND]  [ARG...]
   ```

Here is an example of a command:

```
$ docker run -i -t --name=f21 fedora /bin/bash
```

By default, Docker picks the image with the latest tag:

- The `-i` option starts the container in the interactive mode
- The `-t` option allocates a `pseudo-tty` and attaches it to the standard input

So, with the preceding command, we start a container from the `fedora:latest` image, attach `pseudo-tty`, name it `f21`, and run the `/bin/bash` command. If the name is not specified, then a random string will be assigned as the name.

Also, if the image is not available locally, then it will get downloaded from the registry first and then run. Docker will run the `search` and `pull` commands before running the `run` command.

How it works…

Under the hood, Docker:

- Will merge all the layers that make that image using UnionFS.
- Allocates a unique ID to a container, which is referred to as Container ID.
- Allocates a filesystem and mounts a read/write layer for the container. Any changes on this layer will be temporary and will be discarded if they are not committed.
- Allocates a network/bridge interface.
- Assigns an IP address to the container.
- Executes the process specified by the user.

Also, with the default Docker configuration, it creates a directory with the container's ID inside `/var/lib/docker/containers`, which has the container's specific information such as hostname, configuration details, logs, and `/etc/hosts`.

There's more…

- To exit from the container, press *Ctrl* + *D* or type `exit`. It is similar to exiting from a shell but this will stop the container.
- The `run` command creates and starts the container. With Docker 1.3 or later, it is possible to just create the container using the `create` command and run it later using the `start` command, as shown in the following example:

  ```
  $ ID=$(docker create -t -i fedora bash)
  $ docker start -a -i $ID
  ```

- The container can be started in the background and then we can attach to it whenever needed. We need to use the `-d` option to start the container in the background:

  ```
  $ docker run -d -i -t fedora /bin/bash
  0df95cc49e258b74be713c31d5a28b9d590906ed9d6e1a2dc756 72aa48f28c4f
  ```

  The preceding command returns the container ID of the container to which we can attach later, as follows:

  ```
  $ ID='docker run -d -t -i fedora /bin/bash'
  $ docker attach $ID
  ```

In the preceding case, we chose `/bin/bash` to run inside the container. If we attach to the container, we will get an interactive shell. We can run a noninteractive process and run it in the background to make a daemonized container like this:

```
$ docker run -d  fedora /bin/bash -c  "while [ 1 ]; do echo
hello docker ; sleep 1; done"
```

- To remove the container after it exits, start the container with the `--rm` option, as follows:

  ```
  $ docker run --rm fedora date
  ```

  As soon as the `date` command exits, the container will be removed.

- The `--read-only` option of the `run` command will mount the root filesystem in the `read-only` mode:

  ```
  $ docker run --read-only -d -i -t fedora /bin/bash
  ```

  Remember that this option just makes sure that we cannot modify anything on the root filesystem, but we are writing on volumes, which we'll cover later in the book. This option is very useful when we don't want users to accidentally write content inside the container, which gets lost if the container is not committed or copied out on non-ephemeral storage such as volumes.

- You can also set custom labels to containers, which can be used to group the containers based on labels. Take a look at the *Labeling and filtering containers* recipe in this chapter for more details.

A container can be referred in three ways: by name, by container ID (0df95cc49e258b74be713c31d5a28b9d590906ed9d6e1a2dc75672aa48f28c4f), and by short container ID (0df95cc49e25)

See also

- Look at the `help` option of `docker run`:

  ```
  $ docker run --help
  ```

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#run`
- The Docker 1.3 release announcement `http://blog.docker.com/2014/10/docker-1-3-signed-images-process-injection-security-options-mac-shared-directories/`

Listing containers

We can list both running and stopped containers.

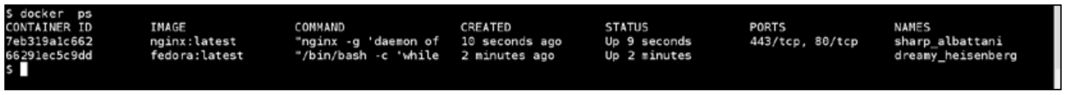
Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client. You will also need a few running and/or stopped containers.

How to do it…

1. To list the containers, run the following command:

```
docker ps [ OPTIONS ]
```



```
$ docker  ps
CONTAINER ID        IMAGE               COMMAND              CREATED              STATUS              PORTS               NAMES
7eb319a1c662        nginx:latest        "nginx -g 'daemon of  10 seconds ago      Up 9 seconds        443/tcp, 80/tcp     sharp_albattani
66291ec5c9dd        fedora:latest       "/bin/bash -c 'while  2 minutes ago       Up 2 minutes                            dreamy_heisenberg
$ 
```

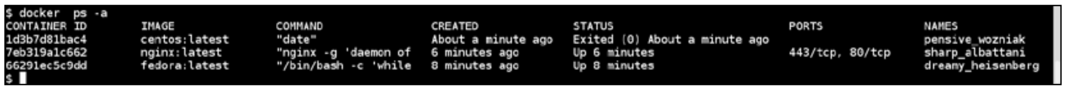
How it works…

The Docker daemon can look at the metadata associated with the containers and list them down. By default, the command returns:

- The container ID
- The image from which it got created
- The command that was run after starting the container
- The details about when it got created
- The current status
- The ports that are exposed from the container
- The name of the container

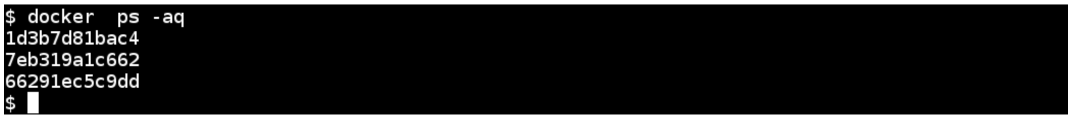
There's more…

- To list both running and stopped containers, use the `-a` option as follows:



```
$ docker  ps -a
CONTAINER ID        IMAGE               COMMAND              CREATED              STATUS                          PORTS               NAMES
1d3b7d81bac4        centos:latest       "date"               About a minute ago   Exited (0) About a minute ago                       pensive_wozniak
7eb319a1c662        nginx:latest        "nginx -g 'daemon of  6 minutes ago       Up 6 minutes                    443/tcp, 80/tcp     sharp_albattani
66291ec5c9dd        fedora:latest       "/bin/bash -c 'while  8 minutes ago       Up 8 minutes                                        dreamy_heisenberg
$ 
```

- To return just the container IDs of all the containers, use the `-aq` option as follows:



```
$ docker  ps -aq
1d3b7d81bac4
7eb319a1c662
66291ec5c9dd
$ 
```

- To show the last created container, including the non-running container, run the following command:

  `$ docker ps -l`

- Using the `--filter/-f` option to `ps` we can list containers with specific labels. Look at the *Labeling and filtering containers* recipe in this chapter for more details.

See also

Look at the `man` page of `docker ps` to see more options:

- Look at the `help` option of `docker ps`:

  `$ docker ps --help`

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#ps`

Looking at the logs of containers

If the container emits logs or output on `STDOUT/STDERR`, then we can get them without logging into the container.

Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client. You will also need a running container, which emits logs/output on `STDOUT`.

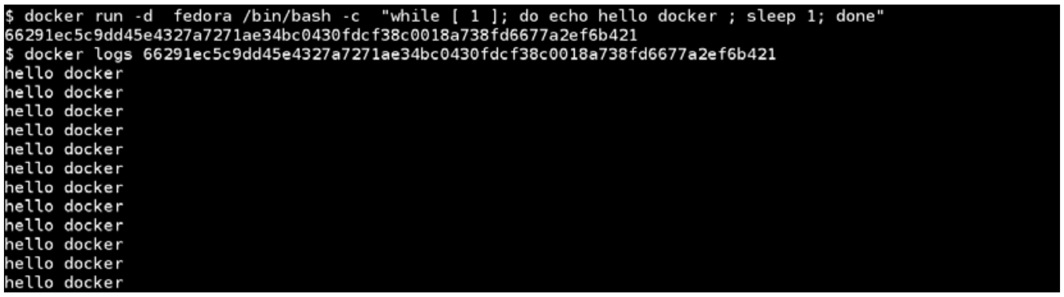
How to do it...

1. To get logs from the container, run the following command:

   ```
   docker logs [-f|--follow[=false]] [-t|--timestamps[=false]]
   CONTAINER
   ```

2. Let's take the example from the earlier section of running a daemonized container and look at the logs:

    ```
    $ docker run -d  fedora /bin/bash -c  "while [ 1 ]; do echo
    hello docker ; sleep 1; done"
    ```



```
$ docker run -d  fedora /bin/bash -c  "while [ 1 ]; do echo hello docker ; sleep 1; done"
66291ec5c9dd45e4327a7271ae34bc0430fdcf38c0018a738fd6677a2ef6b421
$ docker logs 66291ec5c9dd45e4327a7271ae34bc0430fdcf38c0018a738fd6677a2ef6b421
hello docker
hello docker
hello docker
hello docker
hello docker
hello docker
hello docker
hello docker
hello docker
hello docker
hello docker
hello docker
```

How it works...

Docker will look at the container's specific log file from `/var/lib/docker/containers/<Container ID>` and show the result.

There's more...

With the `-t` option, we can get the timestamp with each log line and with `-f` we can get tailf like behavior.

See also

- Look at `help` option of `docker logs`:

    ```
    $ docker logs --help
    ```

- Documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#logs`

Stopping a container

We can stop one or more containers at once. In this recipe, we will first start a container and then stop it.

Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client. You will also need one or more running containers.

How to do it...

1. To stop the container, run the following command:

   ```
   docker stop [-t|--time[=10]] CONTAINER [CONTAINER...]
   ```

2. If you already have a running container, then you can go ahead and stop it; if not, we can create one and then stop it as follows:

   ```
   $ ID='docker run -d -i fedora /bin/bash'
   $ docker stop $ID
   ```

How it works...

This will save the state of the container and stop it. It can be started again, if needed.

There's more...

- To stop a container after waiting for some time, use the `--time/-t` option.
- To stop all the running containers run the following command:

  ```
  $ docker stop 'docker ps -q'
  ```

See also

- Look at `help` option of `docker stop`:

  ```
  $ docker stop --help
  ```

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#stop`

Deleting a container

We can delete a container permanently, but before that we have to stop the container or use the force option. In this recipe, we'll start, stop, and delete a container.

Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client. You will also need some containers in a stopped or running state to delete them.

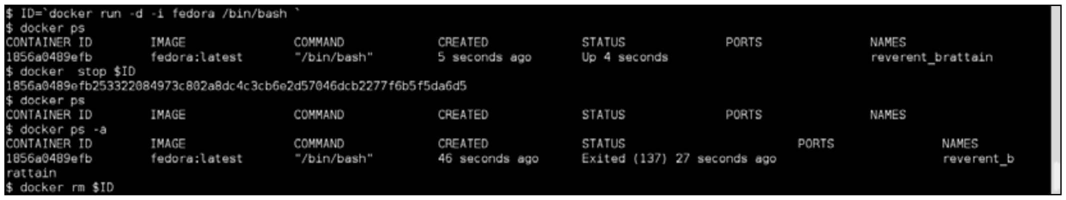
How to do it...

1. Use the following command:

```
$ docker rm [ OPTIONS ] CONTAINER [ CONTAINER ]
```

2. Let's first start a container, stop it, and then delete it using the following commands:

```
$ ID='docker run -d -i fedora /bin/bash '
$ docker stop $ID
$ docker rm $ID
```



```
$ ID=`docker run -d -i fedora /bin/bash `
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
1856a0489efb        fedora:latest       "/bin/bash"         5 seconds ago       Up 4 seconds                            reverent_brattain
$ docker  stop $ID
1856a0489efb253322084973c802a8dc4c3cb6e2d57046dcb2277f6b5f5da6d5
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                       PORTS               NAMES
1856a0489efb        fedora:latest       "/bin/bash"         46 seconds ago      Exited (137) 27 seconds ago                      reverent_b
rattain
$ docker rm $ID
```

As we can see from the preceding screenshot, the container did not show up, which just entered the `docker ps` command after stopping it. We had to provide the `-a` option to list it. After the container is stopped, we can delete it.

There's more...

- To forcefully delete a container without an intermediate stop, use the `-f` option.
- To delete all the containers, we first need to stop all the running containers and then remove them. Be careful before running the commands as these will delete both the running and the stopped containers:

```
$ docker stop 'docker ps -q'
$ docker rm 'docker ps -aq'
```

- There are options to remove a specified link and volumes associated with the container, which we will explore later.

How it works...

The Docker daemon will remove the read/write layer, which was created while starting the container.

See also

- Look at the `help` option of `docker rm`

  ```
  $ docker rm --help
  ```

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#rm`

Setting the restart policy on a container

Before Docker 1.2, there used to be an option to restart the container. With the release of Docker 1.2, it has been added with the `run` command with flags to specify the restart policy. With this policy, we can configure containers to start at boot time. This option is also very useful when a container dies accidentally.

Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client.

How to do it…

You can set the restart policy using the following syntax:

```
$ docker run --restart=POLICY [ OPTIONS ]  IMAGE[:TAG]  [COMMAND]
[ARG...]
```

Here is an example of a command:

```
$ docker run --restart=always -d -i -t fedora /bin/bash
```

There are three restart policies to choose from:

- `no`: This does not start the container if it dies
- `on-failure`: This restarts the container if it fails with nonzero exit code
- `always`: This always restarts the container without worrying about the return code

There's more…

You can also give an optional restart count with the `on-failure` policy as follows:

```
$ docker run --restart=on-failure:3 -d -i -t fedora /bin/bash
```

The preceding command will only restart the container three times, if any failure occurs.

See also

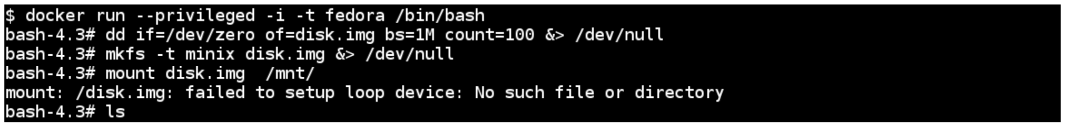
- Look at the `help` option of `docker run`:

  `$ docker run --help`

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#run`.
- If a restart does not suit your requirements, then use `systemd` (`http://freedesktop.org/wiki/Software/systemd/`) for solutions to automatically restart the container on failure. For more information, visit `https://docs.docker.com/articles/host_integration/`.

Getting privileged access inside a container

Linux divides the privileges traditionally associated with superuser into distinct units, known as capabilities (run `man capabilities` on a Linux-based system), which can be independently enabled and disabled. For example, the `net_bind_service` capability allows nonuser processes to bind the port below 1,024. By default, Docker starts containers with limited capabilities. With privileged access inside the container, we give more capabilities to perform operations normally done by root. For example, let's try to create a loopback device while mounting a disk image.



```
$ docker run --privileged -i -t fedora /bin/bash
bash-4.3# dd if=/dev/zero of=disk.img bs=1M count=100 &> /dev/null
bash-4.3# mkfs -t minix disk.img &> /dev/null
bash-4.3# mount disk.img  /mnt/
mount: /disk.img: failed to setup loop device: No such file or directory
bash-4.3# ls
```

Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client.

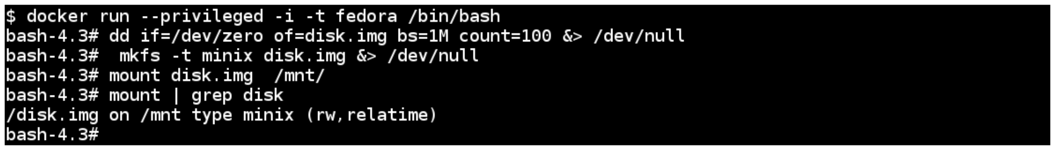
How to do it...

1. To use the `privileged` mode, use the following command:

   `$ docker run --privileged [ OPTIONS ]  IMAGE[:TAG]  [COMMAND] [ARG...]`

2. Now let's try the preceding example with the privileged access:

   ```
   $ docker run  --privileged  -i -t fedora /bin/bash
   ```



```
$ docker run --privileged -i -t fedora /bin/bash
bash-4.3# dd if=/dev/zero of=disk.img bs=1M count=100 &> /dev/null
bash-4.3#  mkfs -t minix disk.img &> /dev/null
bash-4.3# mount disk.img  /mnt/
bash-4.3# mount | grep disk
/disk.img on /mnt type minix (rw,relatime)
bash-4.3#
```

How it works...

By providing almost all capabilities inside the container.

There's more...

This mode causes security risks as containers can get root-level access on the Docker host. With Docker 1.2 or new, two new flags `--cap-add` and `--cap-del` have been added to give fine-grained control inside a container. For example, to prevent any `chown` inside the container, use the following command:

```
$ docker run --cap-drop=CHOWN [ OPTIONS ]  IMAGE[:TAG]  [COMMAND]
[ARG...]
```

Look at *Chapter 9*, *Docker Security*, for more details.

See also

- Look at the `help` option of `docker run`:

  ```
  $ docker run --help
  ```

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#run`
- The Docker 1.2 release announcement `http://blog.docker.com/2014/08/announcing-docker-1-2-0/`

Exposing a port while starting a container

There are a number of ways by which ports on the container can be exposed. One of them is through the `run` command, which we will cover in this chapter. The other ways are through the Docker file and the `--link` command. We will explore them in the other chapters.

Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client.

How to do it...

1. The syntax to expose a port is as follows:

```
$ docker run --expose=PORT [ OPTIONS ]  IMAGE[:TAG]  [COMMAND]
[ARG...]
```

For example, to expose port 22 while starting a container, run the following command:

```
$ docker run --expose=22 -i -t fedora /bin/bash
```

There's more...

There are multiple ways to expose the ports for a container. For now, we will see how we can expose the port while starting the container. We'll look other options to expose the ports in later chapters.

See also

- Look at the `help` option of `docker run`:

  ```
  $ docker run --help
  ```

- Documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#run`

Accessing the host device inside the container

From Docker 1.2 onwards, we can give access of the host device to a container with the `--device` option to the `run` command. Earlier, one has bind mount it with the `-v` option and that had to be done with the `--privileged` option.

Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client. You will also need a device to pass to the container.

How to do it...

1. You can give access of a host device to the container using the following syntax:

   ```
   $ docker run --device=<Host Device>:<Container Device
   Mapping>:<Permissions>   [ OPTIONS ]  IMAGE[:TAG]  [COMMAND]
   [ARG...]
   ```

Here is an example of a command:

```
$ docker run --device=/dev/sdc:/dev/xvdc -i -t fedora /bin/bash
```

How it works...

The preceding command will access `/dev/sdc` inside the container.

See also

- Look at the `help` option of `docker run`:

  ```
  $ docker run --help
  ```

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#run`

Injecting a new process to a running container

While doing development and debugging, we might want to look inside the already running container. There are a few utilities, such as `nsenter` (`https://github.com/jpetazzo/nsenter`), that allow us to enter into the namespace of the container to inspect it. With the `exec` option, which was added in Docker 1.3, we can inject a new process inside a running container.

Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client. You might also need a running container to inject a process in.

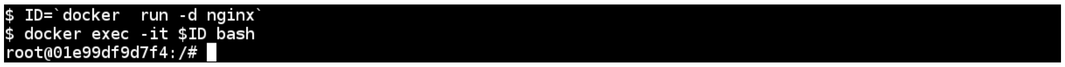
How to do it...

1. You can inject a process inside a running container with the following command:

```
 $ docker exec [-d|--detach[=false]] [--help] [-i|--
interactive[=false]] [-t|--tty[=false]] CONTAINER COMMAND
[ARG...]
```

2. Let's start an `nginx` container and then inject `bash` into that:

```
$ ID='docker run -d nginx'
$ docker run -it $ID bash
```



```
$ ID=`docker  run -d nginx`
$ docker exec -it $ID bash
root@01e99df9d7f4:/# 
```

How it works...

The `exec` command enters into the namespace of the container and starts the new process.

See also

- Look at `help` option of Docker inspect:

  ```
  $ docker exec --help
  ```

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#exec`

Returning low-level information about a container

While doing the debugging, automation, and so on, we will need the container configuration details. Docker provides the `inspect` command to get those easily.

Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client.

How to do it...

1. To inspect a container/image, run the following command:

   ```
   $ docker inspect [-f|--format="" CONTAINER|IMAGE
   [CONTAINER|IMAGE...]
   ```

2. We'll start a container and then inspect it:

   ```
   $ ID='docker run -d -i fedora /bin/bash'
   $ docker inspect $ID
   [{
       "Args": [],
       "Config": {
           "AttachStderr": false,
           "AttachStdin": false,
           "AttachStdout": false,
           "Cmd": [
               "/bin/bash"
           ],
       .........
       .........
   }]
   ```

How it works...

Docker will look into the metadata and configuration for the given image or container and present it.

There's more...

With the `-f | --format` option we can use the Go (programming language) template to get the specific information. The following command will give us an IP address of the container:

```
$ docker inspect --format='{{.NetworkSettings.IPAddress}}'  $ID
172.17.0.2
```

See also

- Look at the `help` option of `docker inspect`:

  ```
  $ docker inspect --help
  ```

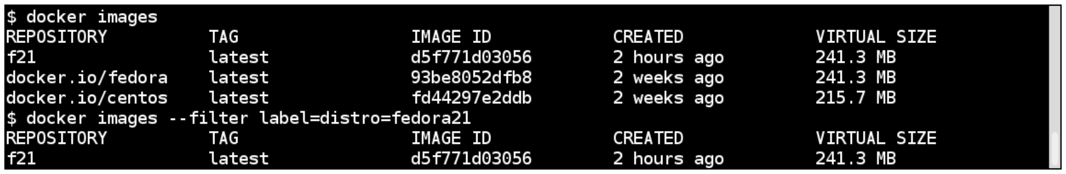
- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#inspect`

Labeling and filtering containers

With Docker 1.6, a feature has been added to label containers and images, through which we can attach arbitrary key-value metadata to them. You can think of them as environment variables, which are not available to running applications inside containers but they are available to programs (Docker CLI) that are managing images and containers. Labels attached to images also get applied to containers started via them. We can also attach labels to containers while starting them.

Docker also provides filters to containers, images, and events (`https://docs.docker.com/reference/commandline/cli/#filtering`), which we can use in conjunction with labels to narrow down our searches.

For this recipe, let's assume that we have an image with the label, `distro=fedora21`. In the next chapter, we will see how to assign a label to an image.



```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
f21                 latest              d5f771d03056        2 hours ago         241.3 MB
docker.io/fedora    latest              93be8052dfb8        2 weeks ago         241.3 MB
docker.io/centos    latest              fd44297e2ddb        2 weeks ago         215.7 MB
$ docker images --filter label=distro=fedora21
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
f21                 latest              d5f771d03056        2 hours ago         241.3 MB
```

As you can see from the preceding screenshot, if we use filters with the `docker images` command, we only get an image where the corresponding label is found in the image's metadata.

Getting ready

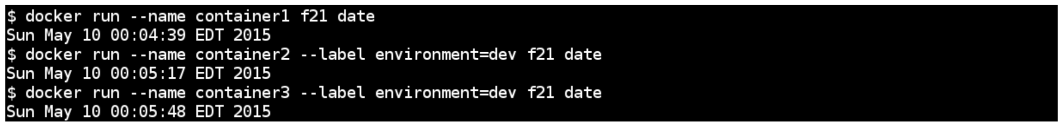
Make sure that the Docker daemon 1.6 and above is running on the host and you can connect through the Docker client.

How to do it...

1. To start the container with the `--label/-l` option, run the following command:

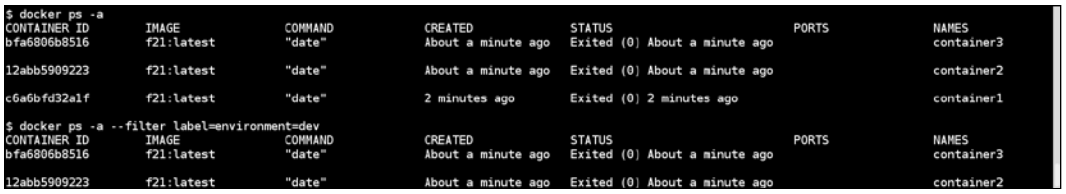
   ```
   $ docker run --label environment=dev f21 date
   ```

2. Let's start a container without a label and start two others with the same label:



```
$ docker run --name container1 f21 date
Sun May 10 00:04:39 EDT 2015
$ docker run --name container2 --label environment=dev f21 date
Sun May 10 00:05:17 EDT 2015
$ docker run --name container3 --label environment=dev f21 date
Sun May 10 00:05:48 EDT 2015
```

If we list all the containers without a label, we will see all the containers, but if we use label, then we get only containers, which matches the label.



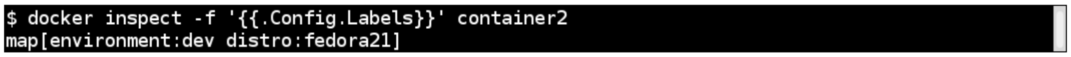
```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED              STATUS                          PORTS               NAMES
bfa6806b8516        f21:latest          "date"              About a minute ago   Exited (0) About a minute ago                       container3

12abb5909223        f21:latest          "date"              About a minute ago   Exited (0) About a minute ago                       container2

c6a6bfd32a1f        f21:latest          "date"              2 minutes ago        Exited (0) 2 minutes ago                            container1

$ docker ps -a --filter label=environment=dev
CONTAINER ID        IMAGE               COMMAND             CREATED              STATUS                          PORTS               NAMES
bfa6806b8516        f21:latest          "date"              About a minute ago   Exited (0) About a minute ago                       container3

12abb5909223        f21:latest          "date"              About a minute ago   Exited (0) About a minute ago                       container2
```

How it works...

Docker attaches label metadata to containers while starting them and matches the label while listing them or other related operations.

There's more...

- We can list all the labels attached to a container through the `inspect` command, which we saw in an earlier recipe. As we can see, the `inspect` command returns both the image and the container labels.



```
$ docker inspect -f '{{.Config.Labels}}' container2
map[environment:dev distro:fedora21]
```

- You can apply labels from a file (with the `--from-file` option) that has a list of labels, separated by a new EOL.
- These labels are different from the Kubernetes label, which we will see in *Chapter 8*, *Docker Orchestration and Hosting Platforms*.

See also

- The documentation on the Docker website `https://docs.docker.com/reference/builder/#label`
- `http://rancher.com/docker-labels/`

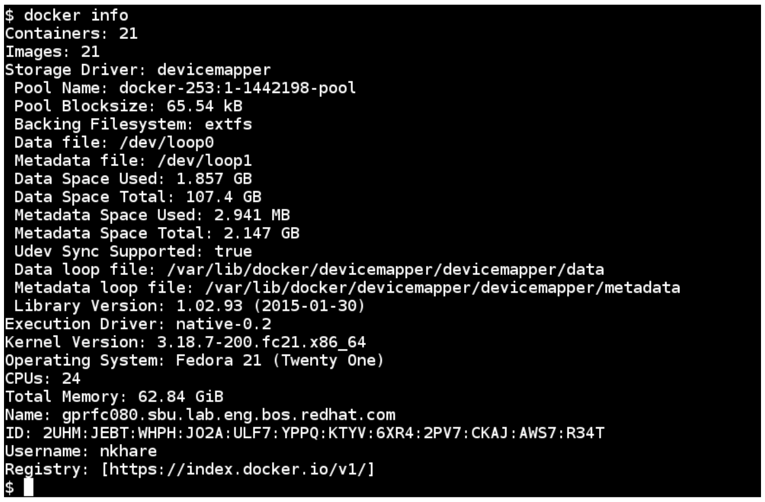
3
Working with Docker Images

In this chapter, we will cover the following recipes:

- Creating an account with Docker Hub
- Creating an image from the container
- Publishing an image to the registry
- Looking at the history of an image
- Deleting an image
- Exporting an image
- Importing an image
- Building images using Dockerfiles
- Building an Apache image – a Dockerfile example
- Accessing Firefox from a container – a Dockerfile example
- Building a WordPress image – a Dockerfile example
- Setting up a private index/registry
- Automated Builds – with GitHub and Bitbucket
- Creating the base image – using supermin
- Creating the base image – using Debootstrap
- Visualizing dependencies between layers

Introduction

In this chapter, we will focus on operations relating to images. As we know, images are required to run containers. You can either use existing images or create new custom images. You will need to create custom images to suit your development and deployment environment. Once you create an image, you can share it through the public or private registry. Before we explore more about Docker images, let's look at the output of the `docker info` command:



```
$ docker info
Containers: 21
Images: 21
Storage Driver: devicemapper
 Pool Name: docker-253:1-1442198-pool
 Pool Blocksize: 65.54 kB
 Backing Filesystem: extfs
 Data file: /dev/loop0
 Metadata file: /dev/loop1
 Data Space Used: 1.857 GB
 Data Space Total: 107.4 GB
 Metadata Space Used: 2.941 MB
 Metadata Space Total: 2.147 GB
 Udev Sync Supported: true
 Data loop file: /var/lib/docker/devicemapper/devicemapper/data
 Metadata loop file: /var/lib/docker/devicemapper/devicemapper/metadata
 Library Version: 1.02.93 (2015-01-30)
Execution Driver: native-0.2
Kernel Version: 3.18.7-200.fc21.x86_64
Operating System: Fedora 21 (Twenty One)
CPUs: 24
Total Memory: 62.84 GiB
Name: gprfc080.sbu.lab.eng.bos.redhat.com
ID: 2UHM:JEBT:WHPH:J02A:ULF7:YPPQ:KTYV:6XR4:2PV7:CKAJ:AWS7:R34T
Username: nkhare
Registry: [https://index.docker.io/v1/]
$
```

The preceding command gives the current system-wide info as follows:

- It has 21 containers and 21 images.
- The current storage driver, `devicemapper`, and its related information, such as thin pool name, data, metadata file, and so on. Other types of storage drivers are aufs, btrfs, overlayfs, vfs, and so on. Devicemapper, btrfs, and overlayfs have native support in the Linux kernel. AUFS support needs a patched kernel. We talked about the Union filesystem in *Chapter 1, Introduction and Installation*.
- To leverage the kernel features that enable containerization, the Docker daemon has to talk to the Linux kernel. This is done through the execution driver. `libconatiner` or `native` is one of that type. The others are `libvirt`, `lxc`, and so on, which we saw in *Chapter 1, Introduction and Installation*.
- The kernel version on the host operating system.
- The user account that is registered on the registry mentioned in the next section to pull/push images.

I am using Fedora 20/21 as my primary environment to run the recipes. They should also work with other environments.

Creating an account with Docker Hub

Docker Hub is like GitHub for images. It is a public registry on which you can host images both public and private, share them and collaborate with others. It has integration with GitHub, Bitbucket, and can trigger automated builds.

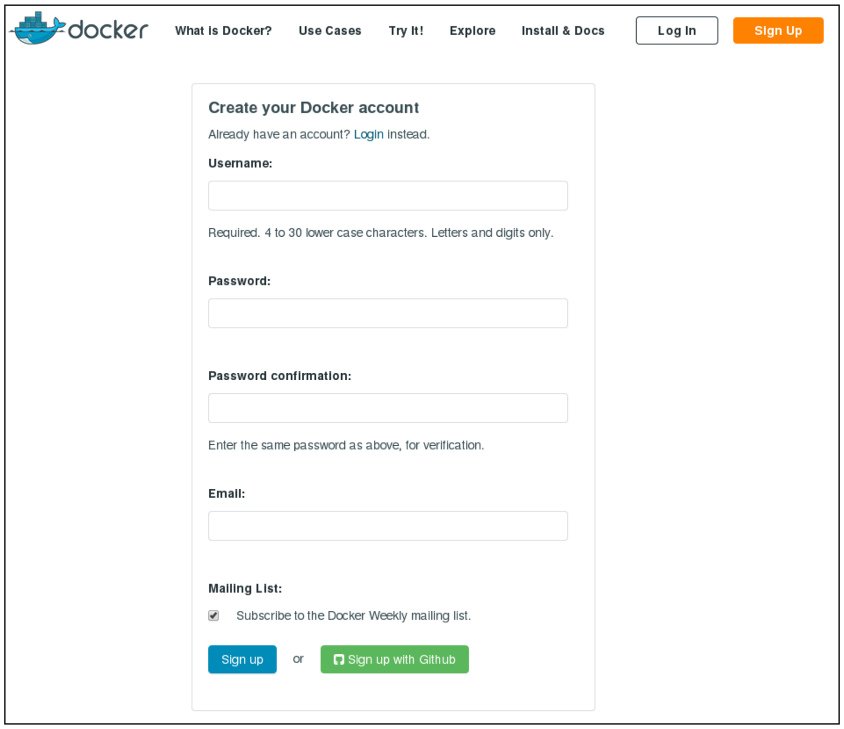
As of now, the creation of an account on Docker Hub is free. A repository can hold different versions of an image. You can create any number of public repositories for your images. By default, you will have one private repository, which will not be accessible to the public. You can buy more private repositories. You can create an account either through a web browser or from the command line.

Getting ready

To sign up from the command line, you will need to have Docker installed on your system.

How to do it...

1. To create an account through a web browser on Docker Hub, visit `https://hub.docker.com/account/signup/` and create an account:



Create your Docker account

Already have an account? Login instead.

Username:

Required. 4 to 30 lower case characters. Letters and digits only.

Password:

Password confirmation:

Enter the same password as above, for verification.

Email:

Mailing List:

☑ Subscribe to the Docker Weekly mailing list.

2. To create an account using the command line, run the following command and submit the required details:

   ```
   $ docker login
   ```

How it works...

The preceding steps will create a Docker Hub account for you. Once the account is created, you'll get a confirmation mail, through which you need to confirm your identity.

See also

- The documentation on the Docker website:
    - `https://docs.docker.com/docker-hub`
    - `https://docs.docker.com/docker-hub/accounts/`

Creating an image from the container

There are a couple of ways to create images, one is by manually committing layers and the other way is through Dockerfiles. In this recipe, we'll see the former and look at Dockerfiles later in the chapter.

As we start a new container, a read/write layer gets attached to it. This layer will get destroyed if we do not save it. In this recipe, we will see how to save that layer and make a new image from the running or stopped container using the `docker commit` command.

Getting ready

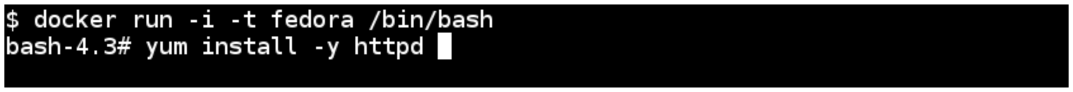
To get a Docker image, start a container with it.

How to do it...

1. To do the commit, run the following command:

   ```
   docker commit -a|--author[=""] -m|--message[=""] CONTAINER
   [REPOSITORY[:TAG]]
   ```

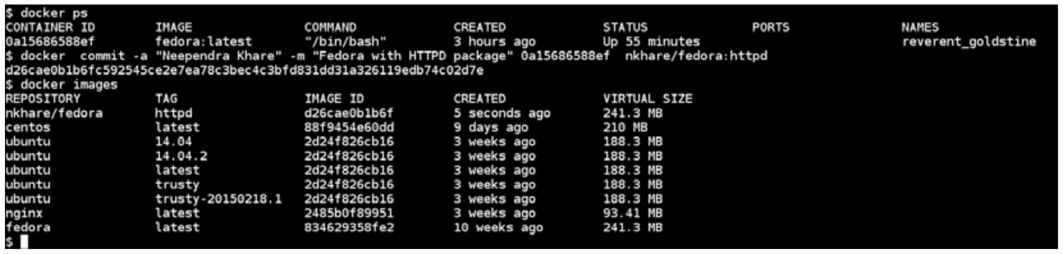
2. Let's start a container and create/modify some files using the `install httpd` package:



```
$ docker run -i -t fedora /bin/bash
bash-4.3# yum install -y httpd
```

3. Then, open a new terminal and create a new image by doing the commit:

   ```
   $ docker commit -a "Neependra Khare" -m "Fedora with HTTPD
   package" 0a15686588ef nkhare/fedora:httpd
   ```



```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
0a15686588ef        fedora:latest       "/bin/bash"         3 hours ago         Up 55 minutes                           reverent_goldstine
$ docker  commit -a "Neependra Khare" -m "Fedora with HTTPD package" 0a15686588ef  nkhare/fedora:httpd
d26cae0b1b6fc592545ce2e7ea78c3bec4c3bfd831dd31a326119edb74c02d7e
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
nkhare/fedora       httpd               d26cae0b1b6f        5 seconds ago       241.3 MB
centos              latest              88f9454e60dd        9 days ago          210 MB
ubuntu              14.04               2d24f826cb16        3 weeks ago         188.3 MB
ubuntu              14.04.2             2d24f826cb16        3 weeks ago         188.3 MB
ubuntu              latest              2d24f826cb16        3 weeks ago         188.3 MB
ubuntu              trusty              2d24f826cb16        3 weeks ago         188.3 MB
ubuntu              trusty-20150218.1   2d24f826cb16        3 weeks ago         188.3 MB
nginx               latest              2485b0f89951        3 weeks ago         93.41 MB
fedora              latest              834629358fe2        10 weeks ago        241.3 MB
$ 
```

As you can see, the new image is now being committed to the local repository with `nkhare/fedora` as a name and `httpd` as a tag.

How it works...

In *Chapter 1*, *Introduction and Installation*, we saw that while starting a container, a read/write filesystem layer will be created on top of the existing image layers from which the container started, and with the installation of a package, some files would have been added/modified in that layer. All of those changes are currently in the ephemeral read/write filesystem layer, which is assigned to the container. If we stop and delete the container, then all of the earlier mentioned modifications would be lost.

Using commit, we create a new layer with the changes that have occurred since the container started, which get saved in the backend storage driver.

There's more...

- To look for files, which are changed since the container started:

  ```
  $ docker diff CONTAINER
  ```

  In our case, we will see something like the following code:

  ```
  $ docker diff 0a15686588ef
  .....
  C /var/log
  A /var/log/httpd
  C /var/log/lastlog
  .....
  ```

We can see a prefix before each entry of the output. The following is a list of those prefixes:

- `A`: This is for when a file/directory has been added
- `C`: This is for when a file/directory has been modified
- `D`: This is for when a file/directory has been deleted

- By default, a container gets paused while doing the commit. You can change its behavior by passing `--pause=false` to commit.

See also

- Look at the `help` option of `docker commit`:

  ```
  $ docker commit --help
  ```

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#commit`

Publishing an image to the registry

Let's say you have created an image that suits the development environment in your organization. You can either share it using tar ball, which we'll see later in this chapter, or put in a central location from where the user can pull it. This central location can be either a public or a private registry. In this recipe, we'll see how to push the image to the registry using the `docker push` command. Later in this chapter, we'll cover how to set up the private registry.

Getting ready

You will need a valid account on Docker Hub to push images/repositories.

A local registry must be set up if you are pushing images/repositories locally.

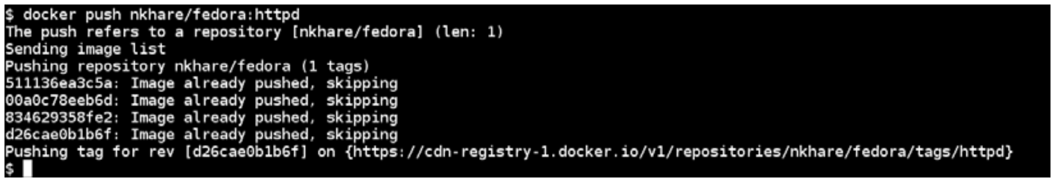
How to do it…

```
$ docker push NAME[:TAG]
```

By default, the preceding command will use the username and registry shown in the `docker info` command to push the images. As shown in the preceding screenshot, the command will use `nkhare` as the username and `https://index.docker.io/v1/` as the registry.

To push the image that we created in the previous section, run the following command:

```
$ docker push nkhare/fedora:httpd
```



```
$ docker push nkhare/fedora:httpd
The push refers to a repository [nkhare/fedora] (len: 1)
Sending image list
Pushing repository nkhare/fedora (1 tags)
511136ea3c5a: Image already pushed, skipping
00a0c78eeb6d: Image already pushed, skipping
834629358fe2: Image already pushed, skipping
d26cae0b1b6f: Image already pushed, skipping
Pushing tag for rev [d26cae0b1b6f] on {https://cdn-registry-1.docker.io/v1/repositories/nkhare/fedora/tags/httpd}
$ 
```

Let's say you want to push the image to the local registry, which is hosted on a host called `local-registry`. To do this, you first need to tag the image with the registry host's name or IP address with the port number on which the registry is running and then push the images.

```
$ docker tag [-f|--force[=false] IMAGE
[REGISTRYHOST/][USERNAME/]NAME[:TAG]
$ docker push [REGISTRYHOST/][USERNAME/]NAME[:TAG]
```

For example, let's say our registry is configured on `shadowfax.example.com`, then to tag the image use the following command:

```
$ docker tag nkhare/fedora:httpd
shadowfax.example.com:5000/nkhare/fedora:httpd
```

Then, to push the image, use the following command:

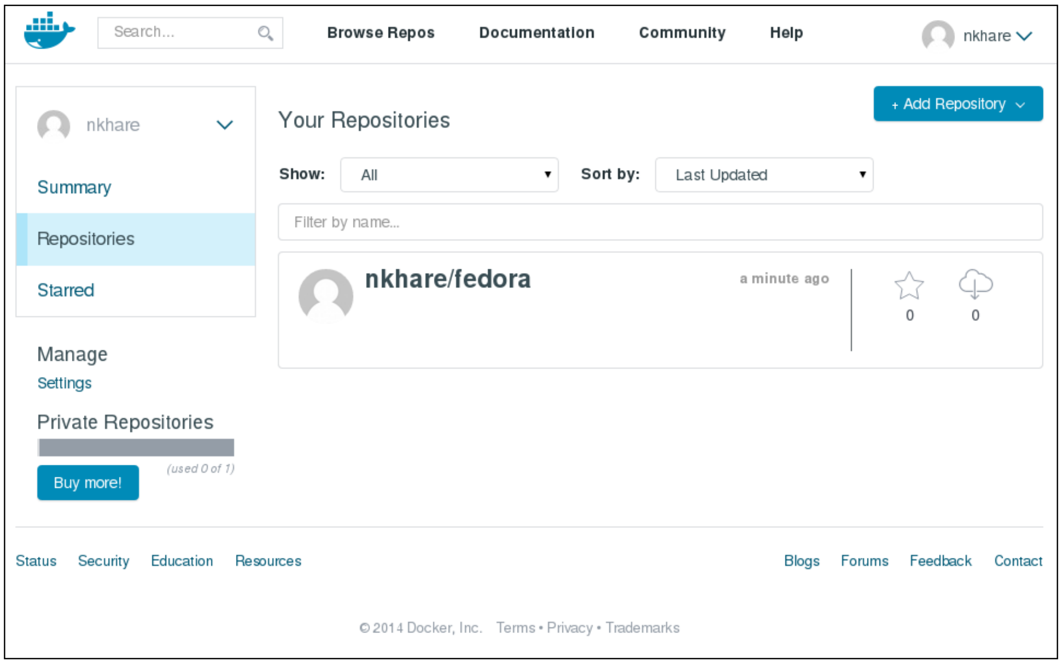
```
$ docker push shadowfax.example.com:5000/nkhare/fedora:httpd
```

How it works...

It will first list down all the intermediate layers that are required to make that specific image. It will then check to see, out of those layers, how many are already present inside the registry. At last, it will copy all the layers, which are not present in the registry with the metadata required to build the image.

There's more…

As we pushed our image to the public registry, we can log in to Docker Hub and look for the image:



See also

- Look at the `help` option of `docker push`:

  ```
  $ docker push --help
  ```

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#push`

Looking at the history of an image

It is handy to know how the image that we are using has been created. The `docker history` command helps us find all the intermediate layers.

Getting ready

Pull or import any Docker image.

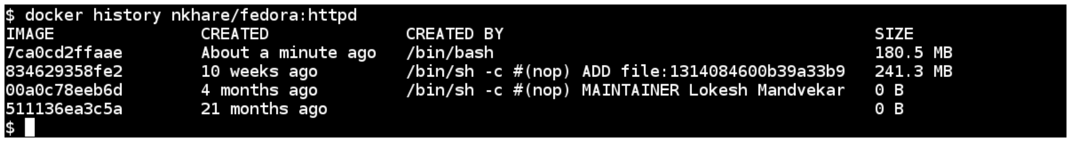
How to do it...

1. To look at the history of the image, consider the following syntax:

   ```
   $ docker history [ OPTIONS ] IMAGE
   ```

   Here's an example using the preceding syntax:

   ```
   $ docker history nkhare/fedora:httpd
   ```



```
$ docker history nkhare/fedora:httpd
IMAGE               CREATED              CREATED BY                                      SIZE
7ca0cd2ffaae        About a minute ago   /bin/bash                                       180.5 MB
834629358fe2        10 weeks ago         /bin/sh -c #(nop) ADD file:1314084600b39a33b9   241.3 MB
00a0c78eeb6d        4 months ago         /bin/sh -c #(nop) MAINTAINER Lokesh Mandvekar   0 B
511136ea3c5a        21 months ago                                                        0 B
$ 
```

How it works...

From the metadata of an image, Docker can know how an image is being created. With the `history` command, it will look at the metadata recursively to get to the origin.

There's more...

Look at the commit message of a layer that got committed:

```
$ docker inspect --format='{{.Comment}}' nkhare/fedora:httpd
Fedora with HTTPD package
```

Currently, there is no direct way to look at the commit message for each layer using one single command, but we can use the `inspect` command, which we saw earlier, for each layer.

See also

- Look at the `help` option of `docker history`:

  ```
  $ docker history --help
  ```

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#history`

Deleting an image

To remove the image from the host, we can use the `docker rmi` command. However, this does not remove images from the registry.

Getting ready

Make sure one or more Docker images are locally available.

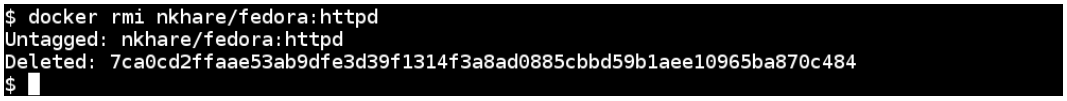
How to do it...

1. To remove the image, consider the following syntax:

   ```
   $ docker rmi [ OPTIONS ] IMAGE [IMAGE...]
   ```

   In our case, here's an example using the preceding syntax:

   ```
   $ docker rmi nkhare/fedora:httpd
   ```



```
$ docker rmi nkhare/fedora:httpd
Untagged: nkhare/fedora:httpd
Deleted: 7ca0cd2ffaae53ab9dfe3d39f1314f3a8ad0885cbbd59b1aee10965ba870c484
$ 
```

There's more...

If you want to remove all containers and images, then do following; however, be sure about what you are doing, as this is very destructive:

- To stop all containers, use the following command:

  ```
  $ docker stop 'docker ps -q'
  ```

- To delete all containers, use the following command:

  ```
  $ docker rm 'docker ps -a -q'
  ```

- To delete all images, use the following command:

  ```
  $ docker rmi 'docker images -q'
  ```

See also

- Look at the `help` option of `docker rmi`:

  ```
  $ docker rmi --help
  ```

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#rmi`

Exporting an image

Let's say you have a customer who has very strict policies that do not allow them to use images from the public domain. In such cases, you can share images through tarballs, which later can be imported on another system. In this recipe, we will see how to do that using the `docker save` command.

Getting ready

Pull or import one or more Docker images on the Docker host.

How to do it...

1. Use the following syntax to save the image in the tar file:

   ```
   $ docker save [-o|--output=""] IMAGE [:TAG]
   ```

   For example, to create a tar archive for Fedora, run the following command:

   ```
   $ docker save --output=fedora.tar fedora
   ```

If the tag name is specified with the image name we want to export, such as `fedora:latest`, then only the layers related to that tag will get exported.

There's more...

If `--output` or `-o` is not used, then the output will be streamed to `STDOUT`:

```
$ docker save fedora:latest > fedora-latest.tar
```

Similarly, the contents of the container's filesystem can be exported using the following command:

```
$ docker export CONTAINER  > containerXYZ.tar
```

See also

- Look at the `help` option of `docker save` and `docker export`:

  ```
  $ docker save -help
  $ docker export --help
  ```

- The documentation on the Docker website:
  - `https://docs.docker.com/reference/commandline/cli/#save`
  - `https://docs.docker.com/reference/commandline/cli/#export`

Importing an image

To get a local copy of the image, we either need to pull it from the accessible registry or import it from the already exported image, as we saw in the earlier recipe. Using the `docker import` command, we import an exported image.

Getting ready

You need an accessible exported Docker image.

How to do it...

1. To import an image, we can use following syntax:

   ```
   $ docker import URL|- [REPOSITORY[:TAG]]
   ```

   Here's an example using the preceding syntax:

   ```
   $ cat fedora-latest.tar | docker import - fedora:latest
   ```

   Alternatively, you can consider the following example:

   ```
   $ docker import http://example.com/example.tar example/image
   ```

The preceding example will first create an empty filesystem and then import the contents.

See also

- Look at the `help` option of `docker import`:

  ```
  $ docker import --help
  ```

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#import`

Building images using Dockerfiles

Dockerfiles help us in automating image creation and getting precisely the same image every time we want it. The Docker builder reads instructions from a text file (a Dockerfile) and executes them one after the other in order. It can be compared as Vagrant files, which allows you to configure VMs in a predictable manner.

Getting ready

A Dockerfile with build instructions.

- Create an empty directory:

  ```
  $ mkdir sample_image
  $ cd sample_image
  ```

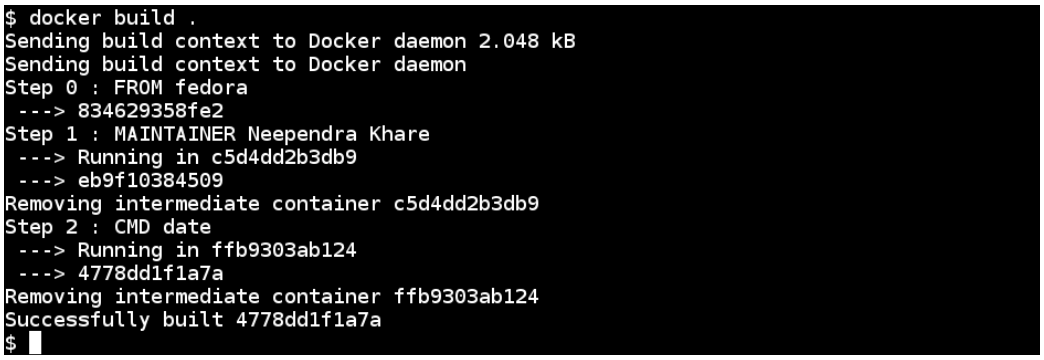
- Create a file named `Dockerfile` with the following content:

  ```
  $ cat Dockerfile
  # Pick up the base image
  FROM fedora
  # Add author name
  MAINTAINER Neependra Khare
  # Add the command to run at the start of container
  CMD date
  ```

How to do it...

1. Run the following command inside the directory, where we created Dockerfile to build the image:

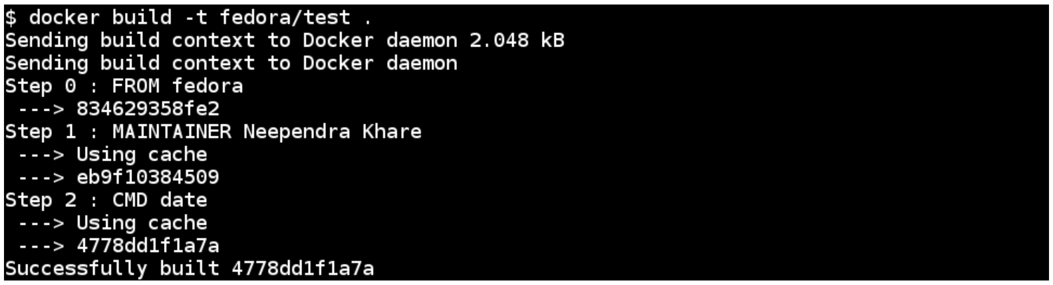
   ```
   $ docker build .
   ```



```
$ docker build .
Sending build context to Docker daemon 2.048 kB
Sending build context to Docker daemon
Step 0 : FROM fedora
 ---> 834629358fe2
Step 1 : MAINTAINER Neependra Khare
 ---> Running in c5d4dd2b3db9
 ---> eb9f10384509
Removing intermediate container c5d4dd2b3db9
Step 2 : CMD date
 ---> Running in ffb9303ab124
 ---> 4778dd1f1a7a
Removing intermediate container ffb9303ab124
Successfully built 4778dd1f1a7a
$ 
```

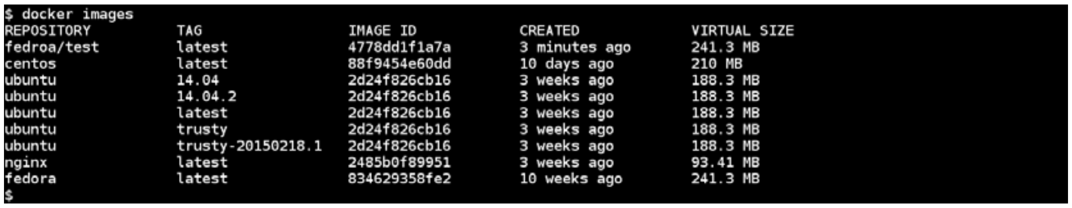
We did not specify any repository or tag name while building the image. We can give those with the `-t` option as follows:

```
$ docker build -t fedora/test .
```



```
$ docker build -t fedora/test .
Sending build context to Docker daemon 2.048 kB
Sending build context to Docker daemon
Step 0 : FROM fedora
 ---> 834629358fe2
Step 1 : MAINTAINER Neependra Khare
 ---> Using cache
 ---> eb9f10384509
Step 2 : CMD date
 ---> Using cache
 ---> 4778dd1f1a7a
Successfully built 4778dd1f1a7a
```

The preceding output is different from what we did earlier. However, here we are using a cache after each instruction. Docker tries to save the intermediate images as we saw earlier and tries to use them in subsequent builds to accelerate the build process. If you don't want to cache the intermediate images, then add the `--no-cache` option with the build. Let's take a look at the available images now:



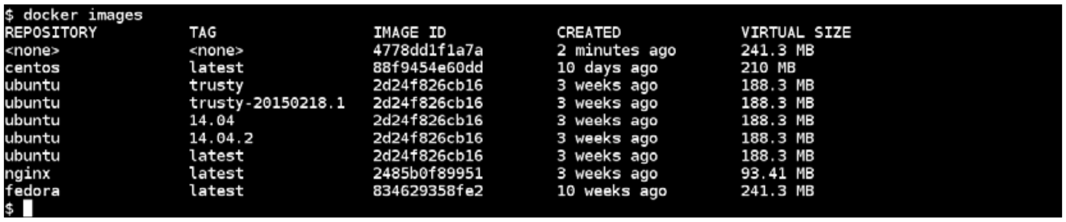
```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
fedroa/test         latest              4778dd1f1a7a        3 minutes ago       241.3 MB
centos              latest              88f9454e60dd        10 days ago         210 MB
ubuntu              14.04               2d24f826cb16        3 weeks ago         188.3 MB
ubuntu              14.04.2             2d24f826cb16        3 weeks ago         188.3 MB
ubuntu              latest              2d24f826cb16        3 weeks ago         188.3 MB
ubuntu              trusty              2d24f826cb16        3 weeks ago         188.3 MB
ubuntu              trusty-20150218.1   2d24f826cb16        3 weeks ago         188.3 MB
nginx               latest              2485b0f89951        3 weeks ago         93.41 MB
fedora              latest              834629358fe2        10 weeks ago        241.3 MB
$
```

How it works...

A context defines the files used to build the Docker image. In the preceding command, we define the context to the build. The build is done by the Docker daemon and the entire context is transferred to the daemon. This is why we see the `Sending build context to Docker daemon 2.048 kB` message. If there is a file named `.dockerignore` in the current working directory with the list of files and directories (new line separated), then those files and directories will be ignored by the build context. More details about `.dockerignore` can be found at `https://docs.docker.com/reference/builder/#the-dockerignore-file`.

After executing each instruction, Docker commits the intermediate image and runs a container with it for the next instruction. After the next instruction has run, Docker will again commit the container to create the intermediate image and remove the intermediate container created in the previous step.

For example, in the preceding screenshot, `eb9f10384509` is an intermediate image and `c5d4dd2b3db9` and `ffb9303ab124` are the intermediate containers. After the last instruction is executed, the final image will be created. In this case, the final image is `4778dd1f1a7a`:



```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
<none>              <none>              4778dd1f1a7a        2 minutes ago       241.3 MB
centos              latest              88f9454e60dd        10 days ago         210 MB
ubuntu              trusty              2d24f826cb16        3 weeks ago         188.3 MB
ubuntu              trusty-20150218.1   2d24f826cb16        3 weeks ago         188.3 MB
ubuntu              14.04               2d24f826cb16        3 weeks ago         188.3 MB
ubuntu              14.04.2             2d24f826cb16        3 weeks ago         188.3 MB
ubuntu              latest              2d24f826cb16        3 weeks ago         188.3 MB
nginx               latest              2485b0f89951        3 weeks ago         93.41 MB
fedora              latest              834629358fe2        10 weeks ago        241.3 MB
$ 
```

The `-a` option can be specified with the `docker images` command to look for intermediate layers:

`$ docker images -a`

There's more…

The format of the Dockerfile is:

`INSTRUCTION arguments`

Generally, instructions are given in uppercase, but they are not case sensitive. They are evaluated in order. A `#` at the beginning is treated like a comment.

Let's take a look at the different types of instructions:

- `FROM`: This must be the first instruction of any Dockerfile, which sets the base image for subsequent instructions. By default, the latest tag is assumed to be:

  `FROM  <image>`

  Alternatively, consider the following tag:

  `FROM  <images>:<tag>`

  There can be more than one `FROM` instruction in one Dockerfile to create multiple images.

If only image names, such as Fedora and Ubuntu are given, then the images will be downloaded from the default Docker registry (Docker Hub). If you want to use private or third-party images, then you have to mention this as follows:

```
            [registry_hostname[:port]/][user_name/](repository_
name:version_tag)
```

Here is an example using the preceding syntax:

```
FROM registry-host:5000/nkhare/f20:httpd
```

- `MAINTAINER`: This sets the author for the generated image, `MAINTAINER <name>`.
- `RUN`: We can execute the `RUN` instruction in two ways—first, run in the shell (`sh -c`):

  ```
  RUN <command> <param1> ... <pamamN>
  ```

  Second, directly run an executable:

  ```
  RUN ["executable", "param1",...,"paramN" ]
  ```

  As we know with Docker, we create an overlay—a layer on top of another layer—to make the resulting image. Through each `RUN` instruction, we create and commit a layer on top of the earlier committed layer. A container can be started from any of the committed layers.

  By default, Docker tries to cache the layers committed by different `RUN` instructions, so that it can be used in subsequent builds. However, this behavior can be turned off using `--no-cache flag` while building the image.

- `LABEL`: Docker 1.6 added a new feature to the attached arbitrary key-value pair to Docker images and containers. We covered part of this in the *Labeling and filtering containers* recipe in *Chapter 2*, *Working with Docker Containers*. To give a label to an image, we use the `LABEL` instruction in the Dockerfile as `LABEL distro=fedora21`.
- `CMD`: The `CMD` instruction provides a default executable while starting a container. If the `CMD` instruction does not have an executable (parameter 2), then it will provide arguments to `ENTRYPOINT`.

  ```
  CMD  ["executable", "param1",...,"paramN" ]
  CMD ["param1", ... , "paramN"]
  CMD <command> <param1> ... <pamamN>
  ```

  Only one `CMD` instruction is allowed in a Dockerfile. If more than one is specified, then only the last one will be honored.

- `ENTRYPOINT`: This helps us configure the container as an executable. Similar to `CMD`, there can be at max one instruction for `ENTRYPOINT`; if more than one is specified, then only the last one will be honored:

  ```
  ENTRYPOINT  ["executable", "param1",...,"paramN" ]
  ENTRYPOINT <command> <param1> ... <pamamN>
  ```

Once the parameters are defined with the ENTRYPOINT instruction, they cannot be overwritten at runtime. However, ENTRYPOINT can be used as CMD, if we want to use different parameters to ENTRYPOINT.

- EXPOSE: This exposes the network ports on the container on which it will listen at runtime:

  ```
  EXPOSE  <port> [<port> ... ]
  ```

  We can also expose a port while starting the container. We covered this in the *Exposing a port while starting a container* recipe in *Chapter 2*, *Working with Docker Containers*.

- ENV: This will set the environment variable <key> to <value>. It will be passed all the future instructions and will persist when a container is run from the resulting image:

  ```
  ENV <key> <value>
  ```

- ADD: This copies files from the source to the destination:

  ```
  ADD <src> <dest>
  ```

  The following one is for the path containing white spaces:

  ```
  ADD ["<src>"... "<dest>"]
  ```

  - <src>: This must be the file or directory inside the build directory from which we are building an image, which is also called the context of the build. A source can be a remote URL as well.
  - <dest>: This must be the absolute path inside the container in which the files/directories from the source will be copied.

- COPY: This is similar to ADD.COPY <src> <dest>:

  ```
  COPY  ["<src>"... "<dest>"]
  ```

- VOLUME: This instruction will create a mount point with the given name and flag it as mounting the external volume using the following syntax:

  ```
  VOLUME ["/data"]
  ```

  Alternatively, you can use the following code:

  ```
  VOLUME /data
  ```

- USER: This sets the username for any of the following run instructions using the following syntax:

  ```
  USER  <username>/<UID>
  ```

- `WORKDIR`: This sets the working directory for the `RUN`, `CMD`, and `ENTRYPOINT` instructions that follow it. It can have multiple entries in the same Dockerfile. A relative path can be given which will be relative to the earlier `WORKDIR` instruction using the following syntax:

  ```
  WORKDIR <PATH>
  ```

- `ONBUILD`: This adds trigger instructions to the image that will be executed later, when this image will be used as the base image of another image. This trigger will run as part of the `FROM` instruction in downstream Dockerfile using the following syntax:

  ```
  ONBUILD [INSTRUCTION]
  ```

See also

- Look at the `help` option of `docker build`:

  ```
  $ docker build -help
  ```

- The documentation on the Docker website `https://docs.docker.com/reference/builder/`

Building an Apache image – a Dockerfile example

I am going to refer Dockerfiles from the Fedora-Dockerfiles GitHub repo (`https://github.com/fedora-cloud/Fedora-Dockerfiles`) after forking it. If you are using Fedora, then you can also install the `fedora-dockerfiles` package to get the sample Dockerfiles in `/usr/share/fedora-dockerfiles`. In each of the subdirectories, you will put a Dockerfile, the supporting files and a README file.

The Fedora-Dockerfiles GitHub repo would have the latest examples and I highly recommend that you try out latest bits.

Getting ready

Clone the Fedora-Dockerfiles Git repo using the following command:

```
$ git clone https://github.com/nkhare/Fedora-Dockerfiles.git
```

Now, go to the `apache` subdirectory:

```
$ cd Fedora-Dockerfiles/apache/
$ cat Dockerfile
FROM fedora:20
```

```
MAINTAINER "Scott Collier" <scollier@redhat.com>

RUN yum -y update && yum clean all
RUN yum -y install httpd && yum clean all
RUN echo "Apache" >> /var/www/html/index.html

EXPOSE 80

# Simple startup script to avoid some issues observed with container
restart
ADD run-apache.sh /run-apache.sh
RUN chmod -v +x /run-apache.sh

CMD ["/run-apache.sh"]
```

The other supporting files are:

- `README.md`: This is the README file
- `run-apache.sh`: This is the script to run `HTTPD` in the foreground
- `LICENSE`: This is the GPL license

How to do it...

With the following `build` command, we can build a new image:

```
$ docker build -t fedora/apache .
Sending build context to Docker daemon 23.55 kB
Sending build context to Docker daemon
Step 0 : FROM fedora:20
 ---> 6cece30db4f9
Step 1 : MAINTAINER "Scott Collier" <scollier@redhat.com>
 ---> Running in 2048200e6338
 ---> ae8e3c258061
Removing intermediate container 2048200e6338
Step 2 : RUN yum -y update && yum clean all
 ---> Running in df8bc8ee3117
.... Installing/Update packages ...
Cleaning up everything
```

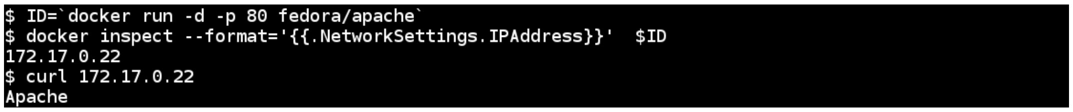
```
 ---> 5a6d449e59f6
Removing intermediate container df8bc8ee3117
Step 3 : RUN yum -y install httpd && yum clean all
 ---> Running in 24449e520f18
.... Installing HTTPD ...
Cleaning up everything
 ---> ae1625544ef6
Removing intermediate container 24449e520f18
Step 4 : RUN echo "Apache" >> /var/www/html/index.html
 ---> Running in a35cbcd8d97a
 ---> 251eea31b3ce
Removing intermediate container a35cbcd8d97a
Step 5 : EXPOSE 80
 ---> Running in 734e54f4bf58
 ---> 19503ae2a8cf
Removing intermediate container 734e54f4bf58
Step 6 : ADD run-apache.sh /run-apache.sh
 ---> de35d746f43b
Removing intermediate container 3eec9a46da64
Step 7 : RUN chmod -v +x /run-apache.sh
 ---> Running in 3664efba393f
mode of '/run-apache.sh' changed from 0644 (rw-r--r--) to 0755 (rwxr-
xr-x)
 ---> 1cb729521c3f
Removing intermediate container 3664efba393f
Step 8 : CMD /run-apache.sh
 ---> Running in cd5e7534e815
 ---> 5f8041b6002c
Removing intermediate container cd5e7534e815
Successfully built 5f8041b6002c
```

How it works...

The build process takes a base image, installs the required `HTTPD` package and creates an HTML page. Then, it exposes port `80` to serve the web page and sets instructions to start Apache at the start of the container.

There's more...

Let's run the container from the created image, get its IP address, and access the web page from it:



```
$ ID=`docker run -d -p 80 fedora/apache`
$ docker inspect --format='{{.NetworkSettings.IPAddress}}'  $ID
172.17.0.22
$ curl 172.17.0.22
Apache
```

See also

- Look at the `help` option of `docker build`:

  `$ docker build --help`

- The documentation on the Docker website `https://docs.docker.com/reference/builder/`

Accessing Firefox from a container – a Dockerfile example

We can do something more interesting through a Dockerfile, such as creating a container that just runs Firefox. This kind of use case can help run multiple browsers of different versions on the same machine, which can be very helpful in doing multibrowser testing.

Getting ready

Clone the Fedora-Dockerfiles Git repo using the following command:

```
$ git clone  https://github.com/nkhare/Fedora-Dockerfiles.git
```

Then, go to the `firefox` subdirectory.

```
$ cd Fedora-Dockerfiles/firefox
$ cat Dockerfile
FROM fedora
MAINTAINER scollier <emailscottcollier@gmail.com>

# Install the appropriate software
RUN yum -y update && yum clean all
RUN yum -y install x11vnc \
firefox xorg-x11-server-Xvfb \
xorg-x11-twm tigervnc-server \
xterm xorg-x11-font \
xulrunner-26.0-2.fc20.x86_64 \
dejavu-sans-fonts \
dejavu-serif-fonts \
xdotool && yum clean all

# Add the xstartup file into the image
ADD ./xstartup /

RUN mkdir /.vnc
RUN x11vnc -storepasswd 123456 /.vnc/passwd
RUN  \cp -f ./xstartup /.vnc/.
RUN chmod -v +x /.vnc/xstartup
RUN sed -i '/\/etc\/X11\/xinit\/xinitrc-common/a [ -x
/usr/bin/firefox ] && /usr/bin/firefox &' /etc/X11/xinit/xinitrc

EXPOSE 5901

CMD    ["vncserver", "-fg" ]
# ENTRYPOINT ["vncserver", "-fg" ]
```

Supporting files:

- `README.md`: This is a README file
- `LICENSE`: This is the GPL license
- `xstartup`: This is the script to set up the X11 environment

How to do it...

Run the following command to build the image:

```
$ docker build  -t fedora/firefox .
Sending build context to Docker daemon 24.58 kB
Sending build context to Docker daemon
Step 0 : FROM fedora
 ---> 834629358fe2
Step 1 : MAINTAINER scollier <emailscottcollier@gmail.com>
```

```
 ---> Running in ae0fd3c2cb2e
 ---> 7ffc6c9af827
Removing intermediate container ae0fd3c2cb2e
Step 2 : RUN yum -y update && yum clean all
 ---> Running in 1c67b8772718
..... Installing/Update packages ...
 ---> 075d6ceef3d0
Removing intermediate container 1c67b8772718
Step 3 : RUN yum -y install x11vnc firefox xorg-x11-server-Xvfb xorg-
x11-twm tigervnc-server xterm xorg-x11-font xulrunner-26.0-
2.fc20.x86_64 dejavu-sans-fonts dejavu-serif-fonts xdotool && yum
clean all
..... Installing required packages packages ...
Cleaning up everything
 ---> 986be48760a6
Removing intermediate container c338a1ad6caf
Step 4 : ADD ./xstartup /
 ---> 24fa081dcea5
Removing intermediate container fe98d86ba67f
Step 5 : RUN mkdir /.vnc
 ---> Running in fdb8fe7e697a
 ---> 18f266ace765
Removing intermediate container fdb8fe7e697a
Step 6 : RUN x11vnc -storepasswd 123456 /.vnc/passwd
 ---> Running in c5b7cdba157f
stored passwd in file: /.vnc/passwd
 ---> e4fcf9b17aa9
Removing intermediate container c5b7cdba157f
Step 7 : RUN \cp -f ./xstartup /.vnc/.
 ---> Running in 21d0dc4edb4e
 ---> 4c53914323cb
Removing intermediate container 21d0dc4edb4e
Step 8 : RUN chmod -v +x /.vnc/xstartup
 ---> Running in 38f18f07c996
mode of '/.vnc/xstartup' changed from 0644 (rw-r--r--) to 0755 (rwxr-
xr-x)
 ---> caa278024354
```

```
Removing intermediate container 38f18f07c996
Step 9 : RUN sed -i '/\/etc\/X11\/xinit\/xinitrc-common/a [ -x /usr/bin/
firefox ] && /usr/bin/firefox &' /etc/X11/xinit/xinitrc
 ---> Running in 233e99cab02c
 ---> 421e944ac8b7
Removing intermediate container 233e99cab02c
Step 10 : EXPOSE 5901
 ---> Running in 530cd361cb3c
 ---> 5de01995c156
Removing intermediate container 530cd361cb3c
Step 11 : CMD vncserver -fg
 ---> Running in db89498ae8ce
 ---> 899be39b7feb
Removing intermediate container db89498ae8ce
Successfully built 899be39b7feb
```

How it works...

We start with the base Fedora image, install X Windows System, Firefox, a VNC server, and other packages. We then set up the VNC server to start X Windows System, which will start Firefox.

There's more...

- To start the container, run the following command:

  ```
  $ docker run -it -p 5901:5901 fedora/firefox
  ```

  And give `123456` as the password.

- While running the container, we mapped the `5901` port of the host to `5901` port of the container. In order to connect to the VNC server inside the container, just run the following command from another terminal:

  ```
  $ vncviewer localhost:1
  ```

  Alternatively, from another machine in the network, replace `localhost` with the Docker host's IP address or FQDN.

See also

- Look at the `help` option of `docker build`:

  ```
  $ docker build --help
  ```

- The documentation on the Docker website `https://docs.docker.com/reference/builder/`

Building a WordPress image – a Dockerfile example

So far we have seen the example of running just one service inside a container. If we want to run an application, which requires us to run one or more services simultaneously, then, either we will need to run them on the same container or run them on different containers and link them together. WordPress is one such example that requires a database and web service.

Docker only likes one process per container running in the foreground. Thus, in order to make Docker happy, we have a controlling process that manages the database and web services. The controlling process, in this case, is supervisord (`http://supervisord.org/`). This is a trick we are using to make Docker happy.

Again, we will use a Dockerfile from the Fedora-Dockerfiles repository.

Getting ready

Clone the Fedora-Dockerfiles Git repo using the following command:

```
$ git clone  https://github.com/nkhare/Fedora-Dockerfiles.git
```

Then, go to the `wordpress_single_container` subdirectory:

```
$ cd Fedora-Dockerfiles/systemd/wordpress_single_container
$ cat Dockerfile
FROM fedora
MAINTAINER scollier <scollier@redhat.com>
RUN yum -y update && yum clean all
RUN yum -y install httpd php php-mysql php-gd pwgen supervisor bash-
completion openssh-server psmisc tar && yum clean all
ADD ./start.sh /start.sh
ADD ./foreground.sh /etc/apache2/foreground.sh
ADD ./supervisord.conf /etc/supervisord.conf
RUN echo %sudo  ALL=NOPASSWD: ALL >> /etc/sudoers
ADD http://wordpress.org/latest.tar.gz /wordpress.tar.gz
RUN tar xvzf /wordpress.tar.gz
RUN mv /wordpress/* /var/www/html/.
RUN chown -R apache:apache /var/www/
RUN chmod 755 /start.sh
RUN chmod 755 /etc/apache2/foreground.sh
RUN mkdir /var/run/sshd
EXPOSE 80
EXPOSE 22
CMD ["/bin/bash", "/start.sh"]
```

The supporting files used in the preceding code are explained as follows:

- `foreground.sh`: This is a script to run HTTPS in the foreground.
- `LICENSE`, `LICENSE.txt`, and `UNLICENSE.txt`: These files contain the license information.
- `README.md`: This is a README file.
- `supervisord.conf`: This is a resulting container which will have to run `SSHD`, `MySQL`, and `HTTPD` at the same time. In this particular case, the supervisor is used to manage them. It is a configuration file of the supervisor. More information about this can be found at `http://supervisord.org/`.
- `start.sh`: This is a script to set up MySQL, HTTPD, and to start the supervisor daemon.

How to do it…

```
$ docker build -t fedora/wordpress  .
Sending build context to Docker daemon 41.98 kB
Sending build context to Docker daemon
Step 0 : FROM fedora
 ---> 834629358fe2
Step 1 : MAINTAINER scollier <scollier@redhat.com>
 ---> Using cache
 ---> f21eaf47c9fc
Step 2 : RUN yum -y update && yum clean all
 ---> Using cache
 ---> a8f497a6e57c
Step 3 : RUN yum -y install httpd php php-mysql php-gd pwgen supervisor 
bash-completion openssh-server psmisc tar && yum clean all
 ---> Running in 303234ebf1e1
.... updating/installing packages ....
Cleaning up everything
 ---> cc19a5f5c4aa
```

```
Removing intermediate container 303234ebf1e1
Step 4 : ADD ./start.sh /start.sh
 ---> 3f911077da44
Removing intermediate container c2bd643236ef
Step 5 : ADD ./foreground.sh /etc/apache2/foreground.sh
 ---> 3799902a60c5
Removing intermediate container c99b8e910009
Step 6 : ADD ./supervisord.conf /etc/supervisord.conf
 ---> f232433b8925
Removing intermediate container 0584b945f6f7
Step 7 : RUN echo %sudo  ALL=NOPASSWD: ALL >> /etc/sudoers
 ---> Running in 581db01d7350
 ---> ec686e945dfd
Removing intermediate container 581db01d7350
Step 8 : ADD http://wordpress.org/latest.tar.gz /wordpress.tar.gz
Downloading [=================================================>] 6.186
MB/6.186 MB
 ---> e4e902c389a4
Removing intermediate container 6bfecfbe798d
Step 9 : RUN tar xvzf /wordpress.tar.gz
 ---> Running in cd772500a776
.......... untarring wordpress .........
---> d2c5176228e5
Removing intermediate container cd772500a776
Step 10 : RUN mv /wordpress/* /var/www/html/.
 ---> Running in 7b19abeb509c
 ---> 09400817c55f
Removing intermediate container 7b19abeb509c
Step 11 : RUN chown -R apache:apache /var/www/
 ---> Running in f6b9b6d83b5c
 ---> b35a901735d9
Removing intermediate container f6b9b6d83b5c
Step 12 : RUN chmod 755 /start.sh
 ---> Running in 81718f8d52fa
 ---> 87470a002e12
Removing intermediate container 81718f8d52fa
```

```
Step 13 : RUN chmod 755 /etc/apache2/foreground.sh
 ---> Running in 040c09148e1c
 ---> 1c76f1511685
Removing intermediate container 040c09148e1c
Step 14 : RUN mkdir /var/run/sshd
 ---> Running in 77177a33aee0
 ---> f339dd1f3e6b
Removing intermediate container 77177a33aee0
Step 15 : EXPOSE 80
 ---> Running in f27c0b96d17f
 ---> 6078f0d7b70b
Removing intermediate container f27c0b96d17f
Step 16 : EXPOSE 22
 ---> Running in eb7c7d90b860
 ---> 38f36e5c7cab
Removing intermediate container eb7c7d90b860
Step 17 : CMD /bin/bash /start.sh
 ---> Running in 5635fe4783da
 ---> c1a327532355
Removing intermediate container 5635fe4783da
Successfully built c1a327532355
```

How it works...

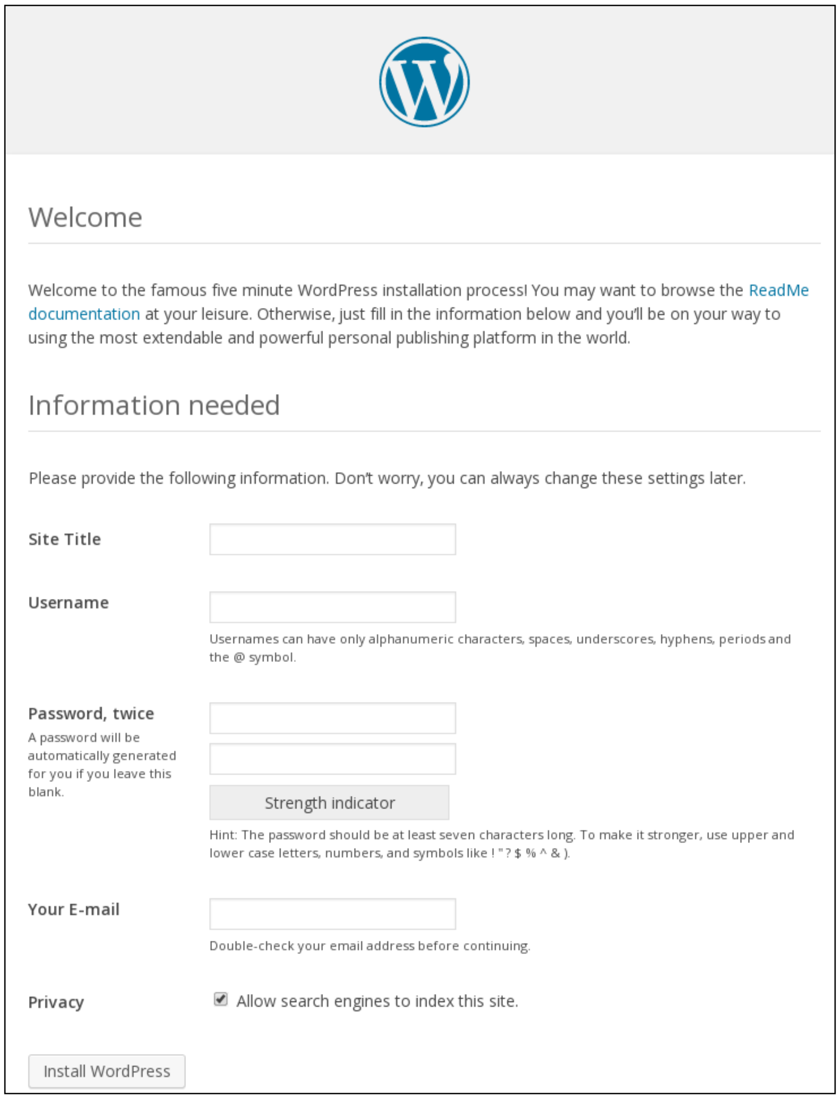
As with the other recipes, we start with the base image, install the required packages, and copy the supporting files. We will then set up `sudo`, `download`, and `untar` WordPress inside the HTTPD document root. After this, we expose the ports and run the start.sh scripts, which sets up MySQL, WordPress, HTTPS permissions and gives control to supervisord. In the `supervisord.conf`, you will see entries, such as the following services that supervisord manages:

```
[program:mysqld]
command=/usr/bin/mysqld_safe
[program:httpd]
command=/etc/apache2/foreground.sh
stopsignal=6
[program:sshd]
command=/usr/sbin/sshd -D
```

```
stdout_logfile=/var/log/supervisor/%(program_name)s.log
stderr_logfile=/var/log/supervisor/%(program_name)s.log
autorestart=true
```

There's more...

- Start the container, get its IP address and open it through a web browser. You should see the Welcome screen, as shown in the following screenshot, after doing the language selection:



Welcome

Welcome to the famous five minute WordPress installation process! You may want to browse the ReadMe documentation at your leisure. Otherwise, just fill in the information below and you'll be on your way to using the most extendable and powerful personal publishing platform in the world.

Information needed

Please provide the following information. Don't worry, you can always change these settings later.

Site Title

Username

Usernames can have only alphanumeric characters, spaces, underscores, hyphens, periods and the @ symbol.

Password, twice

A password will be automatically generated for you if you leave this blank.

Strength indicator

Hint: The password should be at least seven characters long. To make it stronger, use upper and lower case letters, numbers, and symbols like ! " ? $ % ^ &).

Your E-mail

Double-check your email address before continuing.

Privacy

✓ Allow search engines to index this site.

Install WordPress

- It is now possible to run systemd inside the container, which is a more preferred way. Systemd can manage more than one service .You can look at the example of systemd at `https://github.com/fedora-cloud/Fedora-Dockerfiles/tree/master/systemd`.

See also

- Look at the `help` option of `docker build`:

  ```
  $ docker build --help
  ```

- The documentation on the Docker website `https://docs.docker.com/reference/builder/`

Setting up a private index/registry

As we saw earlier, the public Docker registry is the available Docker Hub (`https://registry.hub.docker.com/`) through which users can push/pull images. We can also host a private registry either on a local environment or on the cloud. There are a few ways to set up the local registry:

- Use the Docker registry from Docker Hub
- Build an image from Dockerfile and run a registry container:

  `https://github.com/fedora-cloud/Fedora-Dockerfiles/tree/master/registry`

- Configure the distribution-specific package such as Fedora, which provides the docker-registry package that you can install and configure.

The easiest way to set it up is through the registry container itself.

Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client.

How to do it...

1. To run the registry on the container, run the following command:

   ```
   $ docker run -p 5000:5000 registry
   ```

2. To test the newly created registry, perform the following steps:

1. Start a container and its ID by using the following command:

   ```
   $ ID='docker run -d -i fedora /bin/bash'
   ```

2. If needed, attach to the newly created container and make some changes. Then, commit those changes to the local repository:

   ```
   $ docker commit $ID fedora-20
   ```

3. To push the image to the local registry, we need to tag the image with the hostname or IP address of the registry host. Let's say our registry host is `registry-host`; then, to tag it, use the following command:

   ```
   $ docker tag fedora-20 registry-host:5000/nkhare/f20
   ```

4. As we have not configured HTTPS correctly while starting the registry, we will get an error such as the `ping attempt failed with error: Get https://dockerhost:5000/v1/_ping`, which is expected. For our example to work, we need to add the `--insecure-registry registry-host:5000` option to the daemon. If you have started the Docker daemon manually, then we have to run the command as follows to allow insecure registry:

   ```
   $ docker -d   --insecure-registry registry-host:5000
   ```

5. To push the image, use the following command:

   ```
   $ docker push registry-host:5000/nkhare/f20
   ```

6. To pull the image from the local registry, run the following command:

   ```
   $ docker pull registry-host:5000/nkhare/f20
   ```

How it works...

The preceding command to pull the image will download the official registry image from Docker Hub and run it on port `5000`. The `-p` option publishes the container port to the host system's port. We will look at the details about port publishing in the next chapter.

The registry can also be configured on any existing servers using the docker-registry app. The steps to do this are available at the docker-registry GitHub page:

`https://github.com/docker/docker-registry`

There's more...

Let's look at Dockerfile of docker-registry to understand how the registry image is being created and how to set different configuration options:

```
# VERSION 0.1
# DOCKER-VERSION  0.7.3
```

```
# AUTHOR:          Sam Alba <sam@docker.com>
# DESCRIPTION:     Image with docker-registry project and dependencies
# TO_BUILD:        docker build -rm -t registry .
# TO_RUN:          docker run -p 5000:5000 registry

# Latest Ubuntu LTS
FROM ubuntu:14.04

# Update
RUN apt-get update \
# Install pip
    && apt-get install -y \
        swig \
        python-pip \
# Install deps for backports.lzma (python2 requires it)
        python-dev \
        python-mysqldb \
        python-rsa \
        libssl-dev \
        liblzma-dev \
        libevent1-dev \
    && rm -rf /var/lib/apt/lists/*

COPY . /docker-registry
COPY ./config/boto.cfg /etc/boto.cfg

# Install core
RUN pip install /docker-registry/depends/docker-registry-core

# Install registry
RUN pip install file:///docker-registry#egg=docker-
registry[bugsnag,newrelic,cors]

RUN patch \
 $(python -c 'import boto; import os; print
os.path.dirname(boto.__file__)')/connection.py \
 < /docker-registry/contrib/boto_header_patch.diff

ENV DOCKER_REGISTRY_CONFIG /docker-registry/config/config_sample.yml
ENV SETTINGS_FLAVOR dev
EXPOSE 5000
CMD ["docker-registry"]
```

With the preceding Dockerfile, we will:

- Take Ubuntu's base image install/update packages
- Copy the docker-registry source code inside the image
- Use the `pip install` docker-registry
- Set up the configuration file to use while running the registry using the environment variable
- Set up the flavor to use while running the registry using the environment variable
- Expose port `5000`
- Run the registry executable

Flavors in the configuration file (`/docker-registry/config/config_sample.yml`) provide different ways to configure the registry. With the preceding Dockerfile, we will set the `dev` flavor using the environment variables. The different types of flavors are:

- `common`: This is used by all the other flavors as base settings
- `local`: This stores data on the local filesystem
- `s3`: This stores data in an AWS S3 bucket
- `dev`: This is the basic configuration using the local flavors
- `test`: This is used by unit tests
- `prod`: This is the production configuration (basically a synonym for the S3 flavor)
- `gcs`: This stores data in Google cloud storage
- `swift`: This stores data in OpenStack Swift
- `glance`: This stores data in OpenStack Glance, with a fallback to the local storage
- `glance-swift`: This stores data in OpenStack Glance, with a fallback to Swift
- `elliptics`: This stores data in Elliptics key-value storage

For each of preceding flavors, different configuration options such as loglevel, authentication, and so on are available. The documentation for all of the options are available on the GitHub page of docker-registry, which I mentioned earlier.

See also

- The documentation on GitHub `https://github.com/docker/docker-registry`

Automated builds – with GitHub and Bitbucket

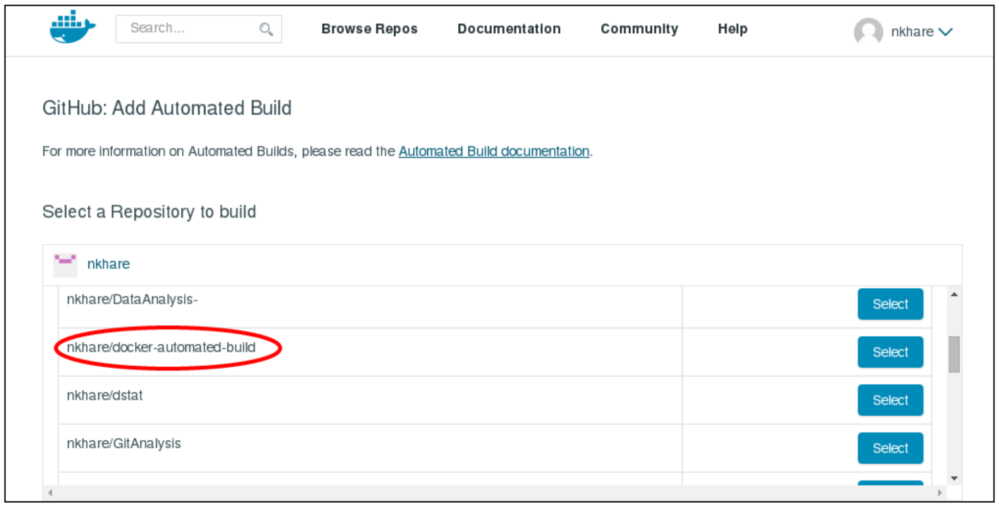
We have seen earlier how to push the Docker images to Docker Hub. Docker Hub allows us to create automated images from a GitHub/Bitbucket repository using its build clusters. The GitHub/Bitbucket repository should contain the Dockerfile and the content required to copy/add inside the image. Let's look at a GitHub example in the upcoming sections.

Getting ready

You will need an account on Docker Hub and GitHub. You will also need a GitHub repository with a corresponding Dockerfile at the top level.

How to do it…

1. Log in to Docker Hub (`https://hub.docker.com/`) and click on the green plus sign. Add the Repository icon on the top right-hand side corner and click on **Automated Build**. Select GitHub as a source to use for automated build. Then, select the **Public and Private (recommended)** option to connect to GitHub. Provide the GitHub username/password when prompted. Select the GitHub repository to perform automated build.



GitHub: Add Automated Build

For more information on Automated Builds, please read the Automated Build documentation.

Select a Repository to build

nkhare

nkhare/DataAnalysis- Select

nkhare/docker-automated-build Select

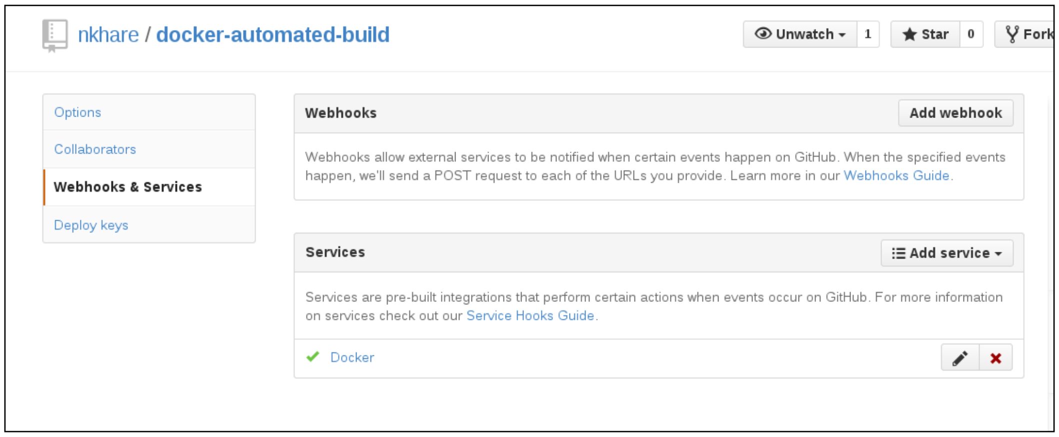
nkhare/dstat Select

nkhare/GitAnalysis Select

2. After selecting the GitHub repository, it will ask you to pick its branch to use for automated build. It will also ask for a tag name to use after the image it automatically built. By default, the latest tag name will be used. Then, click on the **Save and trigger build** button to start the automated build process. That's it!! Your build is now submitted. You can click on the build status to check the status of the build.

How it works…

When we select a GitHub repository for automated build, GitHub enables the Docker service for that repository. You can look at the **Settings** section of the GitHub repository for more configuration. Whenever we make any changes to this GitHub repository, such as commits, an automated build gets triggered using the Dockerfile that resides in the GitHub repository.

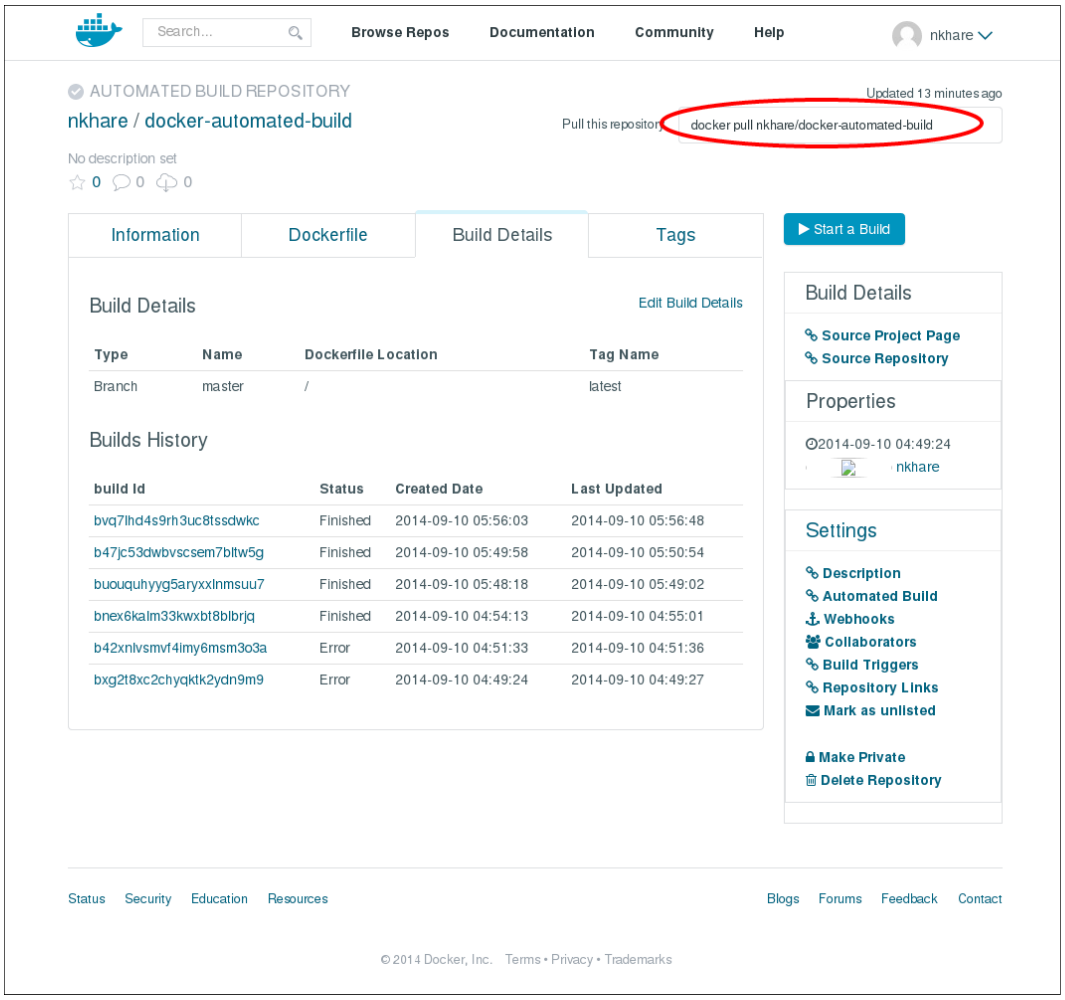


Webhooks allow external services to be notified when certain events happen on GitHub. When the specified events happen, we'll send a POST request to each of the URLs you provide. Learn more in our Webhooks Guide.

Services are pre-built integrations that perform certain actions when events occur on GitHub. For more information on services check out our Service Hooks Guide.

There's more...

You can get the details such as the Dockerfile, build details tags, and other information, by going to the **Your Repositories** section. It also has the details of how to pull your image:



AUTOMATED BUILD REPOSITORY

nkhare / docker-automated-build

Updated 13 minutes ago

Pull this repository docker pull nkhare/docker-automated-build

No description set

☆ 0 0 0

Build Details

Edit Build Details

Type	Name	Dockerfile Location	Tag Name
Branch	master	/	latest

Builds History

build id	Status	Created Date	Last Updated
bvq7lhd4s9rh3uc8tssdwkc	Finished	2014-09-10 05:56:03	2014-09-10 05:56:48
b47jc53dwbvscsem7bltw5g	Finished	2014-09-10 05:49:58	2014-09-10 05:50:54
buouquhyyg5aryxxlnmsuu7	Finished	2014-09-10 05:48:18	2014-09-10 05:49:02
bnex6kalm33kwxbt8blbrjq	Finished	2014-09-10 04:54:13	2014-09-10 04:55:01
b42xnlvsmvf4imy6msm3o3a	Error	2014-09-10 04:51:33	2014-09-10 04:51:36
bxg2t8xc2chyqktk2ydn9m9	Error	2014-09-10 04:49:24	2014-09-10 04:49:27

Build Details

Source Project Page
Source Repository

Properties

2014-09-10 04:49:24
nkhare

Settings

Description
Automated Build
Webhooks
Collaborators
Build Triggers
Repository Links
Mark as unlisted

Make Private
Delete Repository

The images that get created using the automated build process cannot be pushed through the `docker push` command.

You can change the settings in the **Webhooks & Services** section of the repository on GitHub to unregister the Docker service. This will stop doing the automated builds.

See also

- The steps for setting up automated build with Bitbucket are almost identical. The hook for automated build gets configured under the **Hooks** section of Bitbucket repository's **Settings** section.
- The documentation on the Docker website `https://docs.docker.com/docker-hub/builds/`

Creating the base image – using supermin

Earlier in this chapter, we used the `FROM` instruction to pick the base image to start with. The image we create can become the base image to containerize another application and so on. From the very beginning to this chain, we will have a base image from the underlying Linux distribution that we want to use such as Fedora, Ubuntu, CentOS, and so on.

To build such a base image, we will need to have a distribution-specific base system installed into a directory, which can then be imported as an image to Docker. With chroot utility, we can fake a directory as the root filesystem and then put all the necessary files inside it before importing it as a Docker image. Supermin and Debootstrap are the kind of tools that can help us make the preceding process easier.

Supermin is a tool to build supermin appliances. These are tiny appliances, which get fully instantiated on the fly. Earlier this program was called febootstrap.

Getting ready

Install supermin on the system where you want to build the base image. You can install supermin on Fedora with the following command:

```
$ yum install supermin
```

How to do it...

1. Using the `prepare` mode install `bash`, `coreutils`, and the related dependencies inside a directory.

   ```
   $ supermin --prepare -o OUTPUTDIR PACKAGE [PACKAGE ...]
   ```

   Here's an example using the preceding syntax:

   ```
   $ supermin --prepare bash coreutils -o f21_base
   ```

2. Now, with the `build` mode, create a chrooted environment for the base image:

   ```
   $ supermin --build -o OUTPUTDIR -f chroot|ext2 INPUT [INPUT ...]
   ```

Here's an example using the preceding syntax :

```
$ supermin --build --format chroot f21_base -o f21_image
```

3. If we do `ls` on the output directory, we will see a directory tree similar to any Linux root filesystem:

```
$ ls f21_image/
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root
run  sbin  srv  sys  tmp  usr  var
```

4. Now we can export the directory as a Docker image with the following command:

```
$ tar -C f21_image/ -c . | docker import - nkhare/f21_base
d6db8b798dee30ad9c84480ef7497222f063936a398ecf639e60599eed7f6560
```

5. Now, look at the `docker images` output. You should have a new image with `nkhare/f21_base` as the name.

How it works…

Supermins has two modes, `prepare` and `build`. With the `prepare` mode, it just puts all the requested packages with their dependencies inside a directory without copying the host OS specific files.

With the `build` mode, the previously created supermin appliance from the `prepare` mode gets converted into a full blown bootable appliance with all the necessary files. This step will copy the required files/binaries from the host machine to the appliance directory, so the packages must be installed on the host machines that you want to use in the appliance.

The `build` mode has two output formats, chroot, and ext2. With the chroot format, the directory tree gets written into the directory, and with the ext2 format, a disk image gets created. We exported the directory created through the chroot format to create the Docker image.

There's more…

Supermin is not specific to Fedora and should work on any Linux distribution.

See also

- Look at the `man` page of supermin for more information using the following command:

```
$ man supermin
```

- The online documentation `http://people.redhat.com/~rjones/supermin/`
- The GitHub repository `https://github.com/libguestfs/supermin`

Creating the base image – using Debootstrap

Debootstrap is a tool to install a Debian-based system into a directory of an already installed system.

Getting ready

Install `debootstrap` on the Debian-based system using the following command:

```
$ apt-get install debootstrap
```

How to do it...

The following command can be used to create the base image using Debootstrap:

```
$ debootstrap [OPTION...]  SUITE TARGET [MIRROR [SCRIPT]]
```

`SUITE` refers to the release code name and `MIRROR` is the respective repository. If you wanted to create the base image of Ubuntu 14.04.1 LTS (Trusty Tahr), then do the following:

1. Create a directory on which you want to install the OS. Debootstrap also creates the chroot environment to install a package, as we saw earlier with supermin.

   ```
   $ mkdir trusty_chroot
   ```

2. Now, using `debootstrap`, install Trusty Tahr inside the directory we created earlier:

   ```
   $ debootstrap trusty ./trusty_chroot
   http://in.archive.ubuntu.com/ubuntu/
   ```

3. You will see the directory tree similar to any Linux root filesystem, inside the directory in which Trusty Tahr is installed.

   ```
   $ ls ./trusty_chroot
   bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc
   root  run  sbin  srv  sys  tmp  usr  var
   ```

4. Now we can export the directory as a Docker image with the following command:

   ```
   $ tar -C trusty_chroot/ -c . |  docker import -
   nkhare/trusty_base
   ```

5. Now, look at the `docker images` output. You should have a new image with `nkhare/trusty_base` as the name.

See also

- The Debootstrap wiki page `https://wiki.debian.org/Debootstrap`.
- There are a few other ways to create base images. You can find links to them at `https://docs.docker.com/articles/baseimages/`.

Visualizing dependencies between layers

As the number of images grows, it becomes difficult to find relation between them. There are a few utilities for which you can find the relation between images.

Getting ready

One or more Docker images on the host running the Docker daemon.

How to do it...

1. Run the following command to get a tree-like view of the images:

   ```
   $ docker images -t
   ```

How it works...

The dependencies between layers will be fetched from the metadata of the Docker images.

There's more...

From `--viz` to `docker images`, we can see dependencies graphically; to do this, you will need to have the `graphviz` package installed:

```
$ docker images --viz | dot -Tpng -o /tmp/docker.png
$ display /tmp/docker.png
```

As it states in the warning that appears when running the preceding commands, the `-t` and `--viz` options might get deprecated soon.

See also

- The following project tries to visualize Docker data as well by using raw JSON output from Docker `https://github.com/justone/dockviz`

4
Network and Data Management for Containers

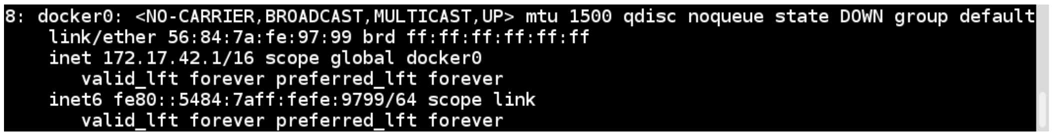
In this chapter, we will cover the following recipes:

- Accessing containers from outside
- Managing data in containers
- Linking two or more containers
- Developing a LAMP application by linking containers
- Networking of multihost container with Flannel
- Assigning IPv6 addresses to containers

Introduction

Until now, we have worked with a single container and accessed it locally. But as we move to more real world use cases, we will need to access the container from the outside world, share external storage within the container, communicate with containers running on other hosts, and so on. In this chapter, we'll see how to fulfill some of those requirements. Let's start by understanding Docker's default networking setup and then go to advanced use cases.

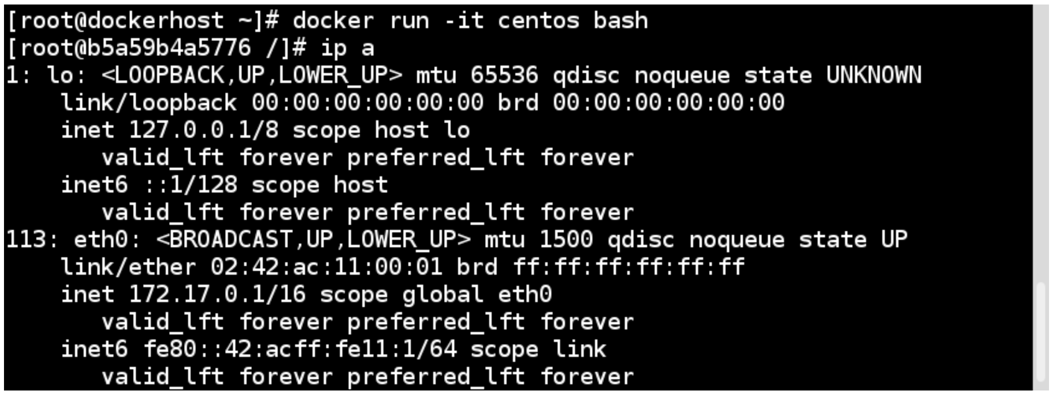
When the Docker daemon starts, it creates a virtual Ethernet bridge with the name `docker0`. For example, we will see the following with the `ip addr` command on the system that runs the Docker daemon:



```
8: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
    inet 172.17.42.1/16 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::5484:7aff:fefe:9799/64 scope link
       valid_lft forever preferred_lft forever
```

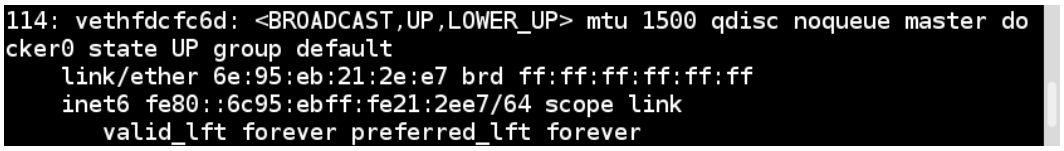
As we can see, `docker0` has the IP address 172.17.42.1/16. Docker randomly chooses an address and subnet from a private range defined in RFC 1918 (`https://tools.ietf.org/html/rfc1918`). Using this bridged interface, containers can communicate with each other and with the host system.

By default, every time Docker starts a container, it creates a pair of virtual interfaces, one end of which is attached to the host system and other end to the created container. Let's start a container and see what happens:



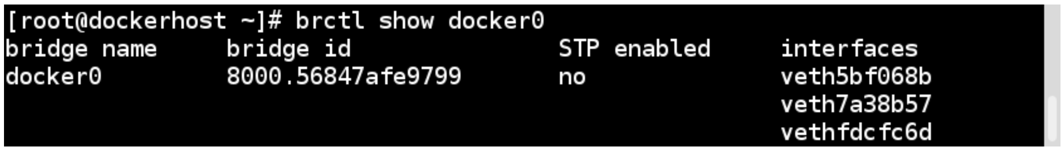
```
[root@dockerhost ~]# docker run -it centos bash
[root@b5a59b4a5776 /]# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
113: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:01 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:1/64 scope link
       valid_lft forever preferred_lft forever
```

The end that is attached to the `eth0` interface of the container gets the 172.17.0.1/16 IP address. We also see the following entry for the other end of the interface on the host system:



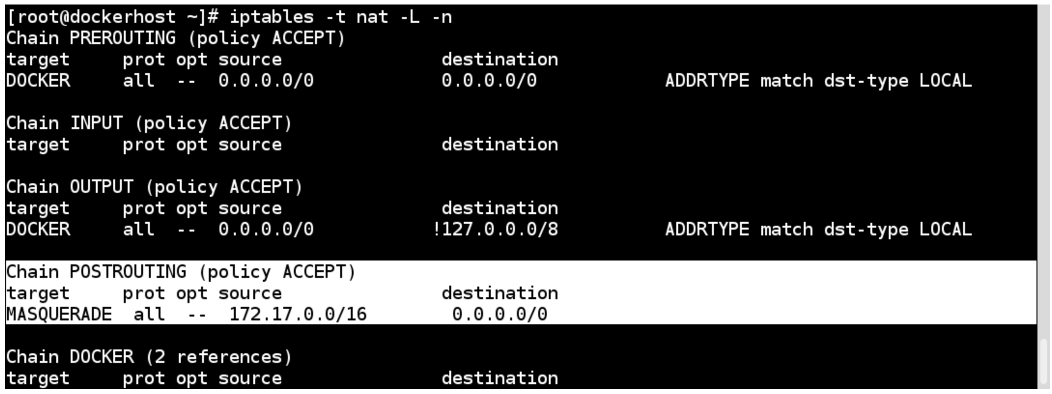
```
114: vethfdcfc6d: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master do
cker0 state UP group default
    link/ether 6e:95:eb:21:2e:e7 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::6c95:ebff:fe21:2ee7/64 scope link
       valid_lft forever preferred_lft forever
```

Now, let's create a few more containers and look at the `docker0` bridge with the `brctl` command, which manages Ethernet bridges:



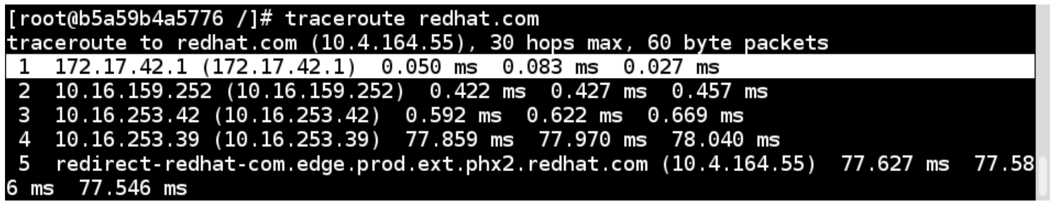
```
[root@dockerhost ~]# brctl show docker0
bridge name     bridge id               STP enabled     interfaces
docker0         8000.56847afe9799       no              veth5bf068b
                                                        veth7a38b57
                                                        vethfdcfc6d
```

Every veth* binds to the `docker0` bridge, which creates a virtual subnet shared between the host and every Docker container. Apart from setting up the `docker0` bridge, Docker creates IPtables NAT rules, such that all containers can talk to the external world by default but not the other way around. Let's look at the NAT rules on the Docker host:



```
[root@dockerhost ~]# iptables -t nat -L -n
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  0.0.0.0/0            0.0.0.0/0            ADDRTYPE match dst-type LOCAL

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  0.0.0.0/0           !127.0.0.0/8          ADDRTYPE match dst-type LOCAL

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  172.17.0.0/16        0.0.0.0/0

Chain DOCKER (2 references)
target     prot opt source               destination
```

If we try to connect to the external world from a container, we will have to go through the Docker bridge that was created by default:



```
[root@b5a59b4a5776 /]# traceroute redhat.com
traceroute to redhat.com (10.4.164.55), 30 hops max, 60 byte packets
 1  172.17.42.1 (172.17.42.1)  0.050 ms  0.083 ms  0.027 ms
 2  10.16.159.252 (10.16.159.252)  0.422 ms  0.427 ms  0.457 ms
 3  10.16.253.42 (10.16.253.42)  0.592 ms  0.622 ms  0.669 ms
 4  10.16.253.39 (10.16.253.39)  77.859 ms  77.970 ms  78.040 ms
 5  redirect-redhat-com.edge.prod.ext.phx2.redhat.com (10.4.164.55)  77.627 ms  77.58
6 ms  77.546 ms
```

Later in this chapter, we will see how the external world can connect to a container.

When starting a container, we have a few modes to select its networking:

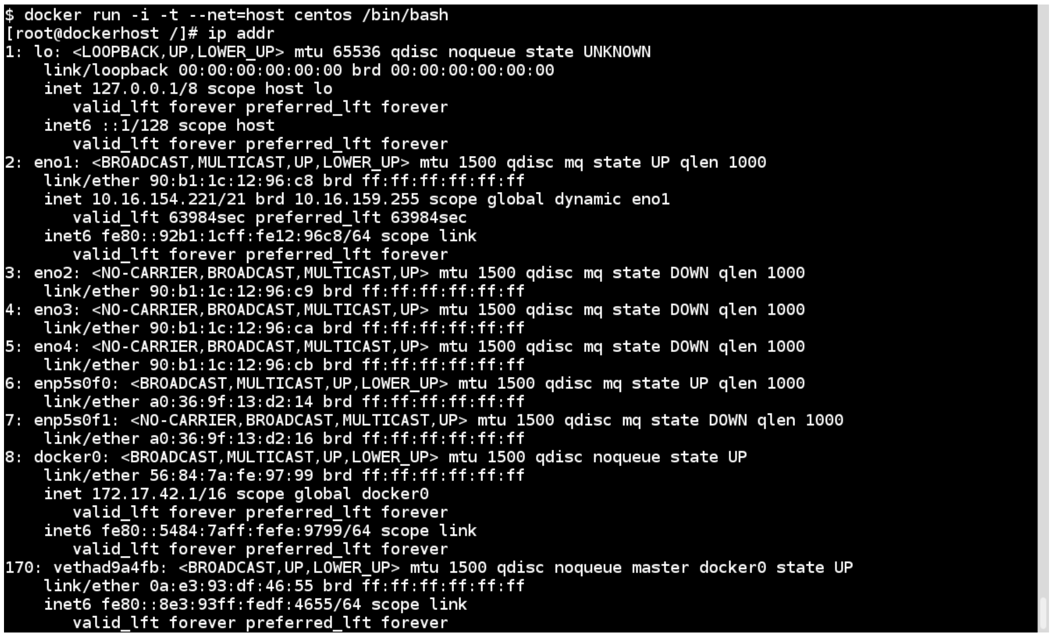
- `--net=bridge`: This is the default mode that we just saw. So, the preceding command that we used to start the container can be written as follows:

  ```
  $ docker run -i -t --net=bridge centos /bin/bash
  ```

- `--net=host`: With this option, Docker does not create a network namespace for the container; instead, the container will network stack with the host. So, we can start the container with this option as follows:

  `$ docker run -i -t  --net=host centos bash`

  We can then run the `ip addr` command within the container as seen here:

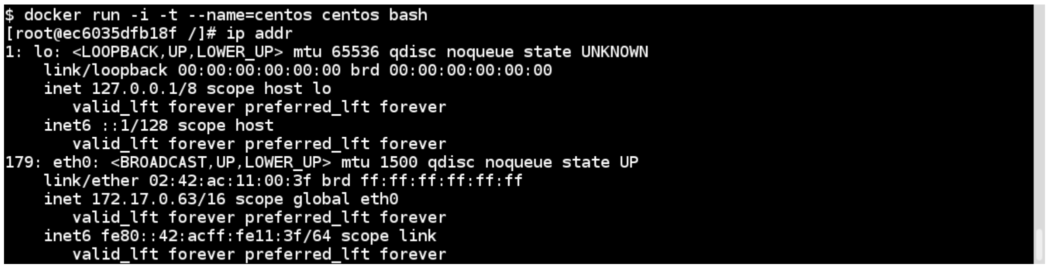


```
$ docker run -i -t --net=host centos /bin/bash
[root@dockerhost /]# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eno1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP qlen 1000
    link/ether 90:b1:1c:12:96:c8 brd ff:ff:ff:ff:ff:ff
    inet 10.16.154.221/21 brd 10.16.159.255 scope global dynamic eno1
       valid_lft 63984sec preferred_lft 63984sec
    inet6 fe80::92b1:1cff:fe12:96c8/64 scope link
       valid_lft forever preferred_lft forever
3: eno2: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc mq state DOWN qlen 1000
    link/ether 90:b1:1c:12:96:c9 brd ff:ff:ff:ff:ff:ff
4: eno3: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc mq state DOWN qlen 1000
    link/ether 90:b1:1c:12:96:ca brd ff:ff:ff:ff:ff:ff
5: eno4: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc mq state DOWN qlen 1000
    link/ether 90:b1:1c:12:96:cb brd ff:ff:ff:ff:ff:ff
6: enp5s0f0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP qlen 1000
    link/ether a0:36:9f:13:d2:14 brd ff:ff:ff:ff:ff:ff
7: enp5s0f1: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc mq state DOWN qlen 1000
    link/ether a0:36:9f:13:d2:16 brd ff:ff:ff:ff:ff:ff
8: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
    inet 172.17.42.1/16 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::5484:7aff:fefe:9799/64 scope link
       valid_lft forever preferred_lft forever
170: vethad9a4fb: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP
    link/ether 0a:e3:93:df:46:55 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::8e3:93ff:fedf:4655/64 scope link
       valid_lft forever preferred_lft forever
```

  We can see all the network devices attached to the host. An example of using such a configuration is to run the `nginx` reverse proxy within a container to serve the web applications running on the host.

- `--net=container:NAME_or_ID`: With this option, Docker does not create a new network namespace while starting the container but shares it from another container. Let's start the first container and look for its IP address:

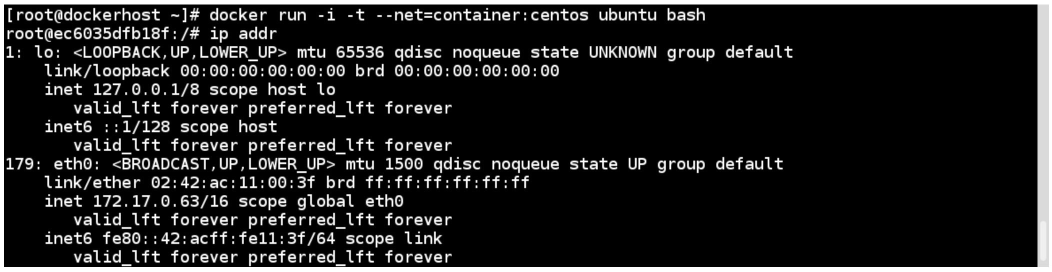
  `$ docker run -i -t --name=centos centos bash`



```
$ docker run -i -t --name=centos centos bash
[root@ec6035dfb18f /]# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
179: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:3f brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.63/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:3f/64 scope link
       valid_lft forever preferred_lft forever
```

Now start another as follows:

```
$ docker run -i -t --net=container:centos ubuntu bash
```



```
[root@dockerhost ~]# docker run -i -t --net=container:centos ubuntu bash
root@ec6035dfb18f:/# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
179: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:11:00:3f brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.63/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:3f/64 scope link
       valid_lft forever preferred_lft forever
```

As we can see, both containers contain the same IP address.

Containers in a Kubernetes (`http://kubernetes.io/`) Pod use this trick to connect with each other. We will revisit this in *Chapter 8*, *Docker Orchestration and Hosting Platforms*.

- `--net=none`: With this option, Docker creates the network namespace inside the container but does not configure networking.

For more information about the different networking we discussed in the preceding section, visit `https://docs.docker.com/articles/networking/#how-docker-networks-a-container`.

From Docker 1.2 onwards, it is also possible to change `/etc/host`, `/etc/hostname`, and `/etc/resolv.conf` on a running container. However, note that these are just used to run a container. If it restarts, we will have to make the changes again.

So far, we have looked at networking on a single host, but in the real world, we would like to connect multiple hosts and have a container from one host to talk to a container from another host. Flannel (`https://github.com/coreos/flannel`), Weave (`https://github.com/weaveworks/weave`), Calio (`http://www.projectcalico.org/getting-started/docker/`), and Socketplane (`http://socketplane.io/`) are some solutions that offer this functionality. Later in this chapter, we will see how to configure Flannel to multihost networking. Socketplane joined Docker Inc in March '15.

Community and Docker are building a **Container Network Model** (**CNM**) with libnetwork (`https://github.com/docker/libnetwork`), which provides a native Go implementation to connect containers. More information on this development can be found at `http://blog.docker.com/2015/04/docker-networking-takes-a-step-in-the-right-direction-2/`.

Accessing containers from outside

Once the container is up, we would like to access it from outside. If you have started the container with the `--net=host` option, then it can be accessed through the Docker host IP. With `--net=none`, you can attach the network interface from the public end or through other complex settings. Let's see what happens in by default—where packets are forwarded from the host network interface to the container.

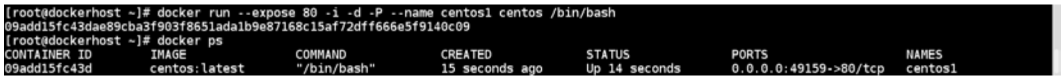
Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client.

How to do it…

1. Let's start a container with the `-P` option:

    ```
    $ docker run --expose 80 -i -d -P --name f20 fedora /bin/bash
    ```



```
[root@dockerhost ~]# docker run --expose 80 -i -d -P --name centos1 centos /bin/bash
09add15fc43dae89cba3f903f8651ada1b9e87168c15af72dff666e5f9140c09
[root@dockerhost ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS                   NAMES
09add15fc43d        centos:latest       "/bin/bash"         15 seconds ago      Up 14 seconds       0.0.0.0:49159->80/tcp   centos1
```

    This automatically maps any network port of the container to a random high port of the Docker host between 49000 to 49900.

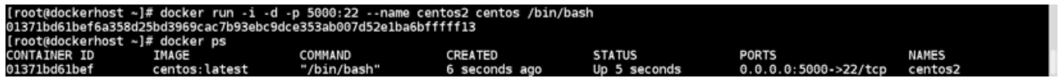
    In the `PORTS` section, we see `0.0.0.0:49159->80/tcp`, which is of the following form:

    ```
    <Host Interface>:<Host Port> -> <Container
    Interface>/<protocol>
    ```

    So, in case any request comes on port `49159` from any interface on the Docker host, the request will be forwarded to port `80` of the `centos1` container.

    We can also map a specific port of the container to the specific port of the host using the `-p` option:

    ```
    $  docker run -i -d -p 5000:22 --name centos2 centos /bin/bash
    ```

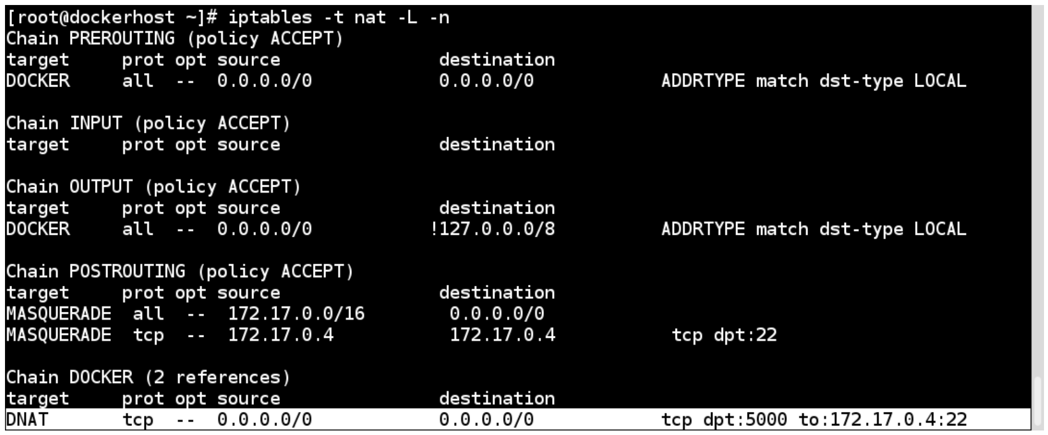


```
[root@dockerhost ~]# docker run -i -d -p 5000:22 --name centos2 centos /bin/bash
01371bd61bef6a358d25bd3969cac7b93ebc9dce353ab007d52e1ba6bfffff13
[root@dockerhost ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS                  NAMES
01371bd61bef        centos:latest       "/bin/bash"         6 seconds ago       Up 5 seconds        0.0.0.0:5000->22/tcp   centos2
```

In this case, all requests coming on port `5000` from any interface on the Docker host will be forwarded to port `22` of the `centos2` container.

How it works...

With the default configuration, Docker sets up the firewall rule to forward the connection from the host to the container and enables IP forwarding on the Docker host:



```
[root@dockerhost ~]# iptables -t nat -L -n
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  0.0.0.0/0            0.0.0.0/0            ADDRTYPE match dst-type LOCAL

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  0.0.0.0/0           !127.0.0.0/8          ADDRTYPE match dst-type LOCAL

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  172.17.0.0/16        0.0.0.0/0
MASQUERADE  tcp  --  172.17.0.4           172.17.0.4           tcp dpt:22

Chain DOCKER (2 references)
target     prot opt source               destination
DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0            tcp dpt:5000 to:172.17.0.4:22
```

As we can see from the preceding example, a `DNAT` rule has been set up to forward all traffic on port `5000` of the host to port `22` of the container.

There's more...

By default, with the `-p` option, Docker will forward all the requests coming to any interface to the host. To bind to a specific interface, we can specify something like the following:

```
$ docker run -i -d -p 192.168.1.10:5000:22 --name f20 fedora /bin/bash
```

In this case, only requests coming to port `5000` on the interface that has the IP `192.168.1.10` on the Docker host will be forwarded to port `22` of the `f20` container. To map port `22` of the container to the dynamic port of the host, we can run following command:

```
$ docker run -i -d -p 192.168.1.10::22 --name f20 fedora /bin/bash
```

We can bind multiple ports on containers to ports on hosts as follows:

```
$  docker run -d -i -p 5000:22 -p 8080:80 --name f20 fedora /bin/bash
```

We can look up the public-facing port that is mapped to the container's port as follows:

```
$ docker port f20 80
0.0.0.0:8080
```

To look at all the network settings of a container, we can run the following command:

```
$ docker inspect  -f "{{ .NetworkSettings }}" f20
```

See also

- Networking documentation on the Docker website at `https://docs.docker.com/articles/networking/`.

Managing data in containers

Any uncommitted data or changes in containers get lost as soon as containers are deleted. For example, if you have configured the Docker registry in a container and pushed some images, as soon as the registry container is deleted, all of those images will get lost if you have not committed them. Even if you commit, it is not the best practice. We should try to keep containers as light as possible. The following are two primary ways to manage data with Docker:

- **Data volumes**: From the Docker documentation (`https://docs.docker.com/userguide/dockervolumes/`), a data volume is a specially-designated directory within one or more containers that bypasses the Union filesystem to provide several useful features for persistent or shared data:
  - Volumes are initialized when a container is created. If the container's base image contains data at the specified mount point, that data is copied into the new volume.
  - Data volumes can be shared and reused between containers.
  - Changes to a data volume are made directly.
  - Changes to a data volume will not be included when you update an image.
  - Volumes persist until no containers use them.
- **Data volume containers**: As a volume persists until no container uses it, we can use the volume to share persistent data between containers. So, we can create a named volume container and mount the data to another container.

Getting ready

Make sure that the Docker daemon is running on the host and you can connect through the Docker client.

How to do it...

1. Add a data volume. With the `-v` option with the `docker run` command, we add a data volume to the container:

   ```
   $ docker run -t -d -P -v /data --name f20 fedora /bin/bash
   ```

We can have multiple data volumes within a container, which can be created by adding `-v` multiple times:

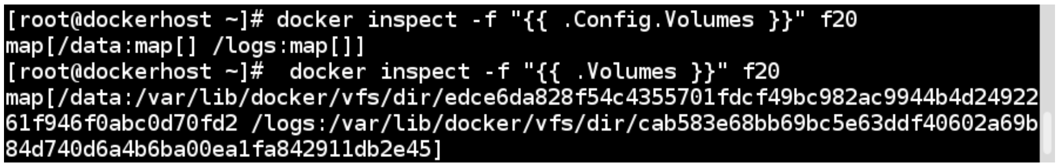
```
$ docker run -t -d -P -v /data -v /logs --name f20 fedora
/bin/bash
```

The `VOLUME` instruction can be used in a Dockerfile to add data volume as well by adding something similar to `VOLUME ["/data"]`.

We can use the `inspect` command to look at the data volume details of a container:

```
$ docker inspect -f "{{ .Config.Volumes }}" f20
```

```
$ docker inspect -f "{{ .Volumes }}" f20
```



```
[root@dockerhost ~]# docker inspect -f "{{ .Config.Volumes }}" f20
map[/data:map[] /logs:map[]]
[root@dockerhost ~]#  docker inspect -f "{{ .Volumes }}" f20
map[/data:/var/lib/docker/vfs/dir/edce6da828f54c4355701fdcf49bc982ac9944b4d24922
61f946f0abc0d70fd2 /logs:/var/lib/docker/vfs/dir/cab583e68bb69bc5e63ddf40602a69b
84d740d6a4b6ba00ea1fa842911db2e45]
```

If the target directory is not there within the container, it will be created.

2. Next, we mount a host directory as a data volume. We can also map a host directory to a data volume with the `-v` option:

   ```
   $ docker run -i -t -v
   /source_on_host:/destination_on_container fedora /bin/bash
   ```

   Consider the following example:

   ```
   $ docker run -i -t -v /srv:/mnt/code fedora /bin/bash
   ```

   This can be very useful in cases such as testing code in different environments, collecting logs in central locations, and so on. We can also map the host directory in read-only mode as follows:

   ```
   $ docker run -i -t -v /srv:/mnt/code:ro fedora /bin/bash
   ```

   We can also mount the entire root filesystem of the host within the container with the following command:

   ```
   $ docker run -i -t -v /:/host:ro fedora /bin/bash
   ```

If the directory on the host (`/srv`) does not exist, then it will be created, given that you have permission to create one. Also, on the Docker host where SELinux is enabled and if the Docker daemon is configured to use SELinux (`docker -d --selinux-enabled`), you will see the `permission denied` error if you try to access files on mounted volumes until you relabel them. To relabel them, use either of the following commands:

```
$ docker run -i -t -v /srv:/mnt/code:z fedora /bin/bash
$ docker run -i -t -v /srv:/mnt/code:Z fedora /bin/bash
```

Please visit *Chapter 9*, *Docker Security*, for more detail.

3. Now, create a data volume container. While sharing the host directory to a container through volume, we are binding the container to a given host, which is not good. Also, the storage in this case is not controlled by Docker. So, in cases when we want data to be persisted even if we update the containers, we can get help from data volume containers. Data volume containers are used to create a volume and nothing else; they do not even run. As the created volume is attached to a container (not running), it cannot be deleted. For example, here's a named data container:

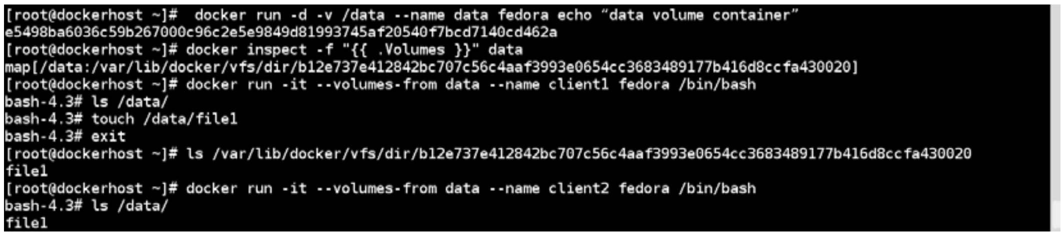
```
$ docker run -d -v /data --name data fedora echo "data volume
container"
```

This will just create a volume that will be mapped to a directory managed by Docker. Now, other containers can mount the volume from the data container using the `--volumes-from` option as follows:

```
$ docker run  -d -i -t --volumes-from data --name client1
fedora /bin/bash
```

We can mount a volume from the data volume container to multiple containers:

```
$ docker run  -d -i -t --volumes-from data --name client2
fedora /bin/bash
```



```
[root@dockerhost ~]#  docker run -d -v /data --name data fedora echo “data volume container”
e5498ba6036c59b267000c96c2e5e9849d81993745af20540f7bcd7140cd462a
[root@dockerhost ~]# docker inspect -f "{{ .Volumes }}" data
map[/data:/var/lib/docker/vfs/dir/b12e737e412842bc707c56c4aaf3993e0654cc3683489177b416d8ccfa430020]
[root@dockerhost ~]# docker run -it --volumes-from data --name client1 fedora /bin/bash
bash-4.3# ls /data/
bash-4.3# touch /data/file1
bash-4.3# exit
[root@dockerhost ~]# ls /var/lib/docker/vfs/dir/b12e737e412842bc707c56c4aaf3993e0654cc3683489177b416d8ccfa430020
file1
[root@dockerhost ~]# docker run -it --volumes-from data --name client2 fedora /bin/bash
bash-4.3# ls /data/
file1
```

We can also use `--volumes-from` multiple times to get the data volumes from multiple containers. We can also create a chain by mounting volumes from the container that mounts from some other container.

How it works...

In case of data volume, when the host directory is not shared, Docker creates a directory within `/var/lib/docker/` and then shares it with other containers.

There's more...

- Volumes are deleted with `-v` flag to `docker rm`, only if no other container is using it. If some other container is using the volume, then the container will be removed (with `docker rm`) but the volume will not be removed.
- In the previous chapter, we saw how to configure the Docker registry, which by default starts with the `dev` flavor. In this registry, uploaded images were saved in the `/tmp/registry` folder within the container we started. We can mount a directory from the host at `/tmp/registry` within the registry container, so whenever we upload an image, it will be saved on the host that is running the Docker registry. So, to start the container, we run following command:

  ```
  $ docker run -v /srv:/tmp/registry -p 5000:5000 registry
  ```

  To push an image, we run the following command:

  ```
  $ docker push registry-host:5000/nkhare/f20
  ```

  After the image is successfully pushed, we can look at the content of the directory that we mounted within the Docker registry. In our case, we should see a directory structure as follows:

  ```
  /srv/
  ├── images
  │   ├──
  3f2fed40e4b0941403cd928b6b94e0fd236dfc54656c00e456747093d10157ac
  │   │   ├── ancestry
  │   │   ├── _checksum
  │   │   ├── json
  │   │   └── layer
  │   ├──
  511136ea3c5a64f264b78b5433614aec563103b4d4702f3ba7d4d2698e22c158
  │   │   ├── ancestry
  │   │   ├── _checksum
  │   │   ├── json
  │   │   └── layer
  │   ├──
  53263a18c28e1e54a8d7666cb835e9fa6a4b7b17385d46a7afe55bc5a7c1994c
  ```

```
|    |    ├── ancestry
|    |    ├── _checksum
|    |    ├── json
|    |    └── layer
|    └──
fd241224e9cf32f33a7332346a4f2ea39c4d5087b76392c1ac5490bf2ec55b68
|         ├── ancestry
|         ├── _checksum
|         ├── json
|         └── layer
├── repositories
|    └── nkhare
|         └── f20
|              ├── _index_images
|              ├── json
|              ├── tag_latest
|              └── taglatest_json
```

See also

- The documentation on the Docker website at `https://docs.docker.com/userguide/dockervolumes/`
- `http://container42.com/2013/12/16/persistent-volumes-with-docker-container-as-volume-pattern/`
- `http://container42.com/2014/11/03/docker-indepth-volumes/`

Linking two or more containers

With containerization, we would like to create our stack by running services on different containers and then linking them together. In the previous chapter, we created a WordPress container by putting both a web server and database in the same container. However, we can also put them in different containers and link them together. Container linking creates a parent-child relationship between them, in which the parent can see selected information of its children. Linking relies on the naming of containers.

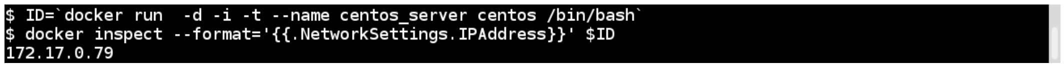
Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client.

How to do it...

1. Create a named container called `centos_server`:

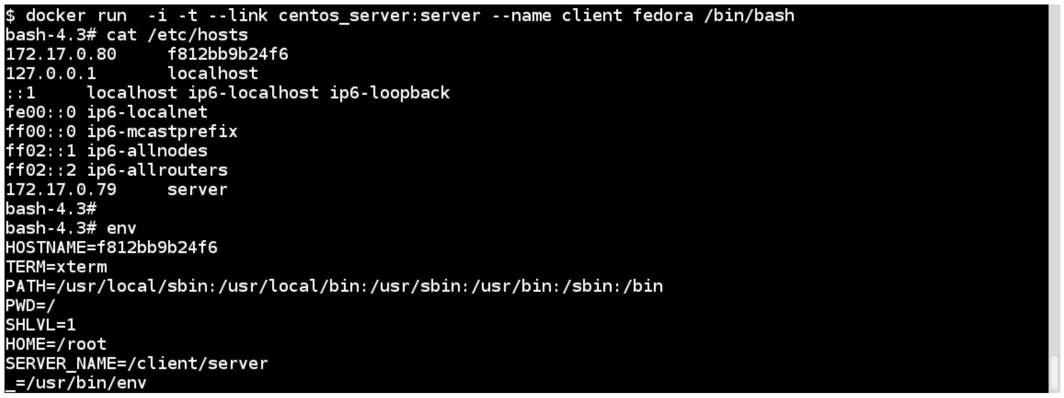
   ```
   $ docker run  -d -i -t --name centos_server centos /bin/bash
   ```



```
$ ID=`docker run  -d -i -t --name centos_server centos /bin/bash`
$ docker inspect --format='{{.NetworkSettings.IPAddress}}' $ID
172.17.0.79
```

2. Now, let's start another container with the `name` client and link it with the `centos_server` container using the `--link` option, which takes the `name:alias` argument. Then look at the `/etc/hosts` file:

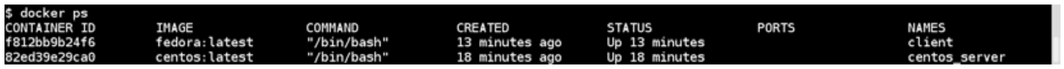
   ```
   $ docker run  -i -t --link centos_server:server --name client
   fedora /bin/bash
   ```



```
$ docker run  -i -t --link centos_server:server --name client fedora /bin/bash
bash-4.3# cat /etc/hosts
172.17.0.80     f812bb9b24f6
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.79     server
bash-4.3#
bash-4.3# env
HOSTNAME=f812bb9b24f6
TERM=xterm
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PWD=/
SHLVL=1
HOME=/root
SERVER_NAME=/client/server
_=/usr/bin/env
```

How it works...

In the preceding example, we linked the `centos_server` container to the client container with an alias server. By linking the two containers, an entry of the first container, which is `centos_server` in this case, is added to the `/etc/hosts` file in the client container. Also, an environment variable called `SERVER_NAME` is set within the client to refer to the server.



```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
f812bb9b24f6        fedora:latest       "/bin/bash"         13 minutes ago      Up 13 minutes                           client
82ed39e29ca0        centos:latest       "/bin/bash"         18 minutes ago      Up 18 minutes                           centos_server
```

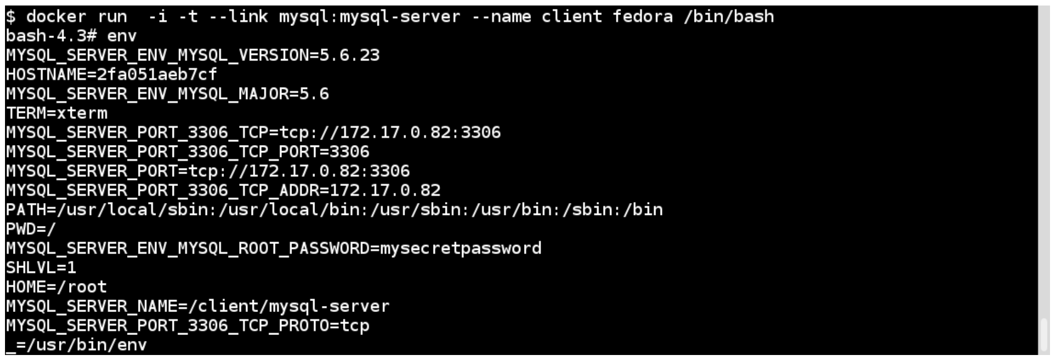
There's more...

Now, let's create a `mysql` container:

```
$ docker run --name mysql -e MYSQL_ROOT_PASSWORD=mysecretpassword -d
mysql
```

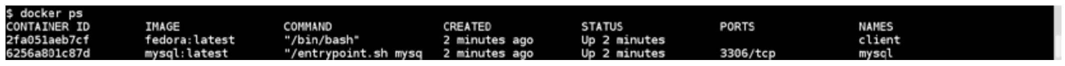
Then, let's link it from a client and check the environment variables:

```
$ docker run  -i -t --link mysql:mysql-server --name client fedora
/bin/bash
```



```
$ docker run  -i -t --link mysql:mysql-server --name client fedora /bin/bash
bash-4.3# env
MYSQL_SERVER_ENV_MYSQL_VERSION=5.6.23
HOSTNAME=2fa051aeb7cf
MYSQL_SERVER_ENV_MYSQL_MAJOR=5.6
TERM=xterm
MYSQL_SERVER_PORT_3306_TCP=tcp://172.17.0.82:3306
MYSQL_SERVER_PORT_3306_TCP_PORT=3306
MYSQL_SERVER_PORT=tcp://172.17.0.82:3306
MYSQL_SERVER_PORT_3306_TCP_ADDR=172.17.0.82
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PWD=/
MYSQL_SERVER_ENV_MYSQL_ROOT_PASSWORD=mysecretpassword
SHLVL=1
HOME=/root
MYSQL_SERVER_NAME=/client/mysql-server
MYSQL_SERVER_PORT_3306_TCP_PROTO=tcp
_=/usr/bin/env
```

Also, let's look at the `docker ps` output:



```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                CREATED             STATUS              PORTS               NAMES
2fa051aeb7cf        fedora:latest       "/bin/bash"            2 minutes ago       Up 2 minutes                            client
6256a801c87d        mysql:latest        "/entrypoint.sh mysq   2 minutes ago       Up 2 minutes        3306/tcp            mysql
```

If you look closely, we did not specify the `-P` or `-p` options to map ports between two containers while starting the `client` container. Depending on the ports exposed by a container, Docker creates an internal secure tunnel in the containers that links to it. And, to do that, Docker sets environment variables within the linker container. In the preceding case, `mysql` is the linked container and client is the linker container. As the `mysql` container exposes port `3306`, we see corresponding environment variables (`MYSQL_SERVER_*`) within the client container.

[As linking depends on the name of the container, if you want to reuse a name, you must delete the old container.]

See also

- Documentation on the Docker website at `https://docs.docker.com/userguide/dockerlinks/`

Developing a LAMP application by linking containers

Let's extend the previous recipe by creating a LAMP application (WordPress) by linking the containers.

Getting ready

To pull MySQL and WordPress images from the Docker registry:

- For MySQL:
    - For image, visit `https://registry.hub.docker.com/_/mysql/`
    - For Dockerfile, visit `https://github.com/docker-library/docker-mysql`
- For WordPress:
    - For image, visit `https://registry.hub.docker.com/_/wordpress/`
    - For Dockerfile, visit `https://github.com/docker-library/wordpress`

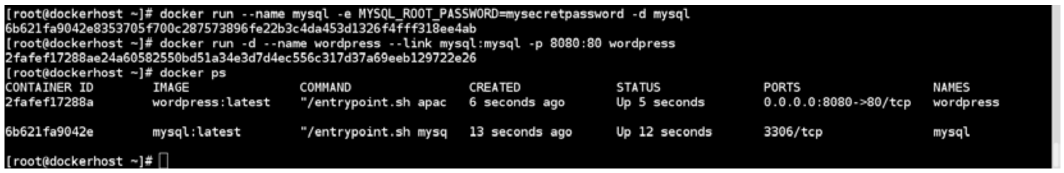
How to do it…

1. First, start a `mysql` container:

```
$ docker run --name mysql -e
MYSQL_ROOT_PASSWORD=mysecretpassword -d mysql
```

2. Then, start the `wordpress` container and link it with the `mysql` container:

```
$ docker run -d --name wordpress --link mysql:mysql -p 8080:80
wordpress
```



```
[root@dockerhost ~]# docker run --name mysql -e MYSQL_ROOT_PASSWORD=mysecretpassword -d mysql
6b621fa9042e8353705f700c287573896fe22b3c4da453d1326f4fff318ee4ab
[root@dockerhost ~]# docker run -d --name wordpress --link mysql:mysql -p 8080:80 wordpress
2fafef17288ae24a60582550bd51a34e3d7d4ec556c317d37a69eeb129722e26
[root@dockerhost ~]# docker ps
CONTAINER ID        IMAGE               COMMAND              CREATED             STATUS              PORTS                  NAMES
2fafef17288a        wordpress:latest    "/entrypoint.sh apac 6 seconds ago       Up 5 seconds        0.0.0.0:8080->80/tcp   wordpress

6b621fa9042e        mysql:latest        "/entrypoint.sh mysq 13 seconds ago      Up 12 seconds       3306/tcp               mysql

[root@dockerhost ~]# 
```

We have the Docker host's `8080` port to container `80` port, so we can connect WordPress by accessing the `8080` port on the Docker host with the `http://<DockerHost>:8080` URL.

How it works…

A link is created between the `wordpress` and `mysql` containers. Whenever the `wordpress` container gets a DB request, it passes it on to the `mysql` container and gets the results. Look at the preceding recipe for more details.

Networking of multihost containers with Flannel

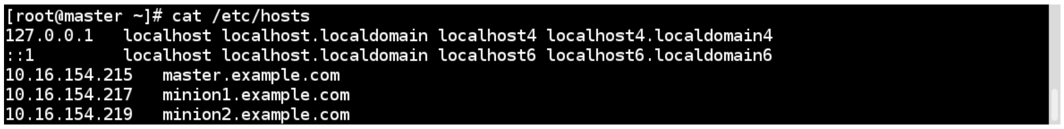
In this recipe, we'll use Flannel (`https://github.com/coreos/flannel`) to set up multihost container networking. Flannel is a generic overlay network that can be used as an alternative to **Software Defined Networking** (**SDN**). It is an IP-based solution that uses **Virtual Extensible LAN** (**VXLAN**), in which unique IP addresses are assigned to each container on a unique subnet given to the host that is running that container. So, in this kind of a solution, a different subnet and communication occurs within each host in the cluster, using the overlay network. Flannel uses the `etcd` service (https://github.com/coreos/etcd) for the key-value store.

Getting ready

For this recipe, we will require three VMs or physical machines with Fedora 21 installed.

How to do it...

1. Let's call one machine/VM `master` and other two `minion1` and `minion2`. According to your system's IP addresses, update the `/etc/hosts` file as follows:



```
[root@master ~]# cat /etc/hosts
127.0.0.1   localhost localhost.localdomain localhost4 localhost4.localdomain4
::1         localhost localhost.localdomain localhost6 localhost6.localdomain6
10.16.154.215   master.example.com
10.16.154.217   minion1.example.com
10.16.154.219   minion2.example.com
```

2. Install `etcd`, `Flannel`, and `Docker` on all the systems we set up:

   ```
   $ yum install -y etcd flannel docker
   ```

3. Modify the value of the `ETCD_LISTEN_CLIENT_URLS` to `http://master.example.com:4001` in the `/etc/etcd/etcd.conf` file as follows:

   ```
   ETCD_LISTEN_CLIENT_URLS="http://master.example.com:4001"
   ```

4. In the master, start the `etcd` service and check its status:

   ```
   $ systemctl start etcd
   $ systemctl enable etcd
   $ systemctl status etcd
   ```

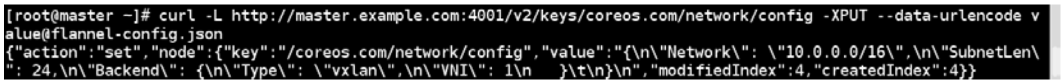
5. In the master, create a file called `flannel-config.json` with the following content:

   ```
   {
   "Network": "10.0.0.0/16",
   "SubnetLen": 24,
   ```

```
"Backend": {
"Type": "vxlan",
"VNI": 1
   }
}
```

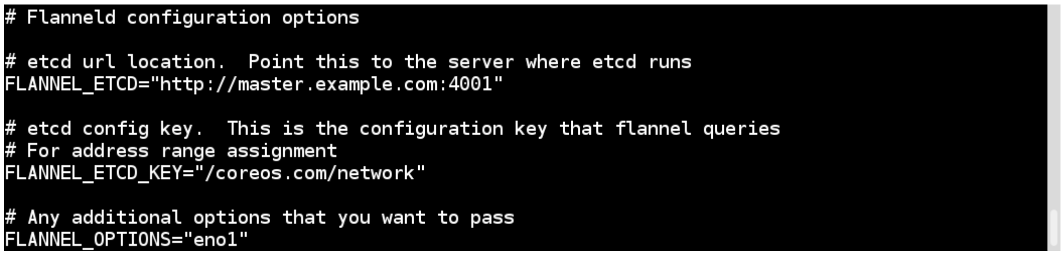
6. Upload the preceding configuration file to `etcd` using `config` as the key:

   ```
   $ curl -L
   http://master.example.com:4001/v2/keys/coreos.com/network/conf
   ig -XPUT --data-urlencode value@flannel-config.json
   ```



```
[root@master ~]# curl -L http://master.example.com:4001/v2/keys/coreos.com/network/config -XPUT --data-urlencode v
alue@flannel-config.json
{"action":"set","node":{"key":"/coreos.com/network/config","value":"{\n\"Network\": \"10.0.0.0/16\",\n\"SubnetLen\
": 24,\n\"Backend\": {\n\"Type\": \"vxlan\",\n\"VNI\": 1\n   }\t\n}\n","modifiedIndex":4,"createdIndex":4}}
```

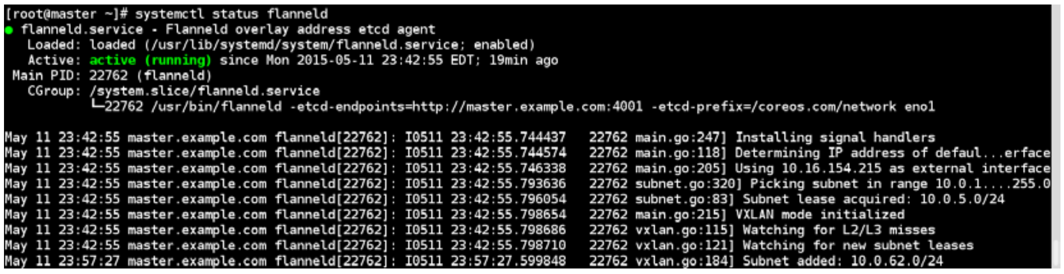
7. In master, update `FLANNEL_OPTIONS` in the `/etc/sysconfig/flanneld` file to reflect the interface of the system. Also, update `FLANNEL_ETCD` to use hostname instead of the 127.0.0.1:4001 address.



```
# Flanneld configuration options

# etcd url location.  Point this to the server where etcd runs
FLANNEL_ETCD="http://master.example.com:4001"

# etcd config key.  This is the configuration key that flannel queries
# For address range assignment
FLANNEL_ETCD_KEY="/coreos.com/network"

# Any additional options that you want to pass
FLANNEL_OPTIONS="eno1"
```

8. To enable and start the `flanneld` service in master:

   ```
   $ systemctl enable flanneld
   $ systemctl start flanneld
   $ systemctl status flanneld
   ```

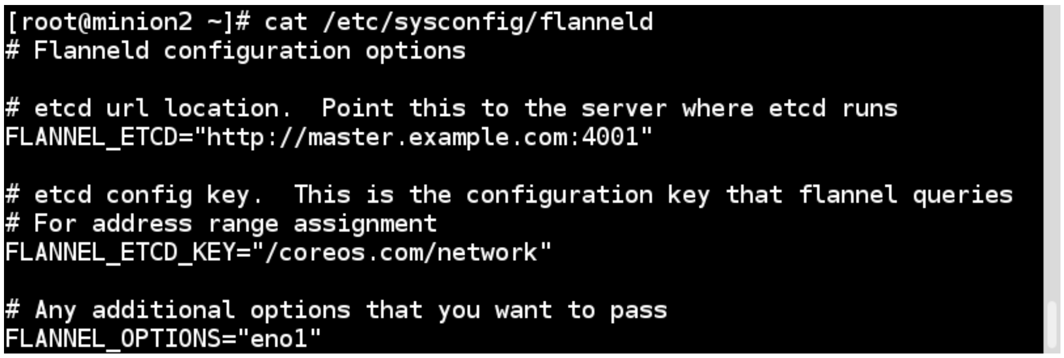


```
[root@master ~]# systemctl status flanneld
● flanneld.service - Flanneld overlay address etcd agent
   Loaded: loaded (/usr/lib/systemd/system/flanneld.service; enabled)
   Active: active (running) since Mon 2015-05-11 23:42:55 EDT; 19min ago
 Main PID: 22762 (flanneld)
   CGroup: /system.slice/flanneld.service
           └─22762 /usr/bin/flanneld -etcd-endpoints=http://master.example.com:4001 -etcd-prefix=/coreos.com/network eno1

May 11 23:42:55 master.example.com flanneld[22762]: I0511 23:42:55.744437   22762 main.go:247] Installing signal handlers
May 11 23:42:55 master.example.com flanneld[22762]: I0511 23:42:55.744574   22762 main.go:118] Determining IP address of defaul...erface
May 11 23:42:55 master.example.com flanneld[22762]: I0511 23:42:55.746338   22762 main.go:205] Using 10.16.154.215 as external interface
May 11 23:42:55 master.example.com flanneld[22762]: I0511 23:42:55.793636   22762 subnet.go:320] Picking subnet in range 10.0.1....255.0
May 11 23:42:55 master.example.com flanneld[22762]: I0511 23:42:55.796054   22762 subnet.go:83] Subnet lease acquired: 10.0.5.0/24
May 11 23:42:55 master.example.com flanneld[22762]: I0511 23:42:55.798654   22762 main.go:215] VXLAN mode initialized
May 11 23:42:55 master.example.com flanneld[22762]: I0511 23:42:55.798686   22762 vxlan.go:115] Watching for L2/L3 misses
May 11 23:42:55 master.example.com flanneld[22762]: I0511 23:42:55.798710   22762 vxlan.go:121] Watching for new subnet leases
May 11 23:57:27 master.example.com flanneld[22762]: I0511 23:57:27.599848   22762 vxlan.go:184] Subnet added: 10.0.62.0/24
```

9. From the minion systems, check the connectivity to master for `etcd`:

   ```
   [root@minion1 ~]#  curl -L
   http://master.example.com:4001/v2/keys/coreos.com/network/conf
   ig
   ```

10. Update the `/etc/sysconfig/flanneld` file in both minions to point to the `etcd` server running in master and update `FLANNEL_OPTIONS` to reflect the interface of the minion host:



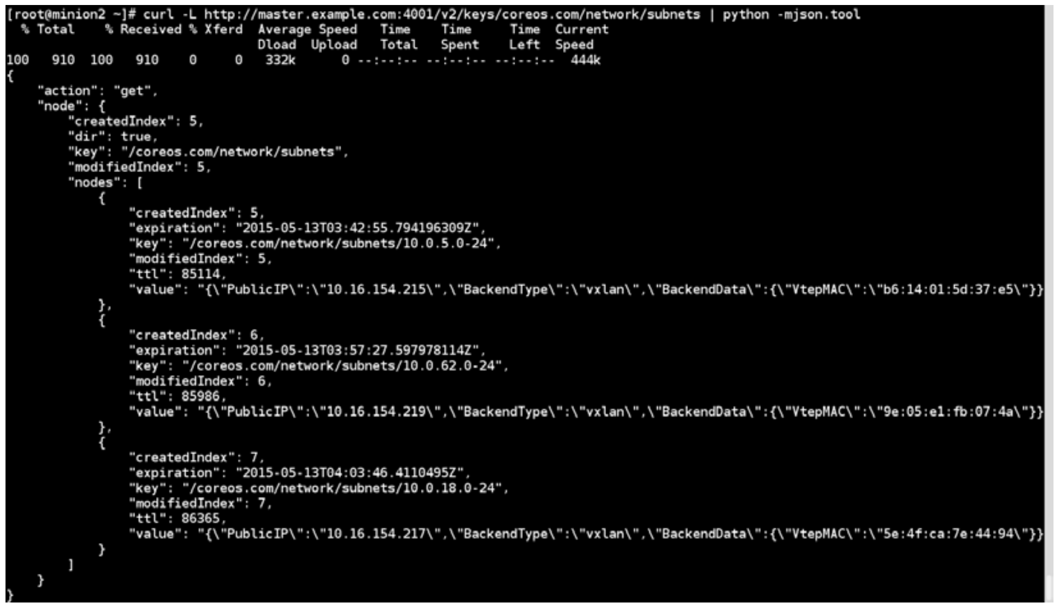
```
[root@minion2 ~]# cat /etc/sysconfig/flanneld
# Flanneld configuration options

# etcd url location.  Point this to the server where etcd runs
FLANNEL_ETCD="http://master.example.com:4001"

# etcd config key.  This is the configuration key that flannel queries
# For address range assignment
FLANNEL_ETCD_KEY="/coreos.com/network"

# Any additional options that you want to pass
FLANNEL_OPTIONS="eno1"
```

11. To enable and start the `flanneld` service in both the minions:

    ```
    $ systemctl enable flanneld
    $ systemctl start flanneld
    $ systemctl status flanneld
    ```

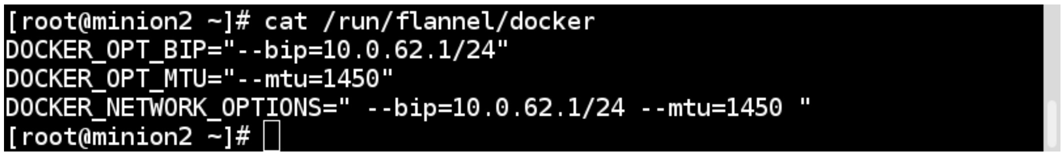
12. In any of the hosts in the cluster, run the following command:

    ```
    $ curl -L
    http://master.example.com:4001/v2/keys/coreos.com/network/subn
    ets | python -mjson.tool
    ```



```
[root@minion2 ~]# curl -L http://master.example.com:4001/v2/keys/coreos.com/network/subnets | python -mjson.tool
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   910  100   910    0     0   332k      0 --:--:-- --:--:-- --:--:--  444k
{
    "action": "get",
    "node": {
        "createdIndex": 5,
        "dir": true,
        "key": "/coreos.com/network/subnets",
        "modifiedIndex": 5,
        "nodes": [
            {
                "createdIndex": 5,
                "expiration": "2015-05-13T03:42:55.794196309Z",
                "key": "/coreos.com/network/subnets/10.0.5.0-24",
                "modifiedIndex": 5,
                "ttl": 85114,
                "value": "{\"PublicIP\":\"10.16.154.215\",\"BackendType\":\"vxlan\",\"BackendData\":{\"VtepMAC\":\"b6:14:01:5d:37:e5\"}}
            },
            {
                "createdIndex": 6,
                "expiration": "2015-05-13T03:57:27.597978114Z",
                "key": "/coreos.com/network/subnets/10.0.62.0-24",
                "modifiedIndex": 6,
                "ttl": 85986,
                "value": "{\"PublicIP\":\"10.16.154.219\",\"BackendType\":\"vxlan\",\"BackendData\":{\"VtepMAC\":\"9e:05:e1:fb:07:4a\"}}
            },
            {
                "createdIndex": 7,
                "expiration": "2015-05-13T04:03:46.4110495Z",
                "key": "/coreos.com/network/subnets/10.0.18.0-24",
                "modifiedIndex": 7,
                "ttl": 86365,
                "value": "{\"PublicIP\":\"10.16.154.217\",\"BackendType\":\"vxlan\",\"BackendData\":{\"VtepMAC\":\"5e:4f:ca:7e:44:94\"}}
            }
        ]
    }
}
```

This tells us the number of hosts in the network and the subnets associated (look at the key for each node) with them. We can associate the subnet with the MAC address on the hosts. On each host, the `/run/flannel/docker` and `/run/flannel/subnet.env` files are populated with subnet information. For instance, in `minion2`, you would see something like the following:

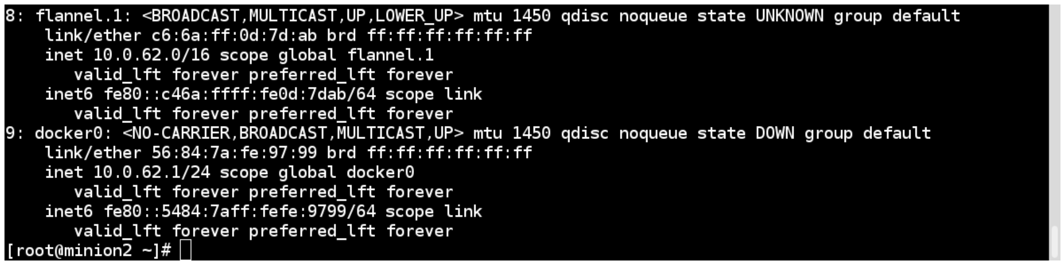


```
[root@minion2 ~]# cat /run/flannel/docker
DOCKER_OPT_BIP="--bip=10.0.62.1/24"
DOCKER_OPT_MTU="--mtu=1450"
DOCKER_NETWORK_OPTIONS=" --bip=10.0.62.1/24 --mtu=1450 "
[root@minion2 ~]#
```

13. To restart the Docker daemon in all the hosts:

    `$ systemctl restart docker`

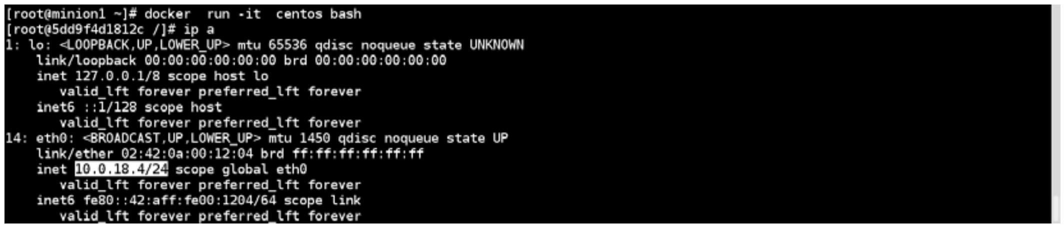
Then, look at the IP address of the `docker0` and `flannel.1` interfaces. In `minion2`, it looks like the following:



```
8: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UNKNOWN group default
    link/ether c6:6a:ff:0d:7d:ab brd ff:ff:ff:ff:ff:ff
    inet 10.0.62.0/16 scope global flannel.1
       valid_lft forever preferred_lft forever
    inet6 fe80::c46a:ffff:fe0d:7dab/64 scope link
       valid_lft forever preferred_lft forever
9: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1450 qdisc noqueue state DOWN group default
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
    inet 10.0.62.1/24 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::5484:7aff:fefe:9799/64 scope link
       valid_lft forever preferred_lft forever
[root@minion2 ~]# 
```

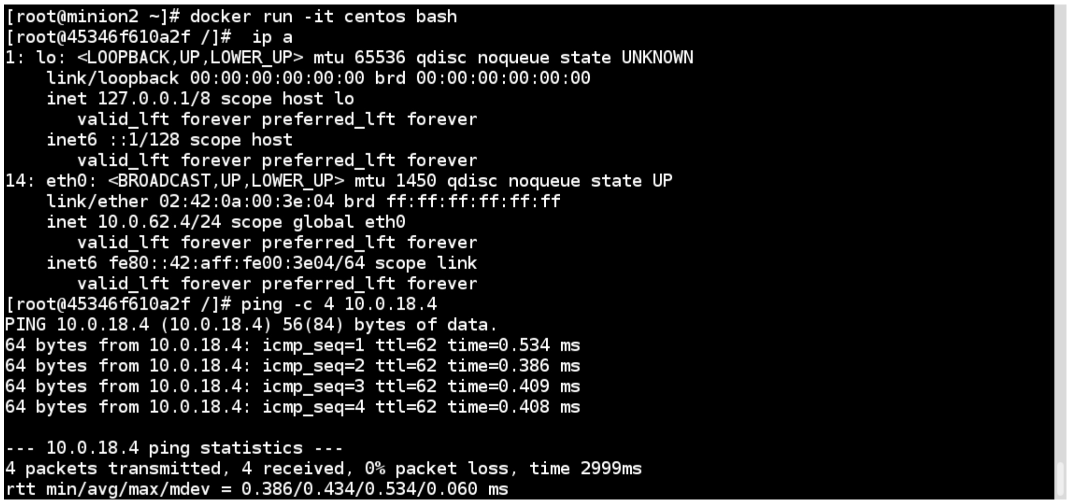
We can see that the `docker0` interface got the IP from the same subnet as the `flannel.1` interface, which is used to route all traffic.

14. We are all set to spawn two containers in any of the hosts and they should be able to communicate. Let's create one container in `minion1` and get its IP address:



```
[root@minion1 ~]# docker  run -it  centos bash
[root@5dd9f4d1812c /]# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
14: eth0: <BROADCAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP
    link/ether 02:42:0a:00:12:04 brd ff:ff:ff:ff:ff:ff
    inet 10.0.18.4/24 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:aff:fe00:1204/64 scope link
       valid_lft forever preferred_lft forever
```

15. Now create another container in `minion2` and ping the container running in `minion1` as follows:



```
[root@minion2 ~]# docker run -it centos bash
[root@45346f610a2f /]#  ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
14: eth0: <BROADCAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP
    link/ether 02:42:0a:00:3e:04 brd ff:ff:ff:ff:ff:ff
    inet 10.0.62.4/24 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:aff:fe00:3e04/64 scope link
       valid_lft forever preferred_lft forever
[root@45346f610a2f /]# ping -c 4 10.0.18.4
PING 10.0.18.4 (10.0.18.4) 56(84) bytes of data.
64 bytes from 10.0.18.4: icmp_seq=1 ttl=62 time=0.534 ms
64 bytes from 10.0.18.4: icmp_seq=2 ttl=62 time=0.386 ms
64 bytes from 10.0.18.4: icmp_seq=3 ttl=62 time=0.409 ms
64 bytes from 10.0.18.4: icmp_seq=4 ttl=62 time=0.408 ms

--- 10.0.18.4 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2999ms
rtt min/avg/max/mdev = 0.386/0.434/0.534/0.060 ms
```

How it works...

With Flannel, we first configure the overlay with the `10.0.0.0/16` network. Then, each host picks up a random `/24` network; for instance, in our case, `minion2` gets the `10.0.62.0/24` subnet and so on. Once configured, a container in the host gets the IP address from that chosen subnet. Flannel encapsulates the packets and sends it to remote hosts using UDP.

Also, during installation, Flannel copies a configuration file (`flannel.conf`) within `/usr/lib/systemd/system/docker.service.d/`, which Docker uses to configure itself.

See also

- The diagram from Flannel GitHub to help you understand the theory of operations at `https://github.com/coreos/flannel/blob/master/packet-01.png`
- The documentation on the CoreOS website at `https://coreos.com/blog/introducing-rudder/`
- Scott Collier's blog post about setting Flannel on Fedora at `http://www.colliernotes.com/2015/01/flannel-and-docker-on-fedora-getting.html`

Assigning IPv6 addresses to containers

By default, Docker assigns IPv4 addresses to containers. With Docker 1.5, a feature has been added to support IPv6 addresses.

Getting ready

Make sure the Docker daemon (version 1.5 and above) is running on the host and you can connect through the Docker client.

How to do it...

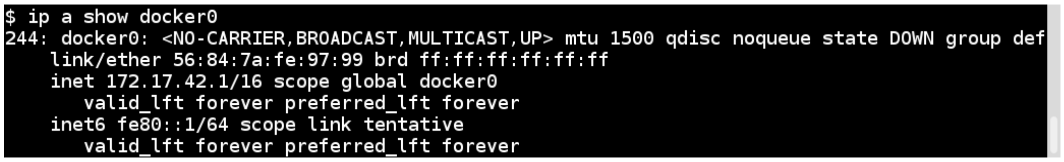
1. To start the Docker daemon with the `--ipv6` option, we can add this option in the daemon's configuration file (`/etc/sysconfig/docker` on Fedora) as follows:

   ```
   OPTIONS='--selinux-enabled --ipv6'
   ```

   Alternatively, if we start Docker in daemon mode, then we can start it as follows:

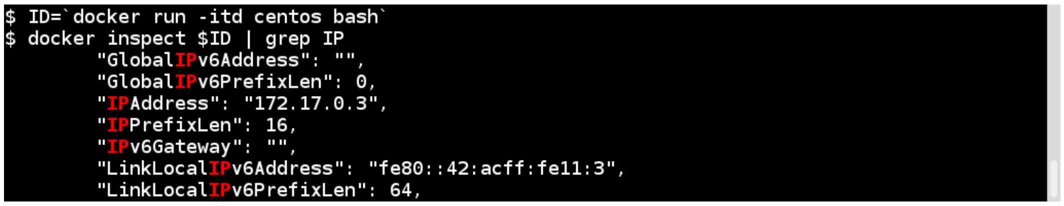
   ```
   $ docker -d --ipv6
   ```

By running either of these commands, Docker will set up the `docker0` bridge with the IPv6 local link address `fe80::1`.



```
$ ip a show docker0
244: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group def
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
    inet 172.17.42.1/16 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::1/64 scope link tentative
       valid_lft forever preferred_lft forever
```

2. Let's start the container and look for the IP addresses assigned to it:



```
$ ID=`docker run -itd centos bash`
$ docker inspect $ID | grep IP
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "IPAddress": "172.17.0.3",
        "IPPrefixLen": 16,
        "IPv6Gateway": "",
        "LinkLocalIPv6Address": "fe80::42:acff:fe11:3",
        "LinkLocalIPv6PrefixLen": 64,
```

As we can see, both the IPv4 and local link IPv6 addresses are available to the container. To ping on the IPv6 address of a container from the host machine, run the following command:

```
$ ping6 -I docker0 fe80::42:acff:fe11:3
```

To ping the `docker0` bridge from the container, run the following command:

```
[root@c7562c38bd0f /]# ping6 -I eth0 fe80::1
```

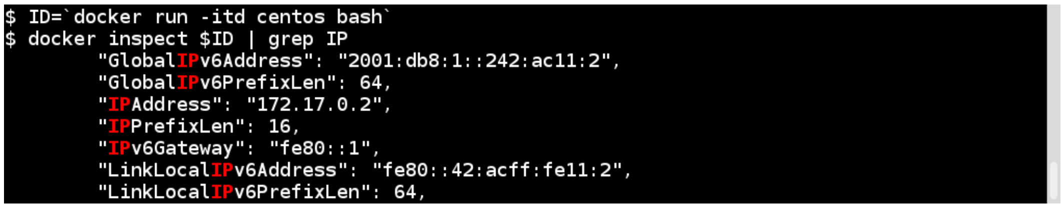
How it works...

Docker configures the `docker0` bridge to assign IPv6 addresses to containers, which enables us to use the IPv6 address of containers.

There's more...

By default, containers will get the link-local address. To assign them a globally routable address, you can pass the IPv6 subnet pick address with `--fixed-cidr-v6` as follows:

```
$ docker -d --ipv6 --fixed-cidr-v6="2001:db8:1::/64"
```



```
$ ID=`docker run -itd centos bash`
$ docker inspect $ID | grep IP
        "GlobalIPv6Address": "2001:db8:1::242:ac11:2",
        "GlobalIPv6PrefixLen": 64,
        "IPAddress": "172.17.0.2",
        "IPPrefixLen": 16,
        "IPv6Gateway": "fe80::1",
        "LinkLocalIPv6Address": "fe80::42:acff:fe11:2",
        "LinkLocalIPv6PrefixLen": 64,
```

From here, we can see that the globally routable address (GlobalIPv6Address) is now being set.

See also

- The Docker 1.5 release notes at `https://blog.docker.com/2015/02/docker-1-5-ipv6-support-read-only-containers-stats-named-dockerfiles-and-more/`.
- The documentation on the Docker website at `http://docs.docker.com/v1.5/articles/networking/#ipv6`.
- You might need to delete the exiting `docker0` bridge on the host before setting the IPv6 option. To understand how to do so, visit `http://docs.docker.com/v1.5/articles/networking/#customizing-docker0`.

5
Docker Use Cases

In this chapter, we will cover the following recipes:

- Testing with Docker
- Doing CI/CD with Shippable and Red Hat OpenShift
- Doing CI/CD with Drone
- Setting up PaaS with OpenShift Origin
- Building and deploying an app on OpenShift v3 from the source code
- Configuring Docker as a hypervisor driver for Openstack

Introduction

Now we know how to work with containers and images. In the last chapter, we also saw how to link containers and share data between the host and other containers. We also saw how containers from one host can communicate with other containers from other hosts.

Now let's look at different use cases of Docker. Let's list a few of them here:

- **Quick prototyping of ideas**: This is one of my favorite use cases. Once we have an idea, it is very easy to prototype it with Docker. All we have to do is set up containers to provide all the backend services we need and connect them together. For example, to set up a LAMP application, get the web and DB servers and link them, as we saw in the previous chapter.
- **Collaboration and distribution**: GitHub is one of the best examples of collaborating and distributing the code. Similarly, Docker provides features such as Dockerfile, registry, and import/export to share and collaborate with others. We have covered all this in earlier chapters.

- **Continuous Integration** (**CI**): The following definition on Martin Fowler's website (`http://www.martinfowler.com/articles/continuousIntegration.html`) covers it all:

  "Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly. This article is a quick overview of Continuous Integration summarizing the technique and its current usage."

Using recipes from other chapters, we can build an environment for CI using Docker. You can create your own CI environment or get services from companies such as Shippable and Drone. We'll see how Shippable and Drone can be used for CI work later in this chapter. Shippable is not a hosted solution but Drone is, which can give you better control. I thought it would be helpful if I talk about both of them here:

- **Continuous Delivery** (**CD**): The next step after CI is Continuous Delivery, through which we can deploy our code rapidly and reliably to our customers, the cloud and other environments without any manual work. In this chapter, we'll see how we can automatically deploy an app on Red Hat OpenShift through Shippable CI.
- **Platform-as-a-Service** (**PaaS**): Docker can be used to build your own PaaS. It can be deployed using tools/platforms such as OpenShift, CoreOS, Atomic, Tsuru, and so on. Later in this chapter, we'll see how to set up PaaS using OpenShift Origin (`https://www.openshift.com/products/origin`).

Testing with Docker

While doing the development or QA, it will be helpful if we can check our code against different environments. For example, we may wish to check our Python code between different versions of Python or on different distributions such as Fedora, Ubuntu, CentOS, and so on. For this recipe, we will pick up sample code from Flask's GitHub repository, which is a microframework for Python (`http://flask.pocoo.org/`). I chose this to keep things simple, and it is easier to use for other recipes as well.

For this recipe, we will create images to have one container with Python 2.7 and other with Python 3.3. We'll then use a sample Python test code to run against each container.

Getting ready

- As we are going to use example code from Flask's GitHub repository, let's clone it:

  ```
  $ git clone https://github.com/mitsuhiko/flask
  ```

- Create a `Dockerfile_2.7` file as follows and then build an image from it:

  ```
  $ cat /tmp/ Dockerfile_2.7
  FROM python:2.7
  RUN pip install flask
  RUN pip install pytest
  WORKDIR /test
  CMD ["/usr/local/bin/py.test"]
  ```

- To build the `python2.7test` image, run the following command:

  ```
  $ docker build -t python2.7test - < /tmp/Dockerfile_2.7
  ```

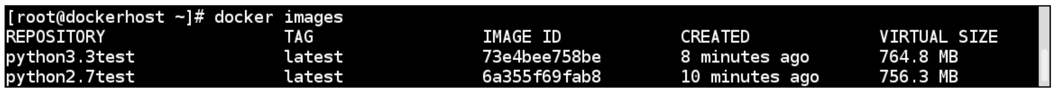
- Similarly, create a Dockerfile with `python:3.3` as the base image and build the `python3.3test` image:

  ```
  $ cat /tmp/Dockerfile_3.3
  FROM python:3.3
  RUN pip install flask
  RUN pip install pytest
  WORKDIR /test
  CMD ["/usr/local/bin/py.test"]
  ```

- To build the image, run the following command:

  ```
  $ docker build -t python3.3test  - < /tmp/Dockerfile_3.3
  ```

Make sure both the images are created.



```
[root@dockerhost ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
python3.3test       latest              73e4bee758be        8 minutes ago       764.8 MB
python2.7test       latest              6a355f69fab8        10 minutes ago      756.3 MB
```

How to do it...

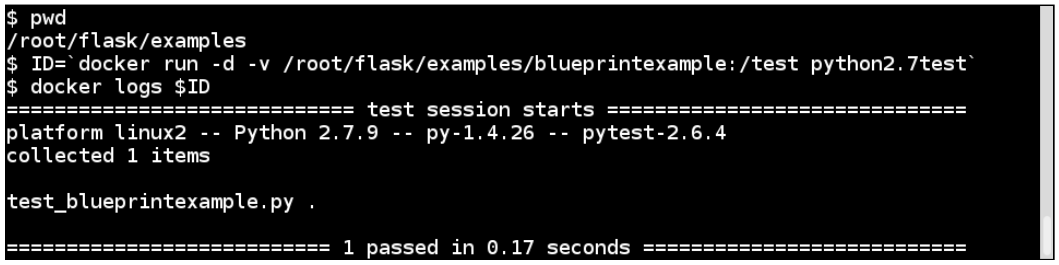
Now, using Docker's volume feature, we will mount the external directory that contains the source code and test cases. To test with Python 2.7, do the following:

1. Go to the directory that contains the Flask examples:

   ```
   $ cd /tmp/flask/examples/
   ```

2. Start a container with the `python2.7` test image and mount `blueprintexample` under `/test`:

   ```
   $ docker run -d -v `pwd`/blueprintexample:/test python2.7test
   ```



```
$ pwd
/root/flask/examples
$ ID=`docker run -d -v /root/flask/examples/blueprintexample:/test python2.7test`
$ docker logs $ID
============================ test session starts =============================
platform linux2 -- Python 2.7.9 -- py-1.4.26 -- pytest-2.6.4
collected 1 items

test_blueprintexample.py .

========================== 1 passed in 0.17 seconds ==========================
```

3. Similarly, to test with Python 3.3, run the following command:

   ```
   $ docker run -d -v `pwd`/blueprintexample:/test python3.3test
   ```

4. While running the preceding test on Fedora/RHEL/CentOS where SELinux is enabled, you will get a `Permission denied` error. To fix it, relabel the host directory while mounting it within the container as follows:

   ```
   $ docker run -d -v `pwd`/blueprintexample:/test:z
   python2.7test
   ```

> For more details on SELinux, please look at *Chapter 9*, *Docker Security*.

How it works...

As you can see from the Dockerfile, before running CMD, which runs the `py.test` binary, we change our working directory to `/test`. And while starting the container, we mount our source code to `/test`. So, as soon as the container starts, it will run the `py.test` binary and run tests.

There's more...

- In this recipe, we have seen how to test our code with different versions of Python. Similarly, you can pick up different base images from Fedora, CentOS, Ubuntu and test them on different Linux distributions.
- If you use Jenkins in your environment, then you can use its Docker plugin to dynamically provision a slave, run a build, and tear it down on the Docker host. More details about this can be found at `https://wiki.jenkins-ci.org/display/JENKINS/Docker+Plugin`.

Doing CI/CD with Shippable and Red Hat OpenShift

In the preceding recipe, we saw an example of how Docker can be used for testing in a local Dev and QA environment. Let's look at an end-to-end example to see how Docker is now used in the CI/CD environment. In this recipe, we'll see how we can use Shippable (`http://www.shippable.com/`) to perform CI/CD and deploy it on Red Hat's OpenShift environment (`https://openshift.redhat.com`).

Shippable is a SaaS platform that lets you easily add Continuous Integration/Deployment to your GitHub and Bitbucket(Git) repositories, which is completely built on Docker. Shippable uses build minions, which are Docker-based containers, to run workloads. Shippable supports many languages such as Ruby, Python, Node.js, Java, Scala, PHP, Go, and Clojure. The default build minions are of Ubuntu 12.04 LTS and Ubuntu 14.04. They have also added support to use custom images from Docker Hub as minions. Shippable CI needs information about the project and build instructions in a `yml` file called `shippable.yml`, which you have to provide in your source code repo. The `yml` file contains the following instructions:

- `build_image`: This is a Docker image to use to build
- `language`: This will show the programming language
    - `versions`: You can specify different versions of the language to get tested in a single build instruction.
- `before_install`: These are the instructions before running the build
- `script`: This is a binary/script to run the test
- `after_success`: These are instructions after the build succeeds; this is used to perform deployment on PaaS such as Heroku, Amazon Elastic Beanstalk, AWS OpsWorks, Google App Engine, Red Hat OpenShift, and others.

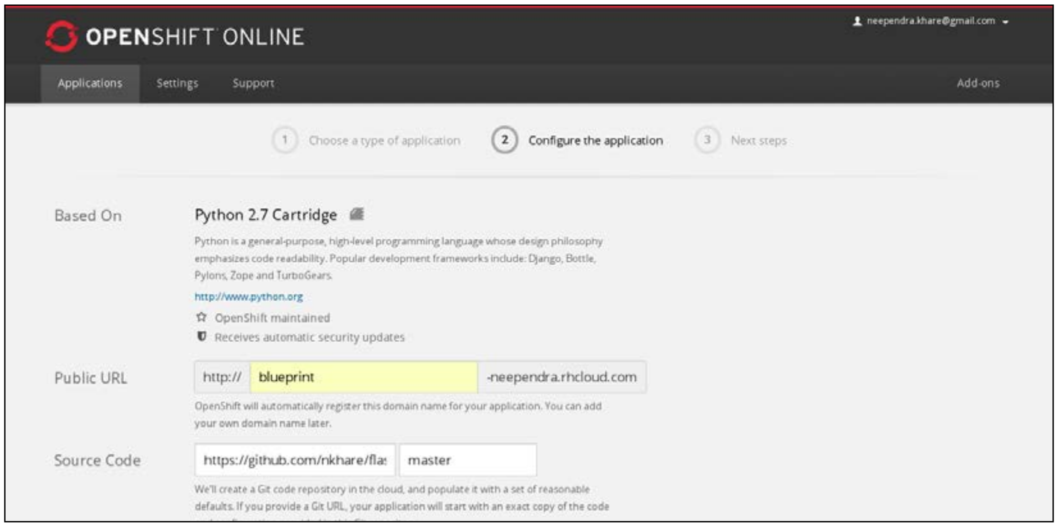
Red Hat's OpenShift is a PaaS platform to host your application. Currently, it uses non-Docker based container technology to host the application, but the next version of OpenShift (`https://github.com/openshift/origin`) is being built on Kubernetes and Docker. This tells us the pace at which Docker is being adopted in the enterprise world. We'll see how to set up OpenShift v3 later in this chapter.

For this recipe, we will use the same example code we used in the previous recipe, to first test on Shippable and then deploy it on OpenShift.

Getting ready

1. Create an account on Shippable (`https://www.shippable.com/`).
2. Fork the flask example from `https://github.com/openshift/flask-example`.

3. Create an app on OpenShift for the forked repository with the following steps:
    1. Create an account (`https://www.openshift.com/app/account/new`) on OpenShift and log in.
    2. Select **Python 2.7 Cartridge** for the application.
    3. Update the **Public URL** section you want. In the **Source Code** section, provide the URL of our forked repo. For this example, I have put down `blueprint` and `https://github.com/nkhare/flask-example` respectively:


OPENSHIFT ONLINE
neependra.khare@gmail.com
Applications
Settings
Support
Add-ons
1 Choose a type of application
2 Configure the application
3 Next steps

Based On: Python 2.7 Cartridge

Python is a general-purpose, high-level programming language whose design philosophy emphasizes code readability. Popular development frameworks include: Django, Bottle, Pylons, Zope and TurboGears.

http://www.python.org

OpenShift maintained

Receives automatic security updates

Public URL: http:// blueprint -neependra.rhcloud.com

OpenShift will automatically register this domain name for your application. You can add your own domain name later.

Source Code: https://github.com/nkhare/fla: master

We'll create a Git code repository in the cloud, and populate it with a set of reasonable defaults. If you provide a Git URL, your application will start with an exact copy of the code

    4. Click on **Create Application** to create the new app. Once created, you should be able to access the Public URL we mentioned in the previous step.

Once the app is created, OpenShift provides a way to manage/update the source code for this app in the `Making code changes` section. As we want to deploy the app using Shippable, we don't have to follow those instructions.

4. Clone the forked repository on the local system:

```
$ git clone git@github.com:nkhare/flask-example.git
```

5. Let's use the same blueprint example that we used earlier. To do so, follow these instructions:
    1. Clone the flask repository:

    ```
    $ git clone https://github.com/mitsuhiko/flask.git
    ```

    2. Copy the blueprint example:

    ```
    $ cp -Rv flask/examples/blueprintexample/* flask-example/
    wsgi/
    ```

6. Update the `flask-example/wsgi/application` file to import the `app` module from the `blueprintexample` module. So, the last line in the `flask-example/wsgi/application` file looks like the following:

   ```
   from blueprintexample import app as application
   ```

7. Add the `requirements.txt` file with the following contents at the top level of the flask-example repository:

   ```
   flask
   pytest
   ```

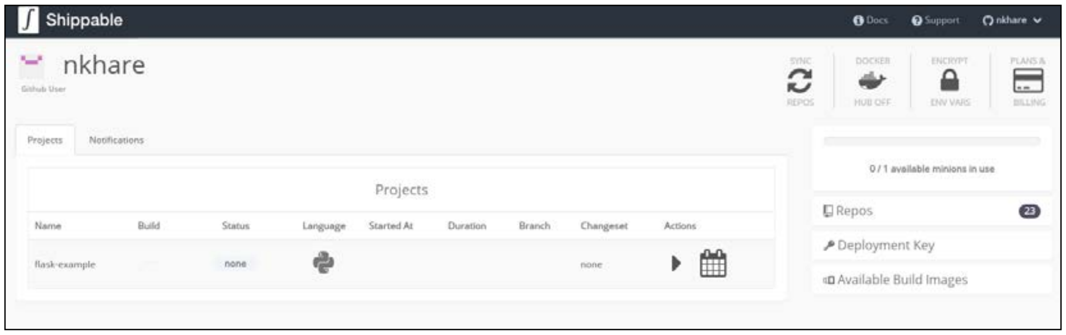
8. Add the `shippable.yml` file with following content:

   ```
   language: python

   python:
     - 2.6
     - 2.7

   install:
     - pip install -r requirements.txt

   # Make folders for the reports
   before_script:
     - mkdir -p shippable/testresults
     - mkdir -p shippable/codecoverage

   script:
     - py.test

   archive: true
   ```

9. Commit the code and push it in your forked repository.

How to do it...

1. Log in to Shippable.
2. After logging in, click on **SYNC ACCOUNT** to get your forked repository listed, if it has not already been listed. Find and enable the repo that you want to build and run tests. For this example, I chose `flask-example` from my GitHub repos. After enabling it, you should see something like the following:



Projects

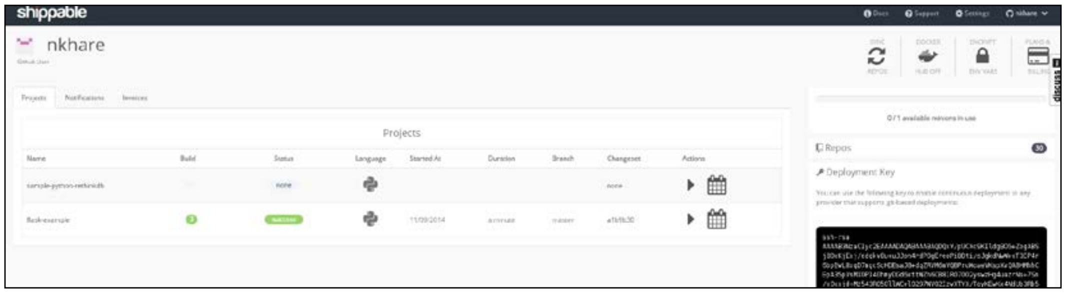
Name	Build	Status	Language	Started At	Duration	Branch	Changeset	Actions
flask-example		none					none	

3. Click on the play button and select branch to build. For this recipe, I chose master:

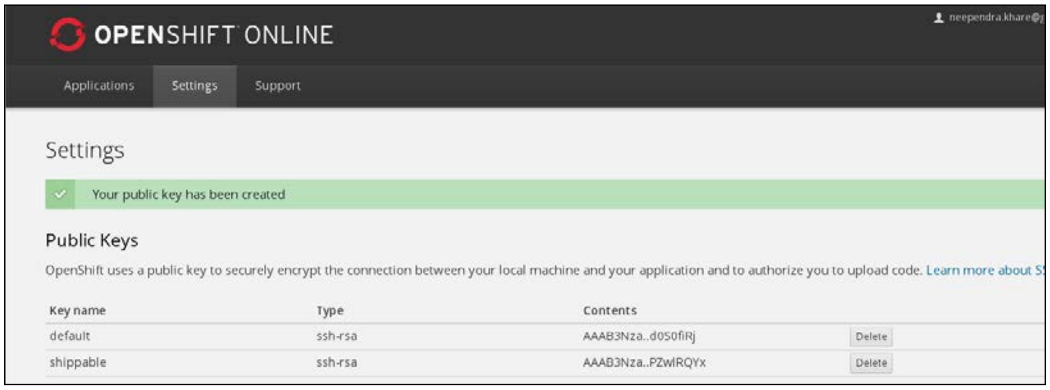
If the build is successful, then you will see the success icon.

Next time you do a commit in your repository, a build on Shippable will be triggered and the code will be tested. Now, to perform Continuous Deployment on OpenShift, let's follow the instructions provided on the Shippable website (`http://docs.shippable.com/deployment/openshift/`):

1. Get the deployment key from your Shippable dashboard (located on the right-hand side, below **Repos**):



2. Copy it under the (`https://openshift.redhat.com/app/console/settings`) **Settings** | **Public Keys** section on OpenShift as follows:



Settings

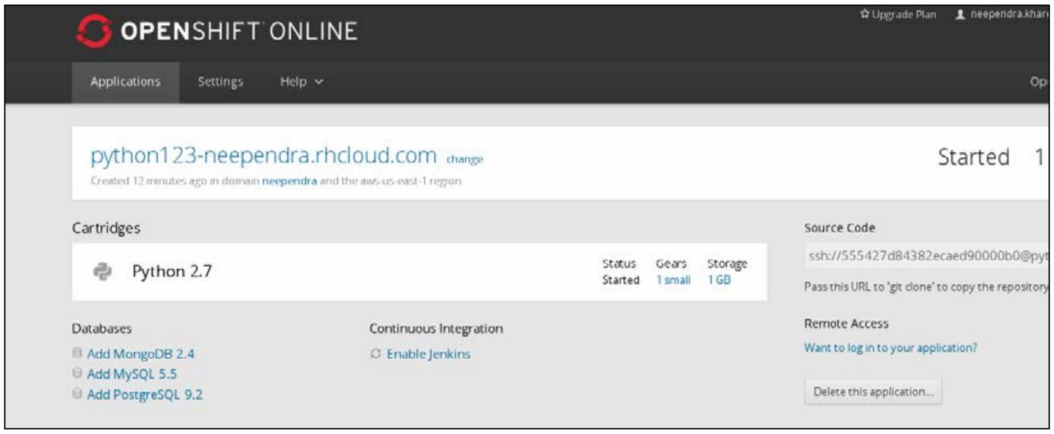
Your public key has been created

Public Keys

OpenShift uses a public key to securely encrypt the connection between your local machine and your application and to authorize you to upload code. Learn more about S

Key name	Type	Contents	
default	ssh-rsa	AAAB3Nza..d050fiRj	Delete
shippable	ssh-rsa	AAAB3Nza..PZwlRQYx	Delete

3. Get the **Source Code** repository link from the OpenShift application page, which will be used as `OPNESHIFT_REPO` in the next step:



python123-neependra.rhcloud.com

Created 12 minutes ago in domain neependra and the aws-us-east-1 region

Started 1

Cartridges

	Status	Gears	Storage
Python 2.7	Started	1 small	1 GB

Databases

Continuous Integration

Source Code

ssh://555427d84382ecaed90000b0@pyt

Pass this URL to 'git clone' to copy the repository

Remote Access

Want to log in to your application?

4. After the deployment key is installed, update the `shippable.yml` file as follows:

```
env:
  global:
    - OPENSHIFT_REPO=ssh://545ea4964382ec337f000009@blueprint-
neependra.rhcloud.com/~/git/blueprint.git

language: python

python:
  - 2.6
  - 2.7
```

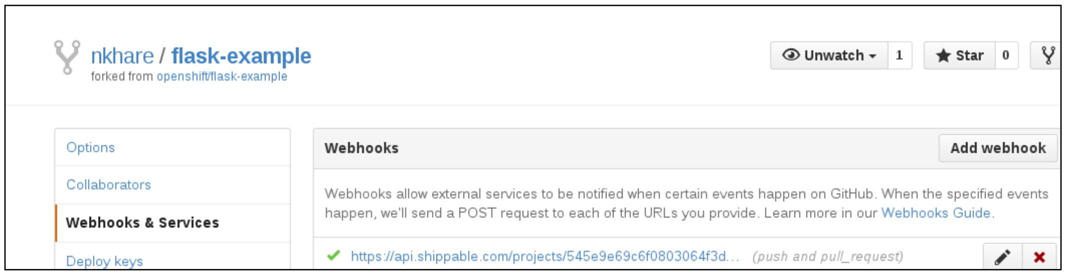
```
install:
  - pip install -r requirements.txt

# Make folders for the reports
before_script:
  - mkdir -p shippable/testresults
  - mkdir -p shippable/codecoverage
  - git remote -v | grep ^openshift || git remote add openshift
$OPENSHIFT_REPO
  - cd wsgi

script:
  - py.test

after_success:
  - git push -f openshift $BRANCH:master

archive: true
```

`OPENSHIFT_REPO` should reflect the app you have deployed using OpenShift. It will be different from what is shown in this example.

5. Now commit these changes and push it to GitHub. You will see a build on Shippable triggered and a new app deployed on OpenShift.
6. Visit your app's homepage, and you should see its updated contents.

How it works...

At every build instruction, Shippable spins off new containers depending on the image and language type specified in the `shippable.yml` file and runs the build to perform testing. In our case, Shippable will spin off two containers, one for Python 2.6 and the other for Python 2.7. Shippable adds a webhook to your GitHub repository as follows when you register it with them:



Webhooks allow external services to be notified when certain events happen on GitHub. When the specified events happen, we'll send a POST request to each of the URLs you provide. Learn more in our Webhooks Guide.

✓ https://api.shippable.com/projects/545e9e69c6f0803064f3d... (push and pull_request)

So every time a change is committed to GitHub, a build on Shippable gets triggered and after the success, it is deployed on OpenShift.

See also

- Detailed documentation is available on the Shippable website at `http://docs.shippable.com/`

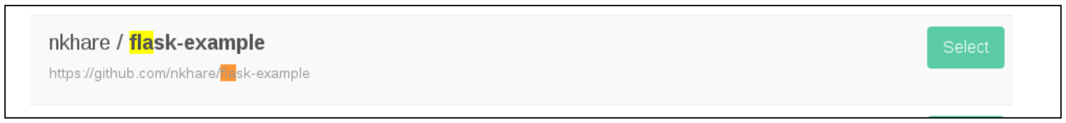
Doing CI/CD with Drone

As mentioned on the Drone website (`https://drone.io/`), Drone is a hosted Continuous Integration service. It enables you to conveniently set up projects to automatically build, test, and deploy as you make changes to your code. They provide an open source version of their platform, which you can host in your environment or on cloud. As of now, they support languages such as C/C++, Dart, Go, Haskell, Groovy, Java, Node.js, PHP, Python, Ruby, and Scala. Using Drone, you can deploy your application on platforms such as Heroku, Dotcloud, Google App Engine, and S3. You can also SSH (rsync) your code to a remote server for deployment.

For this recipe, let's use the same example that we used in the earlier recipes.

Getting ready

1. Log in to Drone (`https://drone.io/`).
2. Click on **New Project** and set up repository. In our case, we'll pick the same repository from GitHub that we used in the previous recipe (`https://github.com/nkhare/flask-example`):

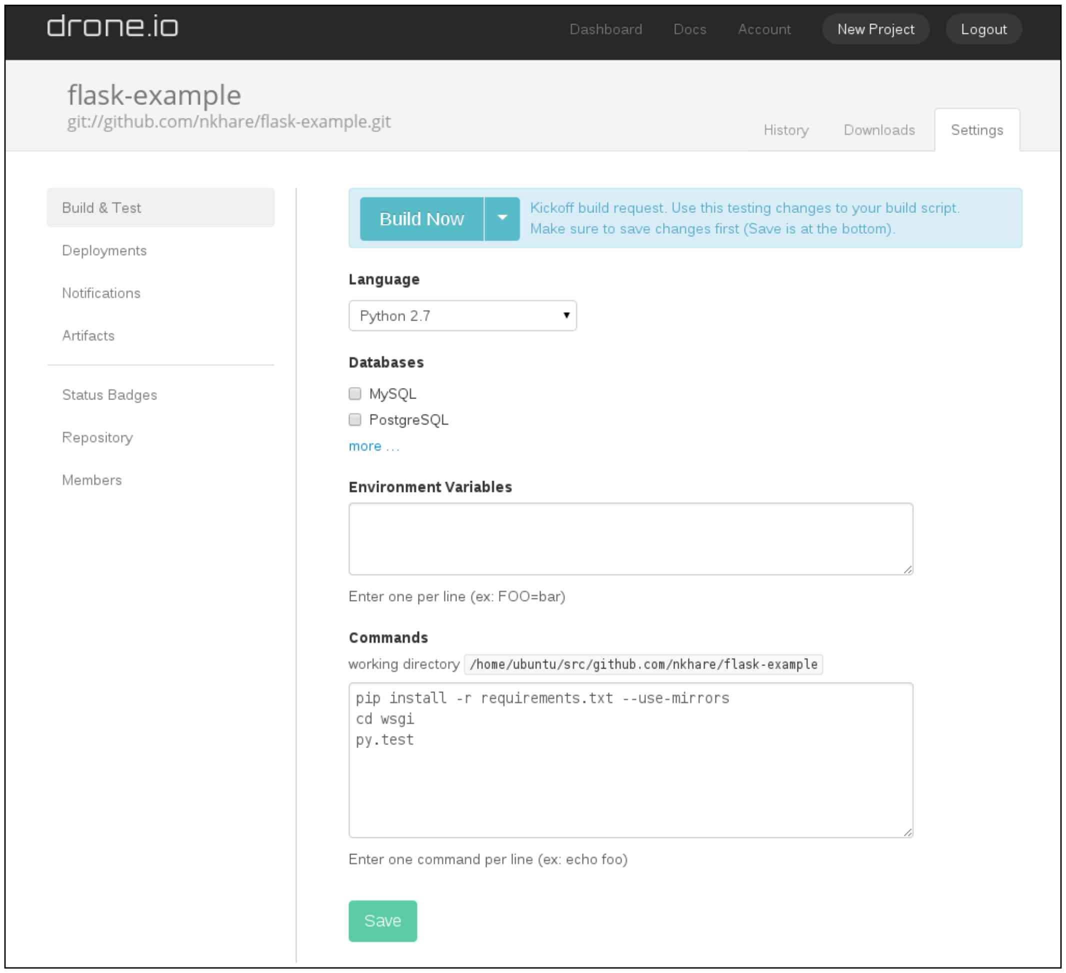

nkhare / flask-example
https://github.com/nkhare/flask-example

3. Once selected, it will ask you to select the programming language for the selected repository. I selected Python in this case.
4. It will then prompt you to set up the build script. For this recipe, we'll put the following and save it:

```
pip install -r requirements.txt --use-mirrors
cd wsgi
py.test
```

How to do it...

1. Trigger a manual build by clicking on **Build Now**, as shown in the following screenshot:


drone.io
Dashboard
Docs
Account
New Project
Logout
flask-example
git://github.com/nkhare/flask-example.git
History
Downloads
Settings
Build & Test
Deployments
Notifications
Artifacts
Status Badges
Repository
Members
Build Now
Kickoff build request. Use this testing changes to your build script. Make sure to save changes first (Save is at the bottom).
Language
Python 2.7
Databases
MySQL
PostgreSQL
more ...
Environment Variables
Enter one per line (ex: FOO=bar)
Commands
working directory /home/ubuntu/src/github.com/nkhare/flask-example
Enter one command per line (ex: echo foo)
Save

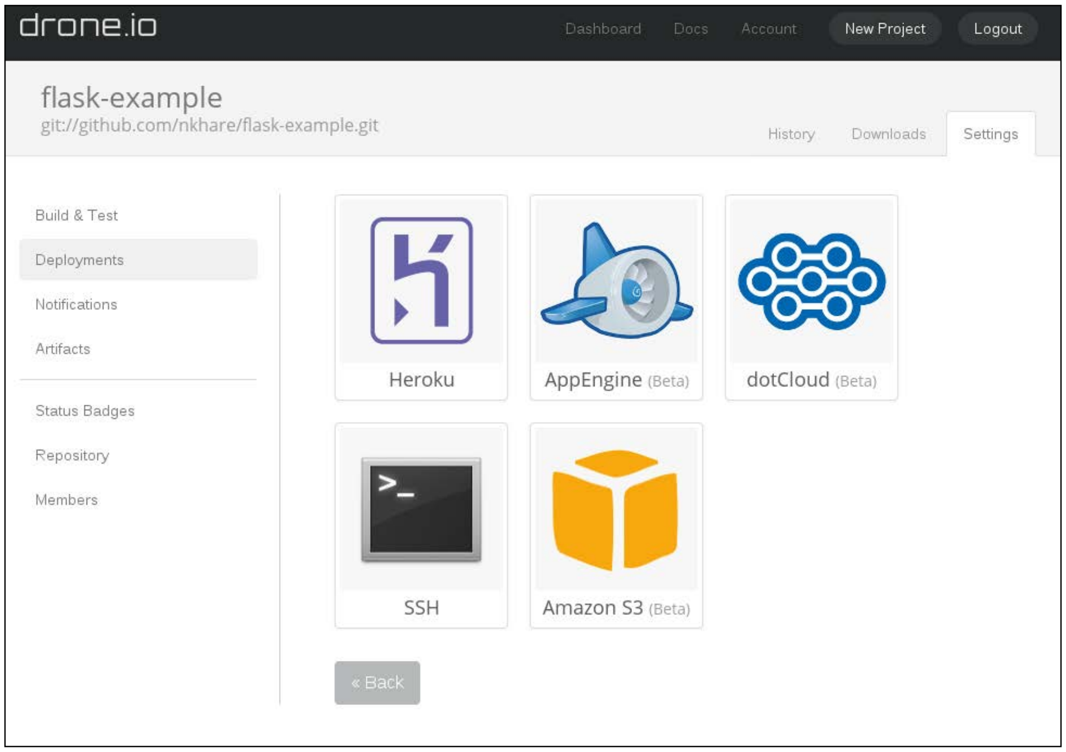
```
pip install -r requirements.txt --use-mirrors
cd wsgi
py.test
```

How it works...

The build process starts a new container, clones the source code repository, and runs the commands that we specified in the **Commands** section (running the test cases) within it.

There's more...

- Once the build is complete, you can look at the console output.
- Drone also adds a webhook in GitHub; so the next time you commit changes in the repository, a build will be triggered.
- Drone also supports Continuous Deployment to different cloud environments, as we have seen in the earlier recipe. To set that up, go to the **Settings** tab, select **Deployment**, and then select **Add New Deployment**. Select your cloud provider and set it up:



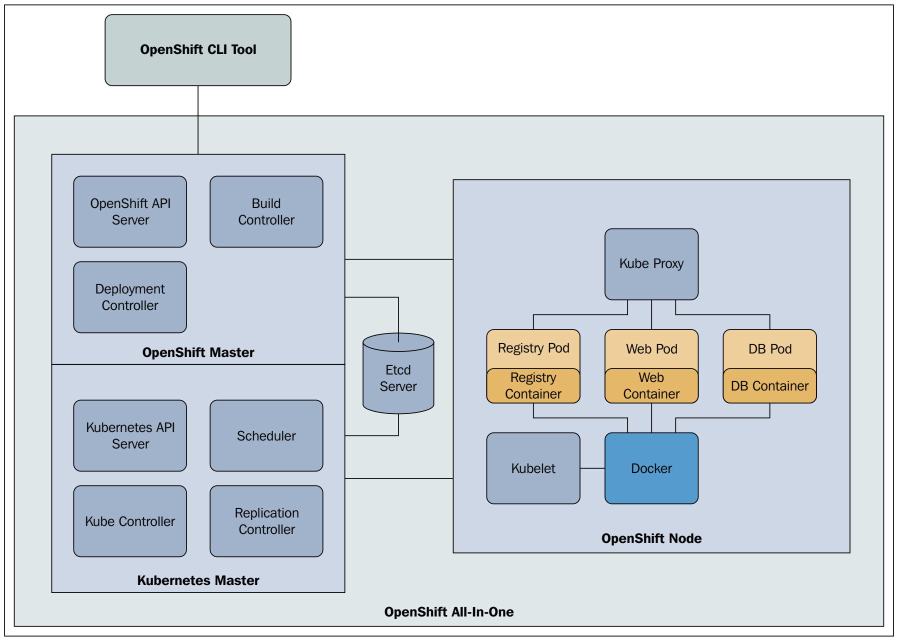
See also

- The Drone documentation at `http://docs.drone.io/`
- The steps to configure a self-hosted Drone environment, which is in the alpha stage as of now, at `https://github.com/drone/drone`

Setting up PaaS with OpenShift Origin

Platform-as-a-Service is a type of cloud service where the consumer controls the software deployments and configuration settings for applications (mostly web), and the provider provides servers, networks, and other services to manage those deployments. The provider can be external (a public provider) or internal (an IT department in an organization). There are many PaaS providers, such as Amazon (`http://aws.amazon.com/`), Heroku (`https://www.heroku.com/`), OpenShift (`https://www.openshift.com/`), and so on. In the recent past, containers seem to have become the natural choice for applications to get deployed to.

Earlier in this chapter, we looked at how we can build a CI/CD solution using Shippable and OpenShift, where we deployed our app to OpenShift PaaS. We deployed our app on Openshift Online, which is the Public Cloud Service. At the time of writing this book, the OpenShift Public Cloud Service uses non-Docker container technology to deploy apps to the Public Cloud Service. The OpenShift team has been working on OpenShift v3 (`https://github.com/openshift/origin`), which is a PaaS that leverages technologies such as Docker and Kubernetes (`http://kubernetes.io`) among others, providing a complete ecosystem to service your cloud-enabled apps. They plan to move this to the Public Cloud Service later this year. As we have talked about Kubernetes in *Chapter 8*, *Docker Orchestration and Hosting Platforms*, it is highly recommended to read that chapter first before continuing with this recipe. I am going to borrow some of the concepts from that chapter.



`https://blog.openshift.com/openshift-v3-deep-dive-docker-kubernetes/`

Kubernetes provides container cluster management with features such as scheduling pods and service discovery, but it does not have the concept of complete application, as well as the capabilities to build and deploy Docker images from the source code. OpenShift v3 extends the base Kubernetes model and fills those gaps. If we fast-forward and look at *Chapter 8*, *Docker Orchestration and Hosting Platforms*, for the Kubernetes section, you will notice that to deploy an app, we need to define Pods, Services, and Replication-Controllers. OpenShift v3 tries to abstract all that information and let you define one configuration file that takes care of all the internal wiring. Furthermore, OpenShift v3 provides other features such as automated deployment through source code push, the centralized administration and management of an application, authentication, team and project isolation, and resource tracking and limiting, all of which are required for enterprise deployment.

In this recipe, we will set up all-in-one OpenShift v3 Origin on a VM and start a pod. In the next recipe, we will see how to build and deploy an app through source code using the **Source-to-image** (**STI**) build feature. As there is active development happening on OpenShift v3 Origin, I have selected a tag from the source code and used that code-base in this recipe and the next one. In the newer version, the command-line options may change. With this information in hand, you should be able to adapt to the latest release. The latest example can be found at `https://github.com/openshift/origin/tree/master/examples/hello-openshift`.

Getting ready

Set up Vagrant (`https://www.vagrantup.com/`) and install the VirtualBox provider (`https://www.virtualbox.org/`). The instructions on how to set these up are outside the scope of this book.

1. Clone the OpenShift Origin repository:
   ```
   $ git clone https://github.com/openshift/origin.git
   ```
2. Check out the `v0.4.3` tag:
   ```
   $ cd origin
   $ git checkout tags/v0.4.3
   ```
3. Start the VM:
   ```
   $ vagrant up --provider=virtualbox
   ```
4. Log in to the container:
   ```
   $ vagrant ssh
   ```

How to do it...

1. Build the OpenShift binary:

```
$ cd /data/src/github.com/openshift/origin
$ make clean build
```

2. Go to the `hello-openshift` examples:

```
$  cd /data/src/github.com/openshift/origin/examples/hello-
openshift
```

3. Start all the OpenShift services in one daemon:

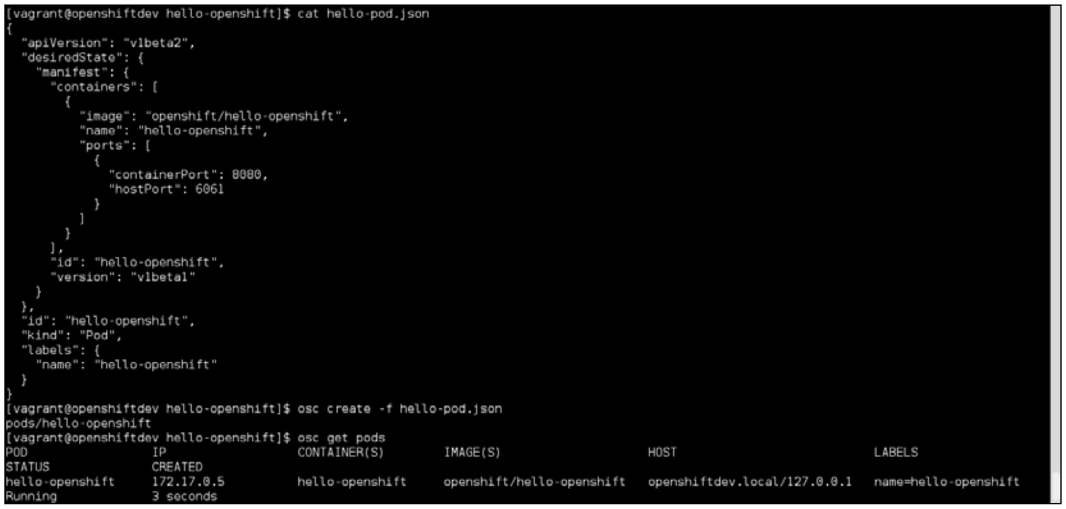
```
$ mkdir logs
$ sudo
/data/src/github.com/openshift/origin/_output/local/go/bin/ope
nshift start --public-master=localhost &> logs/openshift.log &
```

4. OpenShift services are secured by TLS. Our client will need to accept the server certificates and present its own client certificate. Those are generated as part of Openshift start in the current working directory.

```
$ export
OPENSHIFTCONFIG=`pwd`/openshift.local.certificates/admin/.kube
config
$ export
CURL_CA_BUNDLE=`pwd`/openshift.local.certificates/ca/cert.crt
$ sudo chmod a+rwX "$OPENSHIFTCONFIG"
```

5. Create the pod from the `hello-pod.json` definition:

```
$ osc create -f hello-pod.json
```



```
[vagrant@openshiftdev hello-openshift]$ cat hello-pod.json
{
  "apiVersion": "v1beta2",
  "desiredState": {
    "manifest": {
      "containers": [
        {
          "image": "openshift/hello-openshift",
          "name": "hello-openshift",
          "ports": [
            {
              "containerPort": 8080,
              "hostPort": 6061
            }
          ]
        }
      ],
      "id": "hello-openshift",
      "version": "v1beta1"
    }
  },
  "id": "hello-openshift",
  "kind": "Pod",
  "labels": {
    "name": "hello-openshift"
  }
}
[vagrant@openshiftdev hello-openshift]$ osc create -f hello-pod.json
pods/hello-openshift
[vagrant@openshiftdev hello-openshift]$ osc get pods
POD                IP                   CONTAINER(S)         IMAGE(S)                      HOST                            LABELS
STATUS             CREATED
hello-openshift    172.17.0.5           hello-openshift      openshift/hello-openshift     openshiftdev.local/127.0.0.1    name=hello-openshift
Running            3 seconds
```

6. Connect to the pod:

```
$ curl localhost:6061
```

How it works...

When OpenShift starts, all Kubernetes services start as well. Then, we connect to the OpenShift master through CLI and request it to start a pod. That request is then forwarded to Kubernetes, which starts the pod. In the pod configuration file, we mentioned to map port `6061` of the host machine with port `8080` of the pod. So, when we queried the host on port `6061`, we got a reply from the pod.

There's more...

If you run the `docker ps` command, you will see the corresponding containers running.

See also

- The *Learn More* section on `https://github.com/openshift/origin`
- The OpenShift 3 beta 3 Video tutorial at `https://blog.openshift.com/openshift-3-beta-3-training-commons-briefing-12/`
- The latest OpenShift training at `https://github.com/openshift/training`
- The OpenShift v3 documentation at `http://docs.openshift.org/latest/welcome/index.html`

Building and deploying an app on OpenShift v3 from the source code

OpenShift v3 provides the build process to build an image from source code. The following are the build strategies that one can follow to build images:

- **Docker build**: In this, users will supply to the Docker context (Dockerfiles and support files), which can be used to build images. OpenShift just triggers the `docker build` command to create the image.
- **Source-to-image (STI) build**: In this, the developer defines the source code repository and the builder image, which defines the environment used to create the app. STI then uses the given source code and builder image to create a new image for the app. More details about STI can be found at `https://github.com/openshift/source-to-image`.
- **Custom build**: This is similar to the Docker build strategy, but users might customize the builder image that will be used for build execution.

In this recipe, we are going to look at the STI build process. We are going to look at sample-app from the OpenShift v3 Origin repo (`https://github.com/openshift/origin/tree/v0.4.3/examples/sample-app`). The corresponding STI build file is located at `https://github.com/openshift/origin/blob/v0.4.3/examples/sample-app/application-template-stibuild.json`.

In the `BuildConfig` section, we can see that the source is pointing to a GitHub repo (`git://github.com/openshift/ruby-hello-world.git`) and the image under the `strategy` section is pointing to the `openshift/ruby-20-centos7` image. So, we will use the `openshift/ruby-20-centos7` image and build a new image using the source from the GitHub repo. The new image, after the build is pushed to the local or third-party Docker registry, depending on the settings. The `BuildConfig` section also defines triggers on when to trigger a new build, for instance, when the build image changes.

In the same STI build file (`application-template-stibuild.json`), you will find multiple `DeploymentConfig` sections, one of each pod. A `DeploymentConfig` section has information such as exported ports, replicas, the environment variables for the pod, and other info. In simple terms, you can think of `DeploymentConfig` as an extended replication controller of Kubernetes. It also has triggers to trigger new deployment. Each time a new deployment is created, the `latestVersion` field of `DeploymentConfig` is incremented. A `deploymentCause` is also added to `DeploymentConfig` describing the change that led to the latest deployment.

`ImageRepository`, which was recently renamed as `ImageStream`, is a stream of related images. `BuildConfig` and `DeploymentConfig` watch `ImageStream` to look for image changes and react accordingly, based on their respective triggers.

The other sections that you will find in the STI build file are services for pods (database and frontend), a route for the frontend service through which the app can be accessed, and a template. A template describes a set of resources intended to be used together that can be customized and processed to produce a configuration. Each template can define a list of parameters that can be modified for consumption by containers.

Similar to STI build, there are examples of Docker and custom build in the same sample-app example folder. I am assuming you have the earlier recipe, so we will continue from there.

Getting ready

You should have completed the earlier recipe, *Setting up PaaS with OpenShift Origin*.

Your current working directory should be `/data/src/github.com/openshift/origin / examples/hello-openshift` inside the VM, started by Vagrant.

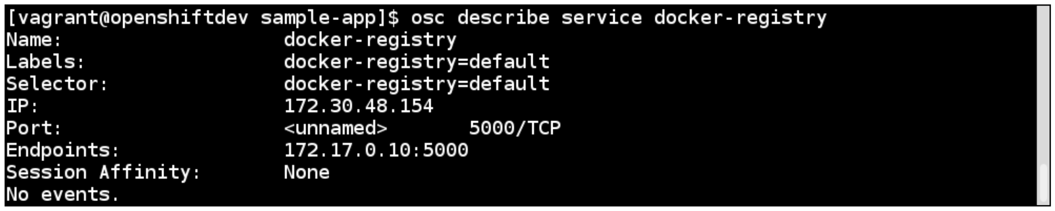
How to do it...

1. Deploy a private Docker registry to host images created by the STI build process:

   ```
   $ sudo openshift ex registry --create --
   credentials=./openshift.local.certificates/openshift-
   registry/.kubeconfig
   ```

2. Confirm the registry has started (this can take a few minutes):

   ```
   $ osc describe service docker-registry
   ```



```
[vagrant@openshiftdev sample-app]$ osc describe service docker-registry
Name:                   docker-registry
Labels:                 docker-registry=default
Selector:               docker-registry=default
IP:                     172.30.48.154
Port:                   <unnamed>       5000/TCP
Endpoints:              172.17.0.10:5000
Session Affinity:       None
No events.
```

3. Create a new project in OpenShift. This creates a namespace `test` to contain the builds and an app that we will generate later:

   ```
   $ openshift ex new-project test --display-name="OpenShift 3
   Sample" --description="This is an example project to
   demonstrate OpenShift v3" --admin=test-admin
   ```

4. Log in with the `test-admin` user and switch to the `test` project, which will be used by every command from now on:

   ```
   $ osc login -u test-admin -p pass
   ```

   ```
   $ osc project test
   ```

5. Submit the application template for processing (generating shared parameters requested in the template) and then request the creation of the processed template:

   ```
   $ osc process -f application-template-stibuild.json | osc
   create -f -
   ```

6. This will not trigger the build. To start the build of your application, run the following command:

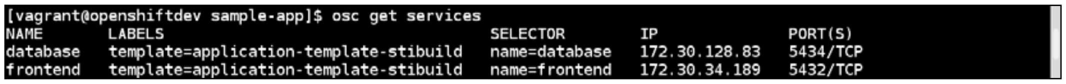
   ```
   $ osc start-build ruby-sample-build
   ```

7. Monitor the build and wait for the status to go to `complete` (this can take a few minutes):

   ```
   $ osc get builds
   ```

8. Get the list of services:

   ```
   $ osc get services
   ```



```
[vagrant@openshiftdev sample-app]$ osc get services
NAME       LABELS                                 SELECTOR        IP              PORT(S)
database   template=application-template-stibuild  name=database   172.30.128.83   5434/TCP
frontend   template=application-template-stibuild  name=frontend   172.30.34.189   5432/TCP
```

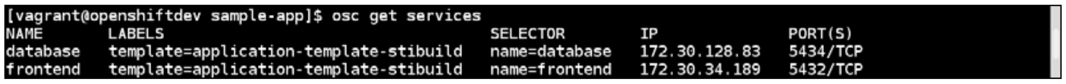
How it works...

In the `BuildConfig` (`ruby-sample-build`) section, we specified our source as the `ruby-hello-world` Git repo (`git://github.com/openshift/ruby-hello-world.git`) and our image as `openshift/ruby-20-centos7`. So the build process takes that image, and with STI builder, a new image called `origin-ruby-sample` is created after building our source on `openshift/ruby-20-centos7`. The new image is then pushed to the Docker registry we created earlier.

With `DeploymentConfig`, frontend and backend pods are also deployed and linked to corresponding services.

There's more...

- The preceding frontend service can be accessed through the service IP and corresponding port, but it will not be accessible from the outside world. To make it accessible, we give our app an FQDN; for instance, in the following example, it is defined as `www.example.com`:



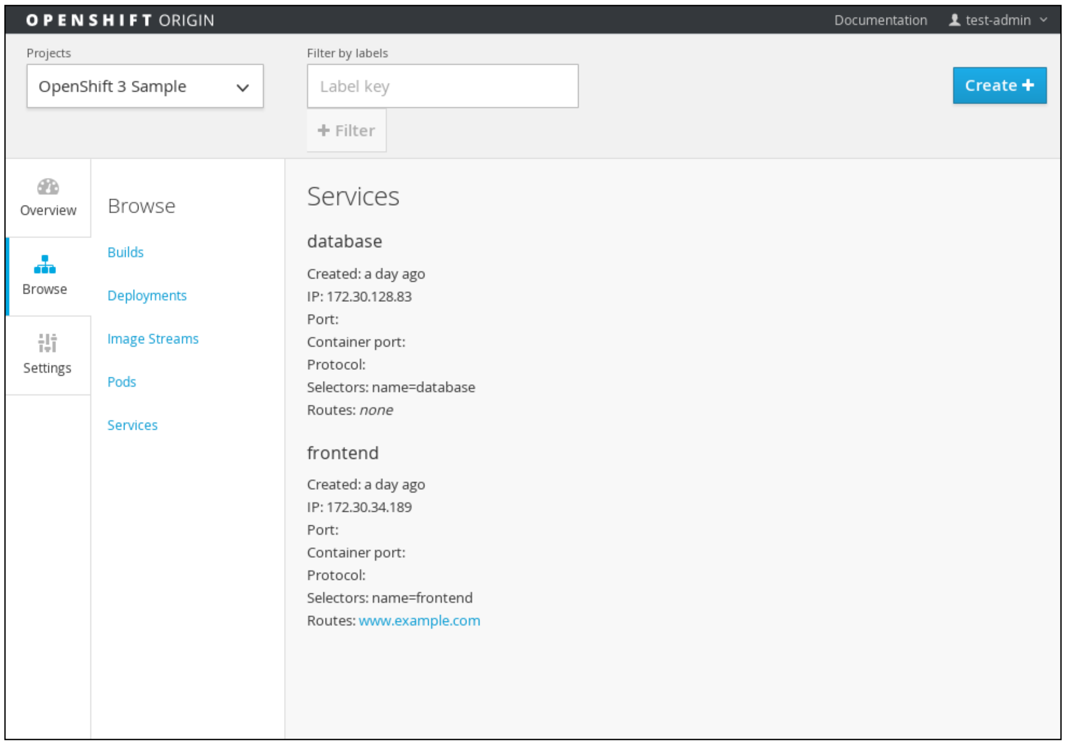
```
[vagrant@openshiftdev sample-app]$ osc get services
NAME       LABELS                                 SELECTOR        IP              PORT(S)
database   template=application-template-stibuild  name=database   172.30.128.83   5434/TCP
frontend   template=application-template-stibuild  name=frontend   172.30.34.189   5432/TCP
```

OpenShift v3 provides an HAProxy router, which can map over FQDN to the corresponding pod. For more information, please visit `http://docs.openshift.org/latest/architecture/core_objects/routing.html`. You will also require an entry in the external DNS to resolve the FQDN provided here.

- OpenShift v3 Origin is also a management GUI. To look at our deployed app on the GUI, bind the username `test-admin` to the view role in the default namespace so you can observe the progress in the web console:

  ```
  $ openshift ex policy add-role-to-user view test-admin
  ```

  Then, through the browser, connect to `https://<host>:8443/console` and log in through the `test-admin` user by giving any password. As Vagrant forwards the traffic of port `8443` on the host machine to the VM, you should be able to connect through the host on which VM is running. Then select **OpenShift 3 Sample** as the project and explore:



Services

database

Created: a day ago
IP: 172.30.128.83
Port:
Container port:
Protocol:
Selectors: name=database
Routes: *none*

frontend

Created: a day ago
IP: 172.30.34.189
Port:
Container port:
Protocol:
Selectors: name=frontend
Routes: www.example.com

- In the multiple node setup, your pods can be scheduled on different systems. OpenShift v3 connects pods though the overlay network pod running on one node can access another. It is called `openshift-sdn`. For more details, please visit `https://github.com/openshift/openshift-sdn`.

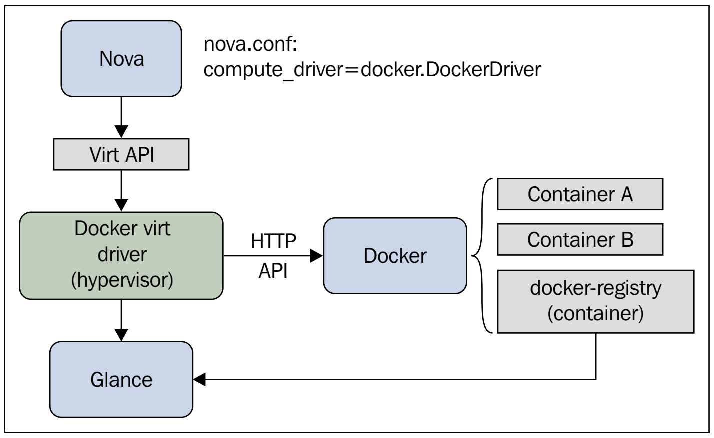
See also

- The *Learn More* section at `https://github.com/openshift/origin`
- The OpenShift 3 beta 3 video tutorial at `https://blog.openshift.com/openshift-3-beta-3-training-commons-briefing-12/`
- The latest OpenShift training at `https://github.com/openshift/training`
- The OpenShift v3 documentation at `http://docs.openshift.org/latest/welcome/index.html`

Configuring Docker as a hypervisor driver for OpenStack

I am assuming that the reader has some exposure to OpenStack for this recipe, as covering it is outside the scope of this book. For more information on OpenStack and its components, please visit `http://www.openstack.org/software/`.

In OpenStack, Nova supports different hypervisors for computation, such as KVM, XEN, VMware, HyperV, and others. We can provision VMs using these drivers. Using Ironic (`https://wiki.openstack.org/wiki/Ironic`), you can provision bare metal as well. Nova added support for containers provisioning using Docker in the Havana (`https://www.openstack.org/software/havana/`) release, but currently, it lives out of the mainline for faster dev cycle. There are plans to merge it in the mainline in the future. Under the hood, it looks like this:


Nova
nova.conf:
compute_driver=docker.DockerDriver
Virt API
Docker virt driver (hypervisor)
HTTP API
Docker
Container A
Container B
docker-registry (container)
Glance

`https://wiki.openstack.org/wiki/File:Docker-under-the-hood.png`

DevStack (`http://docs.openstack.org/developer/devstack/overview.html`) is a collection of scripts to quickly create an OpenStack development environment. It is not a general-purpose installer, but it is a very easy way to get started with OpenStack. In this recipe, we'll configure DevStack's environment with Docker as Nova driver on Fedora21.

Getting ready

1. Install Docker on the system.
2. Clone `nova-docker` and `devstack`:

   ```
   $ git clone https://git.openstack.org/stackforge/nova-docker
   /opt/stack/nova-docker
   ```

   ```
   $ git clone https://git.openstack.org/openstack-dev/devstack
   /opt/stack/devstack
   ```

3. The following step is needed until we can make use of `configure_nova_hypervisor_rootwrap`:

   ```
   $ git clone https://git.openstack.org/openstack/nova
   /opt/stack/nova
   ```

4. Prepare Devstack for installation:

   ```
   $ cd /opt/stack/nova-docker
   $ ./contrib/devstack/prepare_devstack.sh
   ```

5. Create the stack user and add it to `sudo`:

   ```
   $ /opt/stack/devstack/tools/create-stack-user.sh
   ```

6. Install `docker-py` to communicate with docker through Python:

   ```
   $ yum install python-pip
   $ pip install docker-py
   ```

How to do it…

1. After the prerequisite steps are completed, run the following commands to install Devstack:

   ```
   $ cd /opt/stack/devstack
   $ ./stack.sh
   ```

How it works…

- The `prepare_devstack.sh` driver makes the following entries in the `localrc` file set the right environment to set Docker for the Nova driver:

  ```
  export VIRT_DRIVER=docker
  export DEFAULT_IMAGE_NAME=cirros
  export NON_STANDARD_REQS=1
  export IMAGE_URLS=" "
  ```

- After running the `stackrc` file, we can see the following changes with respect to Nova and Glance:
  - The `/etc/nova/nova.conf` file changes the compute driver:

    ```
    [DEFAULT]
    compute_driver = novadocker.virt.docker.DockerDriver
    ```

  - The `/etc/nova/rootwrap.d/docker.filters` file is updated with the following content:

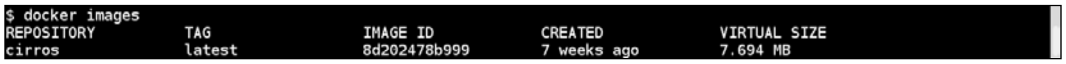
    ```
    [Filters]
    # nova/virt/docker/driver.py: 'ln', '-sf',
    '/var/run/netns/.*'
    ln: CommandFilter, /bin/ln, root
    ```

  - In `/etc/glance/glance-api.conf`, adds `docker` in the container/ image format:

    ```
    [DEFAULT]
    container_formats = ami,ari,aki,bare,ovf,docker
    ```

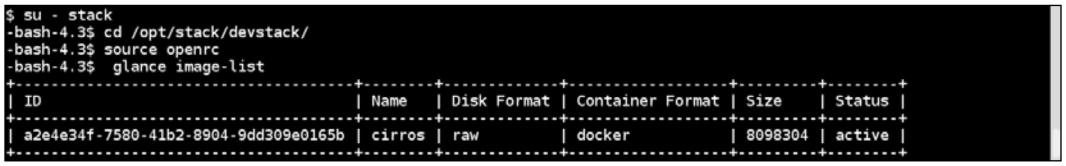
There's more…

- In `localrc`, we mentioned `cirros` as the default image, so once the setup is completed, we can see that the Docker image for `cirros` is downloaded:



```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
cirros              latest              8d202478b999        7 weeks ago         7.694 MB
```

  This is being imported to Glance automatically:



```
$ su - stack
-bash-4.3$ cd /opt/stack/devstack/
-bash-4.3$ source openrc
-bash-4.3$  glance image-list
+--------------------------------------+--------+-------------+------------------+---------+--------+
| ID                                   | Name   | Disk Format | Container Format | Size    | Status |
+--------------------------------------+--------+-------------+------------------+---------+--------+
| a2e4e34f-7580-41b2-8904-9dd309e0165b | cirros | raw         | docker           | 8098304 | active |
+--------------------------------------+--------+-------------+------------------+---------+--------+
```

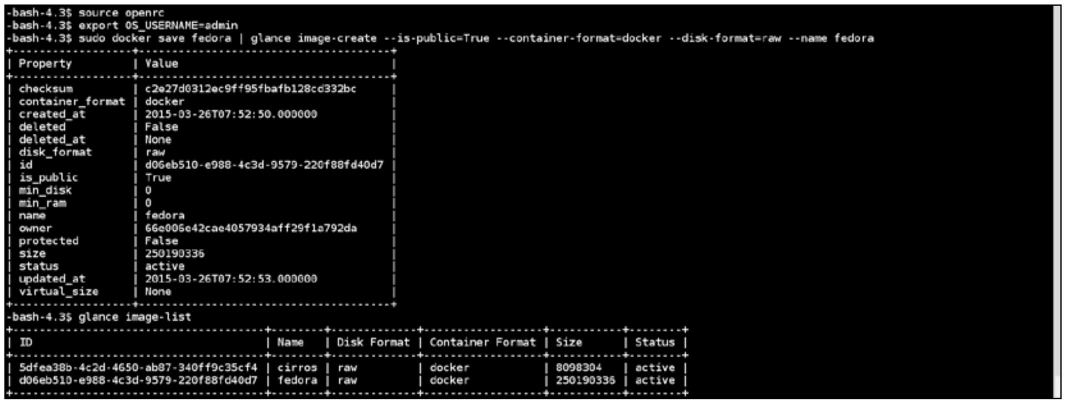
From the preceding screenshot, we can see that the container format is Docker.

- Now you can create an instance using a `cirros` image using Horizon, or from the command line, and look at the container started using the Docker command line.
- To import any image to Glance, you can do something like the following:
    - Pull the required image from Docker Hub:

      ```
      $ docker pull fedora
      ```

    - Import the image (currently only admin can import the image):

      ```
      $ source openrc
      $ export OS_USERNAME=admin
      $ sudo docker save fedora | glance image-create --is-
      public=True --container-format=docker --disk-format=raw
      --name fedora
      ```



```
-bash-4.3$ source openrc
-bash-4.3$ export OS_USERNAME=admin
-bash-4.3$ sudo docker save fedora | glance image-create --is-public=True --container-format=docker --disk-format=raw --name fedora
+------------------+--------------------------------------+
| Property         | Value                                |
+------------------+--------------------------------------+
| checksum         | c2e27d0312ec9ff95fbafb128cd332bc     |
| container_format | docker                               |
| created_at       | 2015-03-26T07:52:50.000000           |
| deleted          | False                                |
| deleted_at       | None                                 |
| disk_format      | raw                                  |
| id               | d06eb510-e988-4c3d-9579-220f88fd40d7 |
| is_public        | True                                 |
| min_disk         | 0                                    |
| min_ram          | 0                                    |
| name             | fedora                               |
| owner            | 66e006e42cae4057934aff29f1a792da     |
| protected        | False                                |
| size             | 250190336                            |
| status           | active                               |
| updated_at       | 2015-03-26T07:52:53.000000           |
| virtual_size     | None                                 |
+------------------+--------------------------------------+
-bash-4.3$ glance image-list
+--------------------------------------+--------+-------------+------------------+-----------+--------+
| ID                                   | Name   | Disk Format | Container Format | Size      | Status |
+--------------------------------------+--------+-------------+------------------+-----------+--------+
| 5dfea38b-4c2d-4650-ab87-340ff9c35cf4 | cirros | raw         | docker           | 8098304   | active |
| d06eb510-e988-4c3d-9579-220f88fd40d7 | fedora | raw         | docker           | 250190336 | active |
+--------------------------------------+--------+-------------+------------------+-----------+--------+
```

- There is a lack of integration with Cinder and Neutron, but things are catching up quickly.
- While installing, if you get the `AttributeError: 'module' object has no attribute 'PY2' error`, then run the following commands to fix it:

  ```
  $ pip uninstall  six
  $ pip install --upgrade   six
  ```

See also

- The documentation on OpenStack website at `https://wiki.openstack.org/wiki/Docker`.
- Docker is also one of the resource types for OpenStack Heat. Learn more about it at `http://docs.openstack.org/developer/heat/template_guide/contrib.html#dockerinc-resource`.
- There is an interesting project in OpenStack called Kolla, which focuses on deploying OpenStack services through Docker containers. Find more about it at `https://github.com/stackforge/kolla/`.

6
Docker APIs and Language Bindings

In this chapter, we will cover the following recipes:

- Configuring the Docker daemon remote API
- Performing image operations using remote APIs
- Performing container operations using remote APIs
- Exploring Docker remote API client libraries
- Securing the Docker daemon remote API

Introduction

In the previous chapters, we learned different commands to manage images, containers, and so on. Though we run all the commands through the command line, the communication between the Docker client (CLI) and the Docker daemon happens through APIs, which are called Docker daemon remote APIs.

Docker also provides APIs to communicate with Docker Hub and Docker registry, which the Docker client uses as well. In addition to these APIs, we have Docker bindings for different programming languages. So, if you want to build a nice GUI for Docker images, container management, and so on, understanding the APIs mentioned earlier would be a good starting point.

In this chapter, we look into the Docker daemon remote API and use the `curl` command (`http://curl.haxx.se/docs/manpage.html`) to communicate with the endpoints of different APIs, which will look something like the following command:

```
$ curl -X <REQUEST> -H <HEADER> <OPTION> <ENDPOINT>
```

The preceding request will return with a return code and an output corresponding to the endpoint and request we chose. `GET`, `PUT`, and `DELETE` are the different kinds of requests, and GET is the default request if nothing is specified. Each API endpoint has its own interpretation for the return code.

Configuring the Docker daemon remote API

As we know, Docker has a client-server architecture. When we install Docker, a user space program and a daemon get started from the same binary. The daemon binds to `unix://var/run/docker.sock` by default on the same host. This will not allow us to access the daemon remotely. To allow remote access, we need to start Docker such that it allows remote access, which can done by changing the `-H` flag appropriately.

Getting ready

Depending on the Linux distribution you are running, figure out the Docker daemon configuration file you need to change. For Fedora, /Red Hat distributions, it would be `/etc/sysconfig/docker` and for Ubuntu/Debian distributions , it would most likely be `/etc/default/docker`.

How to do it...

1. On Fedora 20 systems, add the `-H tcp://0.0.0.0:2375` option in the configuration file (`/etc/sysconfig/docker`), as follows:

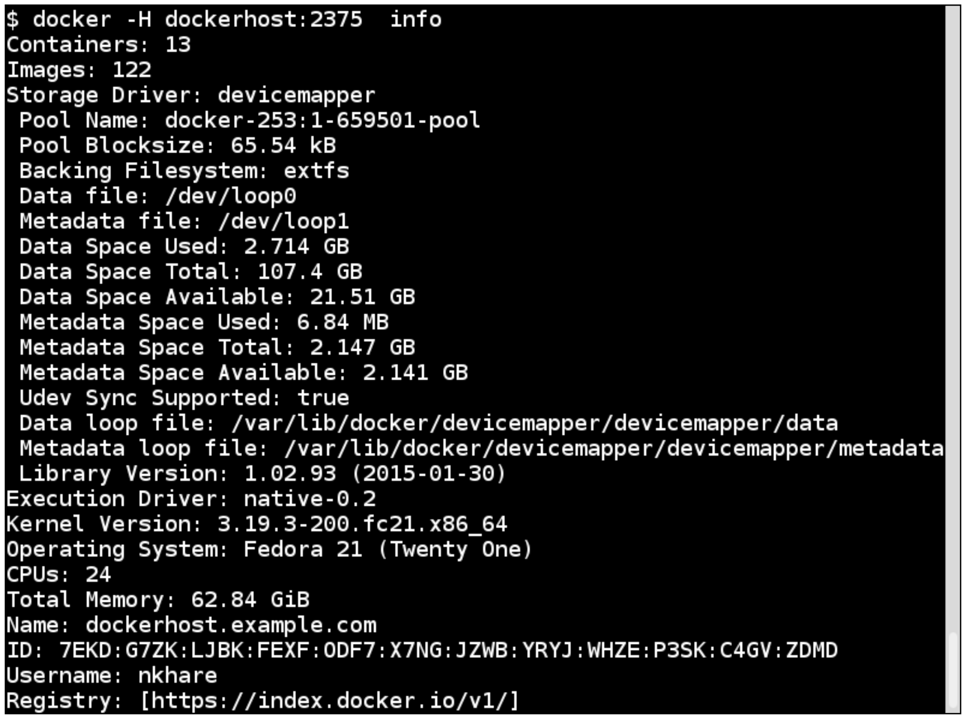
   ```
   OPTIONS=--selinux-enabled -H tcp://0.0.0.0:2375
   ```

2. Restart the Docker service. On Fedora, run the following command:

   ```
   $ sudo systemctl restart docker
   ```

3. Connect to the Docker host from the remote client:

   ```
   $ docker -H <Docker Host>:2375 info
   ```



```
$ docker -H dockerhost:2375  info
Containers: 13
Images: 122
Storage Driver: devicemapper
 Pool Name: docker-253:1-659501-pool
 Pool Blocksize: 65.54 kB
 Backing Filesystem: extfs
 Data file: /dev/loop0
 Metadata file: /dev/loop1
 Data Space Used: 2.714 GB
 Data Space Total: 107.4 GB
 Data Space Available: 21.51 GB
 Metadata Space Used: 6.84 MB
 Metadata Space Total: 2.147 GB
 Metadata Space Available: 2.141 GB
 Udev Sync Supported: true
 Data loop file: /var/lib/docker/devicemapper/devicemapper/data
 Metadata loop file: /var/lib/docker/devicemapper/devicemapper/metadata
 Library Version: 1.02.93 (2015-01-30)
Execution Driver: native-0.2
Kernel Version: 3.19.3-200.fc21.x86_64
Operating System: Fedora 21 (Twenty One)
CPUs: 24
Total Memory: 62.84 GiB
Name: dockerhost.example.com
ID: 7EKD:G7ZK:LJBK:FEXF:ODF7:X7NG:JZWB:YRYJ:WHZE:P3SK:C4GV:ZDMD
Username: nkhare
Registry: [https://index.docker.io/v1/]
```

Make sure the firewall allows access to port `2375` on the system where the Docker daemon is installed.

How it works...

With the preceding command, we allowed the Docker daemon to listen on all network interfaces through port `2375`, using TCP.

There's more...

- With the communication that we mentioned earlier between the client and Docker, the host is insecure. Later in this chapter, we'll see how to enable TLS between them.
- The Docker CLI looks for environment variables; if it is being set then the CLI uses that endpoint to connect, for example, if we connect set as follows:

  ```
  $ export DOCKER_HOST=tcp://dockerhost.example.com:2375
  ```

  Then, the future docker commands in that session connect to remote Docker Host by default and run this:

  ```
  $ docker info
  ```

See also

- The documentation on the Docker website `https://docs.docker.com/reference/api/docker_remote_api/`

Performing image operations using remote APIs

After enabling the Docker daemon remote API, we can do all image-related operations through a client. To get a better understanding of the APIs, let's use `curl` to connect to the remote daemon and do some image-related operations.

Getting ready

Configure the Docker daemon and allow remote access, as explained in the previous recipe.

How to do it...

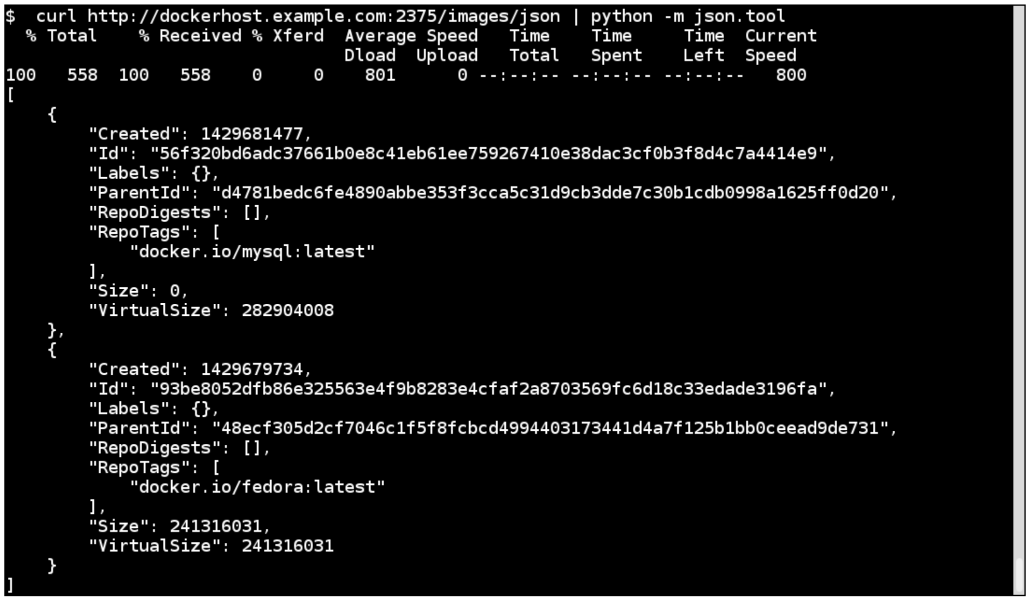
In this recipe, we'll look at a few image operations as follows:

1. To list images, use the following API:

   `GET /images/json`

   Here is an example of the preceding syntax:

   ```
   $ curl http://dockerhost.example.com:2375/images/json
   ```



```
$  curl http://dockerhost.example.com:2375/images/json | python -m json.tool
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   558  100   558    0     0    801      0 --:--:-- --:--:-- --:--:--   800
[
    {
        "Created": 1429681477,
        "Id": "56f320bd6adc37661b0e8c41eb61ee759267410e38dac3cf0b3f8d4c7a4414e9",
        "Labels": {},
        "ParentId": "d4781bedc6fe4890abbe353f3cca5c31d9cb3dde7c30b1cdb0998a1625ff0d20",
        "RepoDigests": [],
        "RepoTags": [
            "docker.io/mysql:latest"
        ],
        "Size": 0,
        "VirtualSize": 282904008
    },
    {
        "Created": 1429679734,
        "Id": "93be8052dfb86e325563e4f9b8283e4cfaf2a8703569fc6d18c33edade3196fa",
        "Labels": {},
        "ParentId": "48ecf305d2cf7046c1f5f8fcbcd4994403173441d4a7f125b1bb0ceead9de731",
        "RepoDigests": [],
        "RepoTags": [
            "docker.io/fedora:latest"
        ],
        "Size": 241316031,
        "VirtualSize": 241316031
    }
]
```

2. To create an image, use the following API:

   ```
   POST /images/create
   ```

   Here are a few examples:

   - Get the Fedora image from Docker Hub:

     ```
     $ curl -X POST
     http://dockerhost.example.com:2375/images/
     create?fromImage=fedora
     ```

   - Get the WordPress image with the `latest` tag:

     ```
     $  curl -X POST
     http://dockerhost.example.com:2375/images/create?fromImage=
     wordpress&tag=latest
     ```

   - Create an image from the `tar` file, which is hosted on the accessible web server:

     ```
     $ curl -X POST
     http://dockerhost.example.com:2375/images/
     create?fromSrc=http://localhost/image.tar
     ```

3. To build an image, use the following API:

   ```
   POST  /commit
   ```

   Here are a few examples:

   - Build an image from the container (`container id = 704a7c71f77d`)

     ```
     $ curl -X POST
     http://dockerhost.example.com:2375/
     commit?container=704a7c71f77d
     ```

   - Build an image from the Docker file:

     ```
     $  curl -X POST  -H "Content-type:application/tar" --data-
     binary '@/tmp/Dockerfile.tar.gz'
     http://dockerhost.example.com:2375/build?t=apache
     ```

   As the API expects the content as a `tar` file, we need to put the Docker file inside a tar and call the API.

4. To delete an image, use the following API:

   ```
   DELETE  /images/<name>
   ```

   Here is an example of the preceding syntax:

   ```
   $ curl -X DELETE
   http://dockerhost.example.com:2375/images/wordpress:3.9.1
   ```

How it works...

In all the cases mentioned earlier, the APIs will connect to the Docker daemon and perform the requested operations.

There's more...

We have not covered all the options of the APIs discussed earlier and Docker provides APIs for other image-related operations. Visit the API documentation for more details.

See also

- Each API endpoint can have different inputs to control the operations. For more details, visit the documentation on the Docker website `https://docs.docker.com/reference/api/docker_remote_api_v1.18/#22-images`.

Performing container operations using remote APIs

In a similar way to how we performed image operations using APIs, we can also do all container-related operations using APIs.

Getting ready

Configure the Docker daemon and allow remote access, as explained in the earlier recipe.

How to do it...

In this recipe, we'll look at a few container operations:

1. To list containers, use the following API:

   ```
   GET   /containers/json
   ```

   Here are a few examples:

   - Get all the running containers:

     ```
     $ curl -X GET
     http://shadowfax.example.com:2375/containers/json
     ```

   - Get all the running containers, including the stopped ones

     ```
     $ curl -X GET http://shadowfax.example.com:2375/containers/
     json?all=True
     ```

2. To create a new container, use the following API:

   ```
   POST   /containers/create
   ```

   Here are a few examples

   - Create a container from the `fedora` image:

     ```
     $ curl -X POST  -H "Content-type:application/json" -d
     '{"Image": "fedora", "Cmd": ["ls"] }'
     http://dockerhost.example.com:2375/containers/create
     ```

   - Create a container from the `fedora` image and name it `f21`:

     ```
     $ curl -X POST  -H "Content-type:application/json" -d
     '{"Image": "fedora", "Cmd": ["ls"] }'
     http://dockerhost.example.com:2375/containers/
     create?name=f21
     ```

3. To start a container, use the following API:

   ```
   POST /containers/<id>/start
   ```

   For example, start a container with the `591ab8ac2650` ID:

   ```
   $ curl -X POST  -H "Content-type:application/json" -d '{"Dns":
   ["4.2.2.1"] }'
   http://dockerhost.example.com:2375/containers/591ab8ac2650/sta
   rt
   ```

   Note that while starting the stopped container, we also passed the DNS option, which will change the DNS configuration of the container.

4. To inspect a container, use the following API:

   ```
   GET  /containers/<id>/json
   ```

   For example, inspect a container with the `591ab8ac2650` ID:

   ```
   $ curl -X GET
   http://dockerhost.example.com:2375/containers/591ab8ac2650/json
   ```

5. To get a list of processes running inside a container, use the following API:

   ```
   GET /containers/<id>/top
   ```

   For example, get the processes running in the container with the `591ab8ac2650` ID:

   ```
   $ curl -X GET
   http://dockerhost.example.com:2375/containers/591ab8ac2650/top
   ```

6. To stop a container, use the following API:

   ```
   POST /containers/<id>/stop
   ```

   For example, stop a container with the `591ab8ac2650` ID:

   ```
   $ curl -X POST
   http://dockerhost.example.com:2375/containers/591ab8ac2650/sto
   p
   ```

How it works...

We have not covered all the options of the APIs discussed earlier and Docker provides APIs for other container-related operations. Visit the API documentation for more details.

See also

- The documentation on the Docker website at `https://docs.docker.com/reference/api/docker_remote_api_v1.18/#21-containers`

Exploring Docker remote API client libraries

In the last few recipes, we explored the APIs provided by Docker to connect and perform operations to the remote Docker daemon. The Docker community has added bindings for different programming languages to access those APIs. Some of them are listed at `https://docs.docker.com/reference/api/remote_api_client_libraries/`.

Note that Docker Maintainers do not maintain these libraries. Let's explore Python bindings with a few examples and see how it uses the Docker remote API.

Getting ready

- Install `docker-py` on Fedora:

  ```
  $ sudo yum install python-docker-py
  ```

  Alternatively, use `pip` to install the package:

  ```
  $ sudo pip install docker-py
  ```

- Import the module:

  ```
  $ python
  >>> import docker
  ```

How to do it...

1. Create the client, using the following steps:
   1. Connect through the Unix Socket:

      ```
      >>> client =
      docker.Client(base_url='unix://var/run/docker.sock',
      version='1.18',  timeout=10)
      ```

   2. Connect over HTTP:

      ```
      >>> client =
      docker.Client(base_url='http://dockerhost.example.com:2375',
      version='1.18',  timeout=10)
      ```

   Here, `base_url` is the endpoint to connect, `version` is the API version the client will use, and `timeout` is the timeout value in seconds.

2. Search for an image using the following code:

   ```
   >>> client.search ("fedora")
   ```

3. Pull an image using the following code:

   ```
   >>> client.pull("fedora", tag="latest")
   ```

4. Start a container using the following code:

   ```
   >>> client.create_container("fedora", command="ls",
   hostname=None, user=None, detach=False, stdin_open=False,
   tty=False, mem_limit=0, ports=None, environment=None,
   dns=None, volumes=None,
   volumes_from=None,network_disabled=False, name=None,
   entrypoint=None, cpu_shares=None,
   working_dir=None,memswap_limit=0)
   ```

How it works...

In all the preceding cases, the Docker Python module will send RESTful requests to the endpoint using the API provided by Docker. Look at the methods such as `search`, `pull`, and `start` in the following code of `docker-py` available at `https://github.com/docker/docker-py/blob/master/docker/client.py`.

There's more...

You can explore different user interfaces written for Docker. Some of them are as follows:

- Shipyard (`http://shipyard-project.com/`)—written in Python
- DockerUI (`https://github.com/crosbymichael/dockerui`)—written in JavaScript using AngularJS

Securing the Docker daemon remote API

Earlier in this chapter, we saw how to configure the Docker daemon to accept remote connections. However, with the approach we followed, anyone can connect to our Docker daemon. We can secure our connection with Transport Layer Security (`http://en.wikipedia.org/wiki/Transport_Layer_Security`).

We can configure TLS either by using the existing **Certificate Authority** (**CA**) or by creating our own. For simplicity, we will create our own, which is not recommended for production. For this example, we assume that the host running the Docker daemon is `dockerhost.example.com`.

Getting ready

Make sure you have the `openssl` library installed.

How to do it...

1. Create a directory on your host to put our CA and other related files:

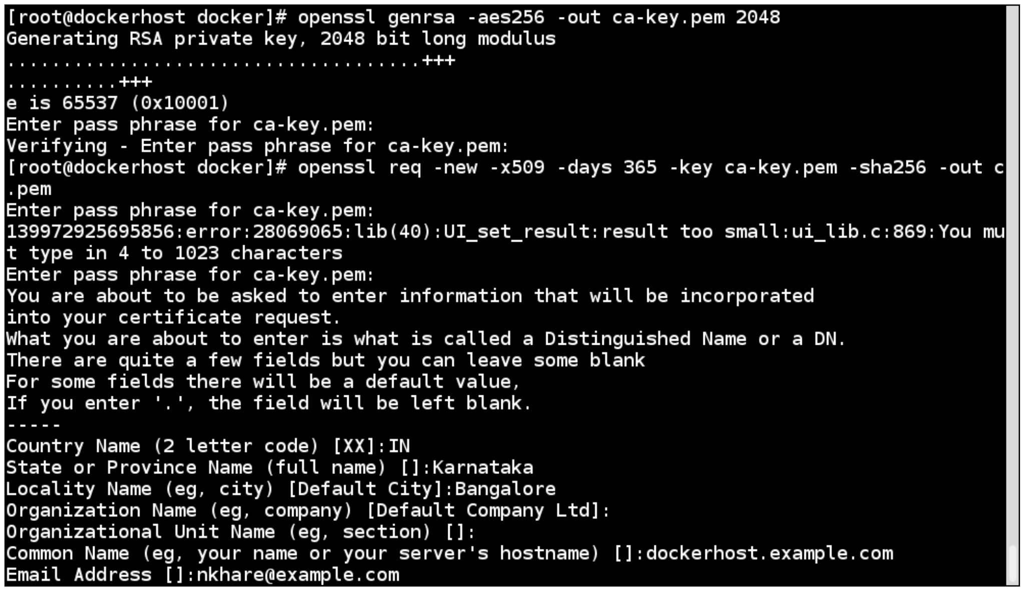
   $ mkdirc-p /etc/docker

   $ cd /etc/docker

2. Create the CA private and public keys:

   $ openssl genrsa -aes256 -out ca-key.pem 2048

   $ openssl req -new -x509 -days 365 -key ca-key.pem -sha256 -out ca.pem

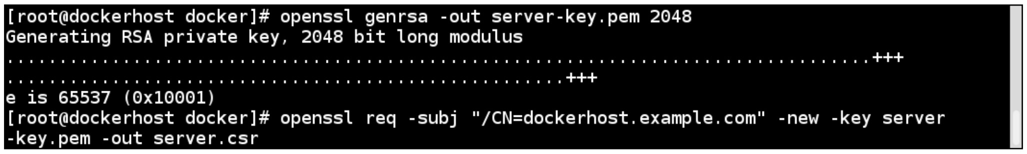


```
[root@dockerhost docker]# openssl genrsa -aes256 -out ca-key.pem 2048
Generating RSA private key, 2048 bit long modulus
....................................+++
.........+++
e is 65537 (0x10001)
Enter pass phrase for ca-key.pem:
Verifying - Enter pass phrase for ca-key.pem:
[root@dockerhost docker]# openssl req -new -x509 -days 365 -key ca-key.pem -sha256 -out c
.pem
Enter pass phrase for ca-key.pem:
139972925695856:error:28069065:lib(40):UI_set_result:result too small:ui_lib.c:869:You mu
t type in 4 to 1023 characters
Enter pass phrase for ca-key.pem:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [XX]:IN
State or Province Name (full name) []:Karnataka
Locality Name (eg, city) [Default City]:Bangalore
Organization Name (eg, company) [Default Company Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (eg, your name or your server's hostname) []:dockerhost.example.com
Email Address []:nkhare@example.com
```

3. Now, let's create the server key and certificate signing request. Make sure that `Common Name` matches the Docker daemon system's hostname. In our case, it is `dockerhost.example.com`.

   $ openssl genrsa -out server-key.pem 2048

   $ openssl req -subj "/CN=dockerhost.example.com" -new -key server-key.pem -out server.csr

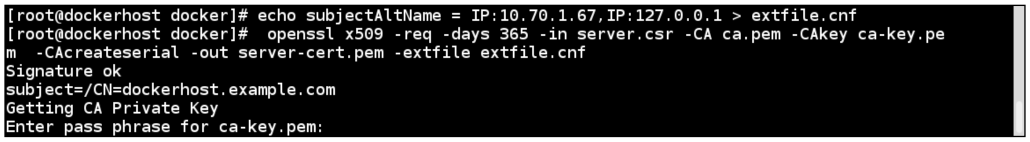


```
[root@dockerhost docker]# openssl genrsa -out server-key.pem 2048
Generating RSA private key, 2048 bit long modulus
...........................................................................................+++
..........................................................+++
e is 65537 (0x10001)
[root@dockerhost docker]# openssl req -subj "/CN=dockerhost.example.com" -new -key server
-key.pem -out server.csr
```

4. To allow connections from 127.0.0.1 and a specific host, for example, 10.70.1.67, create an extensions configuration file and sign the public key with our CA:

   ```
   $ echo subjectAltName = IP:10.70.1.67,IP:127.0.0.1 > extfile.cnf
   ```

   ```
   $ openssl x509 -req -days 365 -in server.csr -CA ca.pem -CAkey
   ca-key.pem    -CAcreateserial -out server-cert.pem -extfile
   extfile.cnf
   ```

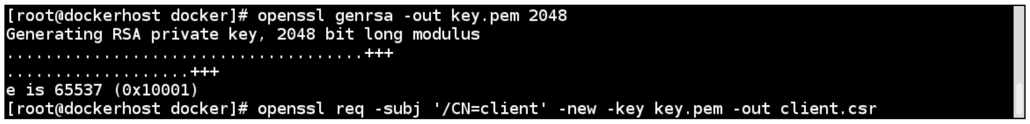


```
[root@dockerhost docker]# echo subjectAltName = IP:10.70.1.67,IP:127.0.0.1 > extfile.cnf
[root@dockerhost docker]#  openssl x509 -req -days 365 -in server.csr -CA ca.pem -CAkey ca-key.pe
m  -CAcreateserial -out server-cert.pem -extfile extfile.cnf
Signature ok
subject=/CN=dockerhost.example.com
Getting CA Private Key
Enter pass phrase for ca-key.pem:
```

5. For client authentication, create a client key and certificate signing request:

   ```
   $ openssl genrsa -out key.pem 2048
   ```

   ```
   $ openssl req -subj '/CN=client' -new -key key.pem -out client.csr
   ```

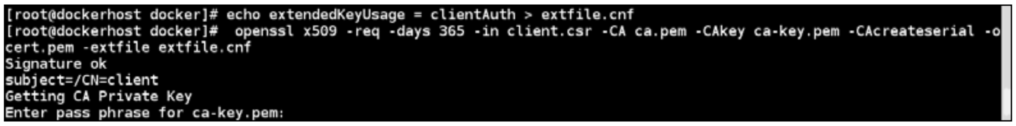


```
[root@dockerhost docker]# openssl genrsa -out key.pem 2048
Generating RSA private key, 2048 bit long modulus
.....................................+++
..................+++
e is 65537 (0x10001)
[root@dockerhost docker]# openssl req -subj '/CN=client' -new -key key.pem -out client.csr
```

6. To make the key suitable for client authentication, create an extensions configuration file and sign the public key:

   ```
   $ echo extendedKeyUsage = clientAuth > extfile_client.cnf
   ```

   ```
   $ openssl x509 -req -days 365 -in client.csr -CA ca.pem -CAkey
   ca-key.pem  -CAcreateserial -out cert.pem -extfile_client.cnf
   ```



```
[root@dockerhost docker]# echo extendedKeyUsage = clientAuth > extfile.cnf
[root@dockerhost docker]#  openssl x509 -req -days 365 -in client.csr -CA ca.pem -CAkey ca-key.pem -CAcreateserial -o
cert.pem -extfile extfile.cnf
Signature ok
subject=/CN=client
Getting CA Private Key
Enter pass phrase for ca-key.pem:
```

7. After generating `cert.pem` and `server-cert.pem`, we can safely remove both the certificate signing requests:

   ```
   $ rm -rf client.csr server.csr
   ```

8. To set tight security and protect keys from accidental damage, let's change the permissions:

   ```
   $ chmod -v 0600 ca-key.pem key.pem server-key.pem ca.pem server-
   cert.pem cert.pem
   ```

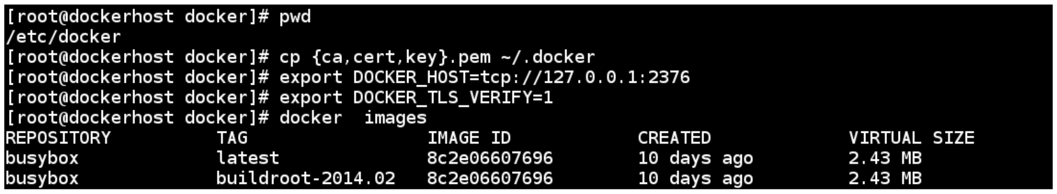
9. Stop the daemon if it is running on `dockerhost.example.com`. Then, start the Docker daemon manually from `/etc/docker`:

```
 $ pwd
   /etc/docker
    $ docker -d --tlsverify --tlscacert=ca.pem --
tlscert=server-cert.pem    --tlskey=server-key.pem   -
H=0.0.0.0:2376
```

10. From another terminal, go to `/etc/docker`. Run the following command to connect to the Docker daemon:

```
$ cd /etc/docker
$ docker --tlsverify --tlscacert=ca.pem --tlscert=cert.pem --
tlskey=key.pem -H=127.0.0.1:2376 version
```

    You will see that a TLS connection is established and you can run the commands over it. You can also put the CA public key and the client's TLS certificate and key in the `.docker` folder in the home directory of the user and use the `DOCKER_HOST` and `DOCKER_TLS_VERIFY` environment variables to make a secure connection by default.



```
[root@dockerhost docker]# pwd
/etc/docker
[root@dockerhost docker]# cp {ca,cert,key}.pem ~/.docker
[root@dockerhost docker]# export DOCKER_HOST=tcp://127.0.0.1:2376
[root@dockerhost docker]# export DOCKER_TLS_VERIFY=1
[root@dockerhost docker]# docker  images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
busybox             latest              8c2e06607696        10 days ago         2.43 MB
busybox             buildroot-2014.02   8c2e06607696        10 days ago         2.43 MB
```

11. To connect from the remote host we mentioned while signing the server key with our CA, we will need to copy the CA public key and the client's TLS certificate and key to the remote machine and then connect to the Docker host as shown in the preceding screenshot.

How it works...

We setup the TLS connection between the Docker daemon and the client for a secure communication.

There's more...

- To set up the Docker daemon to start with the TLS configuration by default, we will need to update the Docker configuration file. For example, on Fedora, you update the `OPTIONS` parameter as follows in `/etc/sysconfig/docker`:

  ```
  OPTIONS='--selinux-enabled -H tcp://0.0.0.0:2376 --tlsverify
  --tlscacert=/etc/docker/ca.pem --tlscert=/etc/docker/server-
  cert.pem --tlskey=/etc/docker/server-key.pem'
  ```

- If you recall, in *Chapter 1*, *Introduction and Installation*, we saw how we can set up the Docker host using the Docker Machine (`http://docs.docker.com/machine/`) and as part of this setup, the TLS setup happens between the client and the host running the Docker daemon. After configuring the Docker host with the Docker Machine, check `.docker/machine` for the user on the client system.

7
Docker Performance

In this chapter, we will cover the following recipes:

- Benchmarking CPU performance
- Benchmarking disk performance
- Benchmarking network performance
- Getting container resource usage using the stats feature
- Setting up performance monitoring

Introduction

In *Chapter 3*, *Working with Docker Images*, we saw, how Dockerfiles can be used to create images consisting of different services/software and later in *Chapter 4*, *Network and Data Management for Containers*, we saw, how one Docker container can talk to the outside world with respect to data and network. In *Chapter 5*, *Docker Use Cases*, we looked into the different use cases of Docker, and in *Chapter 6*, *Docker APIs and Language Bindings*, we looked at how to use remote APIs to connect to a remote Docker host.

Ease of use is all good, but before going into production, performance is one of the key aspects that is considered. In this chapter, we'll see the performance impacting features of Docker and what approach we can follow to benchmark different subsystems. While doing performance evaluation, we need to compare Docker performance against the following:

- Bare metal
- Virtual machine
- Docker running inside a virtual machine

In the chapter, we will look at the approach you can follow to do performance evaluation rather than performance numbers collected from runs to do comparison. However, I'll point out performance comparisons done by different companies, which you can refer to.

Let's first look at some of the Docker performance impacting features:

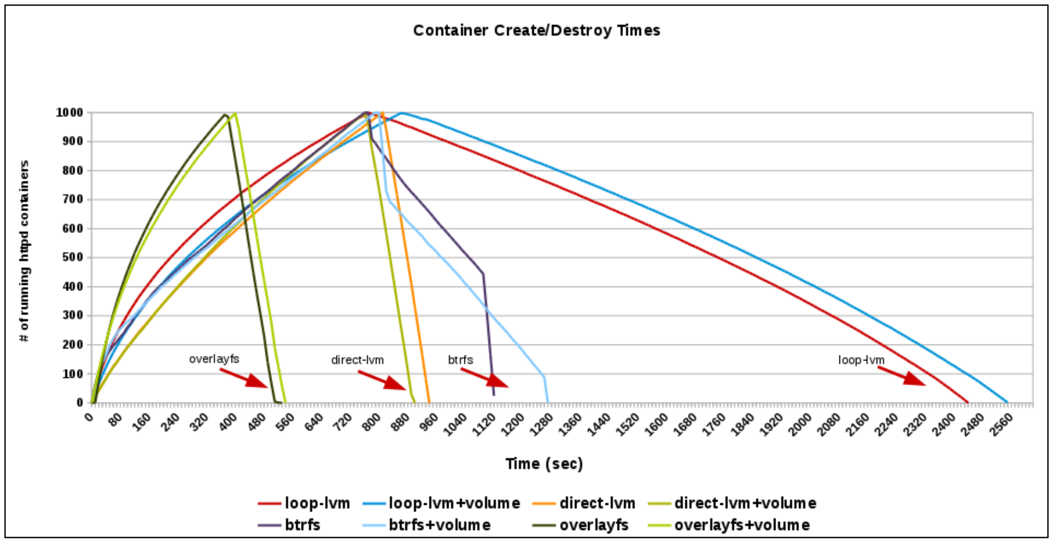
- **Volumes**: While putting down any enterprise class workload, you would like to tune the underlying storage accordingly. You should not use the primary/root filesystem used by containers to store data. Docker provides the facility to attach/mount external storage through volumes. As we have seen in *Chapter 4*, *Network and Data Management for Containers*, there are two types of volumes, which are as follows:
    - Volumes that are mounted through host machines using the `--volume` option
    - Volumes that are mounted through another container using the `--volumes-from` option
- **Storage drivers**: We looked at different storage drivers in *Chapter 1*, *Installation and Introduction*, which are vfs, aufs, btrfs, devicemapper, and overlayFS. Support for ZFS has been merged recently as well. You can check the currently supported storage drivers and their priority of selection if nothing is chosen as the Docker start time at `https://github.com/docker/docker/blob/master/daemon/graphdriver/driver.go`.

  If you are running Fedora, CentOS, or RHEL, then the device mapper will be the default storage driver. You can find some device mapper specific tuning at `https://github.com/docker/docker/tree/master/daemon/graphdriver/devmapper`.

  You can change the default storage driver with the `-s` option to the Docker daemon. You can update the distribution-specific configuration/systems file to make changes across service restart. For Fedora/RHEL/CentOS, you will have the update `OPTIONS` field in `/etc/sysconfig/docker`. Something like the following to use the `btrfs` backend:

  ```
  OPTIONS=-s btrfs
  ```

The following graph shows you how much time it takes to start and stop 1,000 containers with different configurations of storage driver:



http://developerblog.redhat.com/2014/09/30/overview-storage-scalability-docker/

As you can see, overlayFS performs better than other storage drivers.

- **--net=host**: As we know, by default, Docker creates a bridge and associates IPs from it to the containers. Using `--net=host` exposes host networking stack to the container by skipping the creation of a network namespace for the container. From this, it is clear that this option always gives better performance compared to the bridged one.

  This has some limitations, such as not being able to have two containers or host apps listening on the same port.

- **Cgroups**: Docker's default execution driver, `libcontainer`, exposes different Cgroups knobs, which can be used to fine tune container performance. Some of them are as follows:

  - **CPU shares**: With this, we can give proportional weight to the containers and accordingly the resource will be shared. Consider the following example:

    ```
    $ docker run -it -c 100 fedora bash
    ```

  - **CPUsets**: This allows you to create CPU masks, using which execution of threads inside a container on host CPUs is controlled. For example, the following code will run threads inside a container on the 0th and 3rd core:

    ```
    $ docker run -it  --cpuset=0,3 fedora bash
    ```

  - **Memory limits**: We can set memory limits to a container. For example, the following command will limit the memory usage to 512 MB for the container:

    ```
    $ docker run -it -m 512M fedora bash
    ```

- **Sysctl and ulimit settings**: In a few cases, you might have to change some of the `sysclt` values depending on the use case to get optimal performance, such as changing the number of open files. With Docker 1.6 (`https://docs.docker.com/v1.6/release-notes/`) and above we can change the `ulimit` settings with the following command:

  ```
  $ docker run -it --ulimit data=8192 fedora bash
  ```

  The preceding command will change the settings for just that given container, it is a per container tuning variable. We can also set some of these settings through the systemd configuration file of Docker daemon, which will be applicable to all containers by default. For example, looking at the systemd configuration file for Docker on Fedora, you will see something like the following in the service section:

  ```
  LimitNOFILE=1048576  # Open file descriptor setting
  LimitNPROC=1048576   # Number of processes settings
  LimitCORE=infinity   # Core size settings
  ```

  You can update this as per your need.

You can learn about Docker performance by studying the work done by others. Over the last year, some Docker performance-related studies have been published by a few companies:

- From Red Hat:
  - Performance Analysis of Docker on Red Hat Enterprise Linux:

    `http://developerblog.redhat.com/2014/08/19/performance-analysis-docker-red-hat-enterprise-linux-7/`

    `https://github.com/jeremyeder/docker-performance`

  - Comprehensive Overview of Storage Scalability in Docker:

    `http://developerblog.redhat.com/2014/09/30/overview-storage-scalability-docker/`

  - Beyond Microbenchmarks—breakthrough container performance with Tesla efficiency:

    `http://developerblog.redhat.com/2014/10/21/beyond-microbenchmarks-breakthrough-container-performance-with-tesla-efficiency/`

- Containerizing Databases with Red Hat Enterprise Linux:

  `http://rhelblog.redhat.com/2014/10/29/containerizing-databases-with-red-hat-enterprise-linux/`

- From IBM
  - An Updated Performance Comparison of Virtual Machines and Linux Containers:

    `http://domino.research.ibm.com/library/cyberdig.nsf/papers/0929052195DD819C85257D2300681E7B/$File/rc25482.pdf`

    `https://github.com/thewmf/kvm-docker-comparison`

- From VMware
  - Docker Containers Performance in VMware vSphere

    `http://blogs.vmware.com/performance/2014/10/docker-containers-performance-vmware-vsphere.html`

To do the benchmarking, we need to run similar workload on different environments (bare metal/VM/Docker) and then collect the results with the help of different performance stats. To simplify things, we can write common benchmark scripts which can be used to run on different environments. We can also create Dockerfiles to spin off containers with workload generation scripts. For example, in the *Performance Analysis of Docker on Red Hat Enterprise Linux* article, which is listed earlier (`https://github.com/jeremyeder/docker-performance/blob/master/Dockerfiles/Dockerfile`), the author has used a Dockerfile to create a CentOS image and used the `container` environment variable to select Docker and non-Docker environment for benchmark script `run-sysbench.sh`.

Similarly, Dockerfiles and related scripts are published by IBM for their study available at `https://github.com/thewmf/kvm-docker-comparison`.

We will be using some of the Docker files and scripts mentioned earlier in the recipes of this chapter.

Benchmarking CPU performance

We can use benchmarks such as Linpack (`http://www.netlib.org/linpack/`) and sysbench (`https://github.com/nuodb/sysbench`) to benchmark CPU performance. For this recipe, we'll use sysbench. We'll see how to run the benchmark on bare metal and inside the container. Similar steps can be performed in other environments, as mentioned earlier.

Getting ready

We will use the CentOS 7 container to run the benchmark inside the container. Ideally, we should have a system with CentOS 7 installed to get benchmark results on bare metal. For the container test, let's build the image from the GitHub repository that we referred to earlier:

```
$ git clone https://github.com/jeremyeder/docker-performance.git
$ cd docker-performance/Dockerfiles/
$ docker build -t c7perf --rm=true - < Dockerfile
$ docker images
REPOSITORY            TAG             IMAGE ID          CREATED
VIRTUAL SIZE
c7perf               latest         59a10df39a82     About a minute ago
678.3 MB
```

How to do it...

Inside the same GitHub repository, we have a script to run sysbench, `docker-performance/bench/sysbench/run-sysbench.sh`. It has some configurations, which you can modify according to your needs.

1. As the root user, create the `/results` directory on the host:

   ```
   $ mkdir -p /results
   ```

   Now, run the benchmark after setting the container environment variable to something other than Docker, which we used while building the `c7perf` image on the host machine, run the following commands:

   ```
   $ cd docker-performance/bench/sysbench
   $ export container=no
   $ sh ./run-sysbench.sh  cpu test1
   ```

   By default, the results are collected in `/results`. Make sure you have write access to it or change the `OUTDIR` parameter in the benchmark script.

2. To run the benchmark inside the container, we need to first start the container and then run the benchmark script:

   ```
   $ mkdir /results_container
   $ docker run -it -v /results_container:/results c7perf bash
   $ docker-performance/bench/sysbench/run-sysbench.sh cpu test1
   ```

   As we mounted the host directory, `/results_container`, inside the `/results` container, the result will be collected on the host.

3. While running the preceding test on Fedora/RHEL/CentOS, where SELinux is enabled, you will get a `Permission denied` error. To fix it, relabel the host directory while mounting it inside the container as follows:

   ```
   $ docker run -it -v /results_container:/results:z c7perf bash
   ```

   Alternatively, for the time being, put SELinux in permissive mode:

   ```
   $  setenforce 0
   ```

   Then, after the test, put it back in permissive mode:

   ```
   $  setenforce 1
   ```

   Refer to *Chapter 9*, *Docker Security*, for more details about SELinux.

How it works...

The benchmark script internally calls sysbench's CPU benchmark for the given input. CPU is benchmarked by using the 64-bit integer manipulation using Euklid algorithms for prime number computation. The result for each run gets collected in the corresponding results directory, which can be used for comparison.

There's more...

Almost no difference is reported in bare metal and Docker CPU performance.

See also

- Look at the CPU benchmark results published in IBM and VMware using Linpack in the links referenced earlier in this chapter.

Benchmarking disk performance

There are tools such as Iozone (`http://www.iozone.org/`), smallfile (`https://github.com/bengland2/smallfile`), and Flexible IO (`https://github.com/axboe/fio`) available to benchmark disk performance. For this recipe, we will use FIO. For that, we need to write a job file, which mimics the workload you want to run. Using this job file, we can simulate the workload on the target. For this recipe, let's take the FIO example from the benchmark results, which IBM has published (`https://github.com/thewmf/kvm-docker-comparison/tree/master/fio`).

Getting ready

In the bare metal / VM / Docker container, install FIO and mount the disk containing a filesystem for each test under `/ferrari` or anything which is mentioned in the FIO job file. On bare metal, you can mount natively and on VM it can be mounted using the virtual disk driver or we can do device pass through. On Docker, we can attach the filesystem from the host machine using Docker volumes.

Prepare the workload file. We can pick `https://github.com/thewmf/kvm-docker-comparison/blob/master/fio/mixed.fio`:

```
[global]
ioengine=libaio
direct=1
size=16g
group_reporting
thread
filename=/ferrari/fio-test-file

[mixed-random-rw-32x8]
stonewall
rw=randrw
rwmixread=70
bs=4K
iodepth=32
numjobs=8
runtime=60
```

Using the preceding job file, we can do random direct I/O on `/ferrari/fio-test-file` with 4K block size using the `libaio` driver on a 16 GB file. The I/O depth is 32 and the number of parallel jobs is 8. It is a mix workload, which does 70 percent read and 30 percent write.

How to do it...

1. For the bare metal and VM tests, you can just run the FIO job file and collect the result:

   ```
   $ fio mixed.fio
   ```

2. For the Docker test, you can prepare a Docker file as follows:

   ```
   FROM ubuntu
   MAINTAINER nkhare@example.com
   RUN apt-get update
   RUN apt-get -qq install -y fio
   ```

```
ADD mixed.fio /
VOLUME ["/ferrari"]
ENTRYPOINT ["fio"]
```

3. Now, create an image using the following command:

   ```
   $ docker build -t docker_fio_perf .
   ```

4. Start the container as follows to run the benchmark and collect the results:

   ```
   $ docker run --rm -v /ferrari:/ferrari docker_fio_perf
   mixed.fio
   ```

5. While running the preceding test on Fedora/RHEL/CentOS, where SELinux is enabled, you will get the `Permission denied` error. To fix it, re-label the host directory while mounting it inside the container as follows:

   ```
   $ docker run --rm -v /ferrari:/ferrari:z docker_fio_perf
   mixed.fio
   ```

How it works...

FIO will run the workload given in the job file and spit out the results.

There's more...

Once the results are collected, you can do the result comparison. You can even try out different kinds of I/O patterns using the job file and get the desired result.

See also

- Look at the disk benchmark results published in IBM and VMware using FIO in the links referenced earlier in this chapter

Benchmarking network performance

Network is one of the key aspects to consider while deploying the applications in the container environment. To do performance comparison with bare metal, VM and containers, we have to consider different scenarios as follows:

- Bare metal to bare metal
- VM to VM
- Docker container to container with the default networking mode (bridge)
- Docker container to container with host net (`--net=host`)
- Docker container running inside VM with the external world

In any of the preceding cases, we can pick up two endpoints to do the benchmarking. We can use tools such as `nuttcp` (`http://www.nuttcp.net/`) and `netperf` (`http://netperf.org/netperf/`) to measure the network bandwidth and request/response, respectively.

Getting ready

Make sure both the endpoints can reach each other and have the necessary packages/software installed. On Fedora 21, you can install `nuttcp` with the following command:

```
$ yum install -y nuttcp
```

And, get `netperf` from its website.

How to do it…

To measure the network bandwidth using `nuttcp`, perform the following steps:

1. Start the `nuttcp` server on one endpoint:

   ```
   $ nuttcp -S
   ```

2. Measure the transmit throughput (client to server) from the client with the following command:

   ```
   $ nuttcp -t <SERVER_IP>
   ```

3. Measure the receiver throughput on the client (server to client) with the following command:

   ```
   $ nuttcp -r <SERVER_IP>
   ```

4. To run the request/response benchmark using `netperf`, perform the following steps:
5. Start `netserver` on one endpoint:

   ```
   $ netserver
   ```

6. Connect to the server from the other endpoint and run the request/response test:
   - For TCP:

     ```
     $ netperf  -H 172.17.0.6 -t TCP_RR
     ```

   - For UDP:

     ```
     $ netperf  -H 172.17.0.6 -t UDP_RR
     ```

How it works…

In both the cases mentioned earlier, one endpoint becomes the client and sends the requests to the server on the other endpoint.

There's more...

We can collect the benchmark results for different scenarios and compare them. `netperf` can also be used for throughput tests.

See also

- Look at the network benchmark results published by IBM and VMware in the links referenced earlier in this chapter

Getting container resource usage using the stats feature

With the release of version 1.5, Docker added a feature to get container resource usage from in-built commands.

Getting ready

A Docker host with version 1.5 or later installed, which can be accessed via the Docker client. Also, start a few containers to get stats.

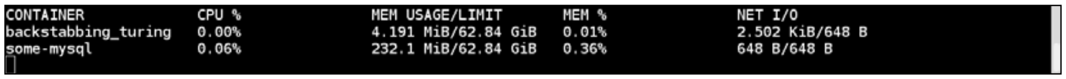
How to do it...

1. Run the following command to get stats from one or more containers:

   ```
   $ docker stats [CONTAINERS]
   ```

   For example, if we have two containers with the names `some-mysql` and `backstabbing_turing`, then run the following command to get the stats:

   ```
   $ docker stats some-mysql backstabbing_turing
   ```



```
CONTAINER             CPU %               MEM USAGE/LIMIT       MEM %               NET I/O
backstabbing_turing   0.00%               4.191 MiB/62.84 GiB   0.01%               2.502 KiB/648 B
some-mysql            0.06%               232.1 MiB/62.84 GiB   0.36%               648 B/648 B
```

How it works...

The Docker daemon fetches the resource information from the Cgroups and serves it through the APIs.

See also

- Refer to the release notes of Docker 1.5 at `https://docs.docker.com/v1.5/release-notes/`

Setting up performance monitoring

We have tools such as SNMP, Nagios, and so on to monitor bare metal and VM performance. Similarly, there are a few tools/plugins available to monitor container performance such as cAdvisor (`https://github.com/google/cadvisor`) and sFlow (`http://blog.sflow.com/2014/06/docker-performance-monitoring.html`). In this recipe, let's see how we can configure cAdvisor.

Getting ready

Setting up cAdvisor.

- The easiest way to run cAdvisor is to run its Docker container, which can be done with the following command:

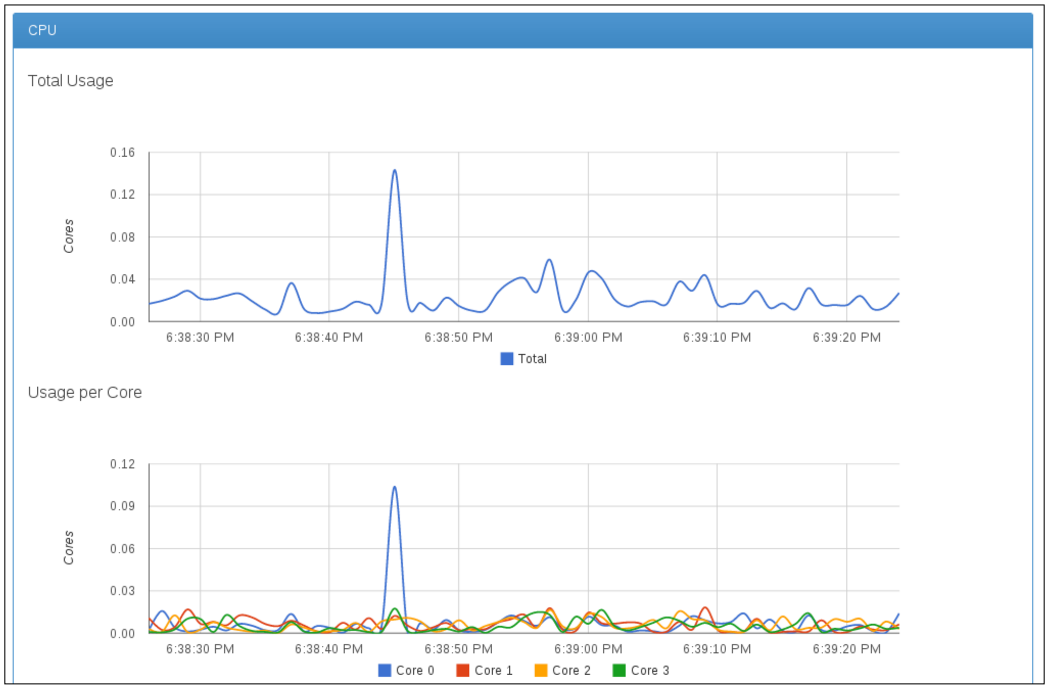
```
sudo docker run \
  --volume=/:/rootfs:ro \
  --volume=/var/run:/var/run:rw \
  --volume=/sys:/sys:ro \
  --volume=/var/lib/docker/:/var/lib/docker:ro \
  --publish=8080:8080 \
  --detach=true \
  --name=cadvisor \
  google/cadvisor:latest
```

- If you want to run cAdvisor outside Docker, then follow the instructions given on the cAdvisor home page at `https://github.com/google/cadvisor/blob/master/docs/running.md#standalone`

How to do it...

After the container starts, point your browser to `http://localhost:8080`. You will first get the graphs for CPU, memory usage and other information for the host machine. Then, by clicking on the Docker Containers link, you will get the URLs for the containers running on the machine under the **Subcontainers** section. If you click on any one of them, you will see the resource usage information for the corresponding container.

The following is the screenshot of one such container:



How it works...

With the `docker run` command, we have mounted few volumes from host machines in read-only mode. cAdvisor will read the relevant information from those like the Cgroup details for containers and show them graphically.

There's more...

cAdvisor supports exporting the performance matrices to influxdb (`http://influxdb.com/`). Heapster (`https://github.com/GoogleCloudPlatform/heapster`) is another project from Google, which allows cluster-wide (Kubernetes) monitoring using cAdvisor.

See also

- You can look at the matrices used by cAdvisor from Cgroups in the documentation on the Docker website `https://docs.docker.com/articles/runmetrics/`

8
Docker Orchestration and Hosting Platforms

In this chapter, we will cover the following recipes:

- Running applications with Docker Compose
- Setting up Cluster with Docker Swarm
- Setting up CoreOS for Docker orchestration
- Setting up a Project Atomic host
- Doing atomic update/rollback with Project Atomic
- Adding more storage for Docker in Project Atomic
- Setting up Cockpit for Project Atomic
- Setting up a Kubernetes cluster
- Scaling up and down in a Kubernetes cluster
- Setting up WordPress with a Kubernetes cluster

Introduction

Running Docker on a single host may be good for the development environment, but the real value comes when we span multiple hosts. However, this is not an easy task. You have to orchestrate these containers. So, in this chapter, we'll cover some of the orchestration tools and hosting platforms.

Docker Inc. announced two such tools:

Docker Compose (`https://docs.docker.com/compose`) to create apps consisting of multiple containers and Docker Swarm (`https://docs.docker.com/swarm/`) to cluster multiple Docker hosts. Docker Compose was previously called Fig (`http://www.fig.sh/`).

CoreOS (`https://coreos.com/`) created etcd (`https://github.com/coreos/etcd`) for consensus and service discovery, fleet (`https://coreos.com/using-coreos/clustering`) to deploy containers in a cluster, and flannel (`https://github.com/coreos/flannel`) for overlay networking.

Google started Kubernetes (`http://kubernetes.io/`) for Docker orchestration. Kubernetes provides mechanisms for application deployment, scheduling, updating, maintenance, and scaling.

Red Hat launched a container-specific operating system called Project Atomic (`http://www.projectatomic.io/`), which can leverage the orchestration capabilities of Kubernetes.

Even Microsoft announced a specialized operating system for Docker (`http://azure.microsoft.com/blog/2015/04/08/microsoft-unveils-new-container-technologies-for-the-next-generation-cloud/`).

Apache Mesos (`http://mesos.apache.org/`), which provides resource management and scheduling across entire datacenter and cloud environments, also added support for Docker (`http://mesos.apache.org/documentation/latest/docker-containerizer/`).

VMware also launched the container-specific host VMware Photon (`http://vmware.github.io/photon/`).

This is definitely a very interesting space, but the policy management tools of many orchestration engines do not make the lives of developers and operators easy. They have to learn different tools and formats when they move from one platform to another. It would be great if we could have a standard way to build and launch composite, multicontainer apps. The Project Atomic community seems to be working on one such platform-neutral specification called Nulecule (`https://github.com/projectatomic/nulecule/`). A good description about Nulecule is available at `http://www.projectatomic.io/blog/2015/05/announcing-the-nulecule-specification-for-composite-applications/`:

> *"Nulecule defines a pattern and model for packaging complex multi-container applications, referencing all their dependencies, including orchestration metadata, in a single container image for building, deploying, monitoring, and active management. Just create a container with a Nulecule file and the app will 'just work'. In the Nulecule spec, you define orchestration providers, container locations and configuration parameters in a graph, and the Atomic App implementation will piece them together for you with the help of Providers. The Nulecule specification supports aggregation of multiple composite applications, and it's also container and orchestration agnostic, enabling the use of any container and orchestration engine."*

AtomicApp is a reference implementation (`https://github.com/projectatomic/atomicapp/`) of the Nulecule specification. It can be used to bootstrap container applications and to install and run them. AtomicApp currently has a limited number of providers (Docker, Kubernetes, OpenShift), but support for others will be added soon.

On a related note, the CentOS community is building a CI environment, which will take advantage of Nulecule and AtomicApp. For further information, visit `http://wiki.centos.org/ContainerPipeline`.

All of the preceding tools and platforms need separate chapters for themselves. In this chapter, we'll explore Compose, Swarm, CoreOS, Project Atomic, and Kubernetes briefly.

Running applications with Docker Compose

Docker Compose (`http://docs.docker.com/compose/`) is the native Docker tool to run the interdependent containers that make up an application. We define a multicontainer application in a single file and feed it to Docker Compose, which sets up the application. At the time of writing, Compose is still not production-ready. In this recipe, we'll once again use WordPress as a sample application to run.

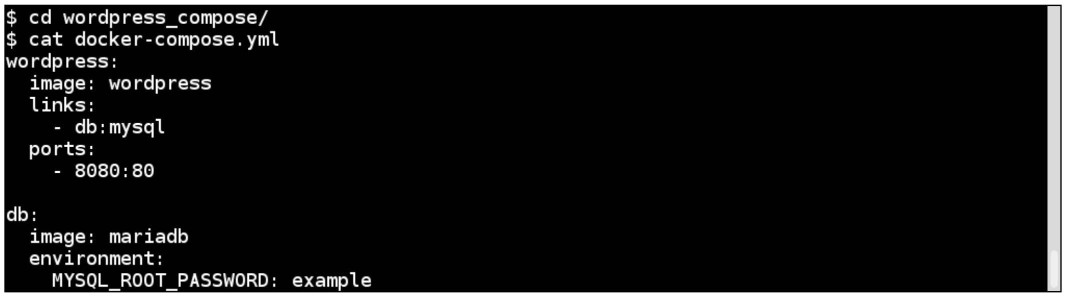
Getting ready

Make sure you have Docker Version 1.3 or later installed on the system. To install Docker Compose, run the following command:

```
$ sudo pip install docker-compose
```

How to do it...

1. Create a directory for the application, and within it create `docker-compose.yml` to define the app:



```
$ cd wordpress_compose/
$ cat docker-compose.yml
wordpress:
  image: wordpress
  links:
    - db:mysql
  ports:
    - 8080:80

db:
  image: mariadb
  environment:
    MYSQL_ROOT_PASSWORD: example
```

2. We have taken the preceding example from the official WordPress Docker repo on Docker Hub (`https://registry.hub.docker.com/_/wordpress/`).

3. Within the app directory, run the following command to build the app:

   ```
   $ docker-compose up
   ```

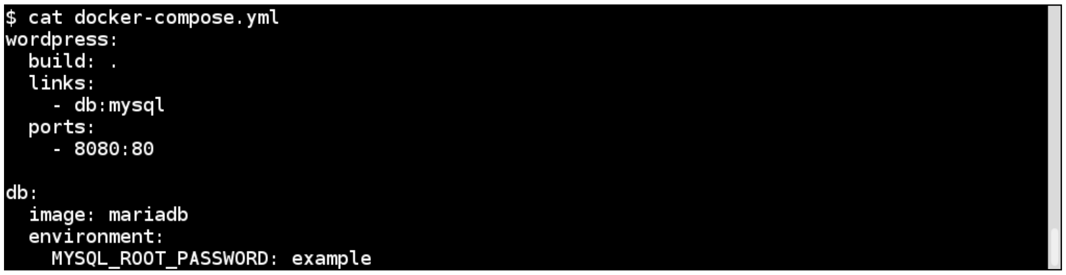
4. Once the build is complete, access the WordPress installation page from `http://localhost:8080` or `http://<host-ip>:8080`.

How it works...

Docker Compose downloads both the `mariadb` `wordpress` images, if not available locally from the official Docker registry. First, it starts the `db` container from the `mariadb` image; then it starts the `wordpress` container. Next, it links with the `db` container and exports the port to the host machine.

There's more...

We can even build images from the Dockerfile during the compose and then use it for the app. For example, to build the `wordpress` image, we can get the corresponding Dockerfile and other supporting file from within the application's Compose directory and update the `docker-compose.yml` file in a similar manner as follows:



```
$ cat docker-compose.yml
wordpress:
  build: .
  links:
    - db:mysql
  ports:
    - 8080:80

db:
  image: mariadb
  environment:
    MYSQL_ROOT_PASSWORD: example
```

We can start, stop, rebuild, and get the status of the app. Visit its documentation on the Docker website.

See also

- The Docker Compose YAML file reference at `http://docs.docker.com/compose/yml/`
- The Docker Compose command-line reference at `http://docs.docker.com/compose/cli/`
- The Docker Compose GitHub repository at `https://github.com/docker/compose`

Setting up cluster with Docker Swarm

Docker Swarm (`http://docs.docker.com/swarm/`) is native clustering to Docker. It groups multiple Docker hosts into a single pool in which one can launch containers. In this recipe, we'll use Docker Machine (`http://docs.docker.com/machine/`) to set up a Swarm cluster. At the time of writing, Swarm is still not production-ready. If you recall, we used Docker Machine to set up a Docker host on Google Compute Engine in *Chapter 1*, *Introduction and Installation*. To keep things simple, here we'll use VirtualBox as the backend for Docker Machine to configure hosts.

Getting ready

1. Install VirtualBox on your system (`https://www.virtualbox.org/`). Instructions to configure VirtualBox are outside the scope of this book.
2. Download and set up Docker Machine. In Fedora x86_64, run the following commands:

```
$ wget
https://github.com/docker/machine/releases/download/v0.2.0/doc
ker-machine_linux-amd64
$ sudo mv  docker-machine_linux-amd64 /usr/local/bin/docker-
machine
$ chmod a+x  /usr/local/bin/docker-machine
```

How to do it…

1. Using the Swarm discovery service, we first need to create a Swarm token to identify our cluster uniquely. Other than the default hosted discovery service, Swarm supports different types of discovery services such as etcd, consul, and zookeeper. For more details, please visit `https://docs.docker.com/swarm/discovery/`. To create a token using the default hosted discovery service, we'll first set up the Docker host using Docker Machine on a VM and then get the token:

```
$ docker-machine create -d virtualbox local
```

2. To access the Docker we just created from your local Docker client, run the following command:

```
$ eval "$(docker-machine env local)"
```

3. To get the token, run the following command:

```
$ docker run swarm create
7c3a21b42708cde81d99884116d68fa1
```

4. Using the token created in the preceding step, set up Swarm master:

   ```
   $ docker-machine create  -d virtualbox  --swarm  --swarm-
   master  --swarm-discovery
   token://7c3a21b42708cde81d99884116d68fa1  swarm-master
   ```

5. Similarly, let's create two Swarm nodes:

   ```
   $ docker-machine create -d virtualbox  --swarm  --swarm-
   discovery token://7c3a21b42708cde81d99884116d68fa1 swarm-node-
   1
   ```

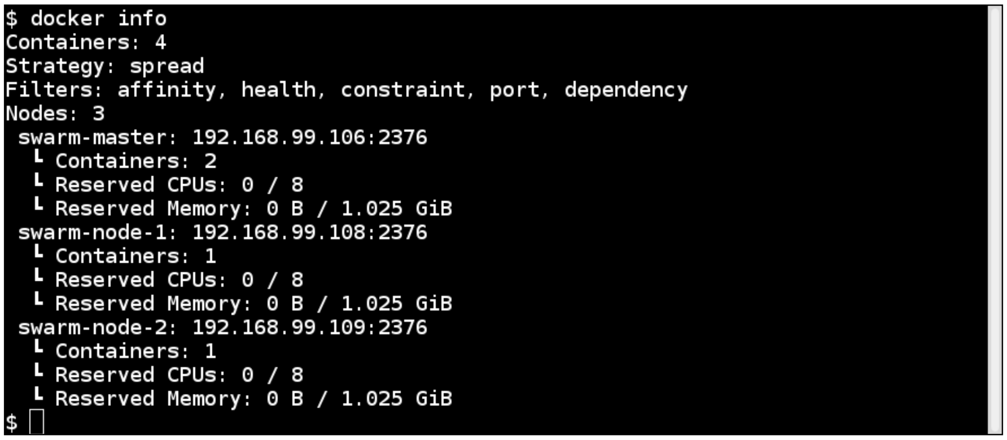
   ```
   $ docker-machine create -d virtualbox  --swarm  --swarm-
   discovery token://7c3a21b42708cde81d99884116d68fa1 swarm-node-
   2
   ```

6. Now, connect to Docker Swarm from your local Docker client:

   ```
   $ eval "$(docker-machine env swarm-master)"
   ```

7. Swarm APIs are compatible with Docker client APIs. Let's run the `docker info` command to see Swarm's current configuration/setup:

   ```
   $ docker info
   ```



```
$ docker info
Containers: 4
Strategy: spread
Filters: affinity, health, constraint, port, dependency
Nodes: 3
 swarm-master: 192.168.99.106:2376
  └ Containers: 2
  └ Reserved CPUs: 0 / 8
  └ Reserved Memory: 0 B / 1.025 GiB
 swarm-node-1: 192.168.99.108:2376
  └ Containers: 1
  └ Reserved CPUs: 0 / 8
  └ Reserved Memory: 0 B / 1.025 GiB
 swarm-node-2: 192.168.99.109:2376
  └ Containers: 1
  └ Reserved CPUs: 0 / 8
  └ Reserved Memory: 0 B / 1.025 GiB
$ 
```

As you can see, we have three nodes in the cluster: one master and two nodes.

How it works...

Using the unique token we got from the hosted discovery service, we registered the master and nodes in a cluster.

There's more...

- In the preceding `docker info` output, we also scheduled policy (strategy) and filters. More information on these can be found at `https://docs.docker.com/swarm/scheduler/strategy/` and `https://docs.docker.com/swarm/scheduler/filter/`. These define where the container will run.
- There is active development happening to integrate Docker Swarm and Docker Compose so that we point and compose the app to the Swarm cluster. The app will then start on the cluster. Visit `https://github.com/docker/compose/blob/master/SWARM.md`

See also

- The Swarm documentation on the Docker website at `https://docs.docker.com/swarm/`
- Swarm's GitHub repository at `https://github.com/docker/swarm`

Setting up CoreOS for Docker orchestration

CoreOS (`https://coreos.com/`) is a Linux distribution that has been rearchitected to provide the features needed to run modern infrastructure stacks. It is Apache 2.0 Licensed. It has a product called CoreOS Managed Linux (`https://coreos.com/products/managed-linux/`) for which the CoreOS team provides commercial support.

Essentially, CoreOS provides platforms to host a complete applications stack. We can set up CoreOS on different cloud providers, bare metal, and in the VM environment. Let's look at the building blocks of CoreOS:

- etcd
- Container runtime
- Systemd
- Fleet

Let's discuss each in detail:

- **etcd**: From the GitHub page of etcd (`https://github.com/coreos/etcd/#etcd`). `etcd` is a highly available key-value store for shared configuration and service discovery. It is inspired by Apache ZooKeeper and doozer with a focus on being:
  - **Simple**: Curl-able user-facing API (HTTP plus JSON)
  - **Secure**: Optional SSL client certificate authentication

- **Fast**: Benchmark of 1,000s of writes per instance
- **Reliable**: Proper distribution using Raft

It is written in Go and uses the Raft consensus algorithm (`https://raftconsensus.github.io/`) to manage a highly available replicated log. etcd can be used independent of CoreOS. We can:

- Set up a single or multinode cluster. More information on this can be found at `https://github.com/coreos/etcd/blob/master/Documentation/clustering.md`.
- Access using CURL and different libraries, found at `https://github.com/coreos/etcd/blob/master/Documentation/libraries-and-tools.md`.

In CoreOS, `etcd` is meant for the coordination of clusters. It provides a mechanism to store configurations and information about services in a consistent way.

- **Container runtime**: CoreOS supports Docker as a container runtime environment. In December 2014, CoreOS announced a new container runtime Rocket (`https://coreos.com/blog/rocket/`). Let's restrict our discussion to Docker, which is currently installed on all CoreOS machines.
- **systemd**: `systemd` is an init system used to start, stop, and manage processes. In CoreOS, it is used to:
  - Launch Docker containers
  - Register services launched by containers to etcd

Systemd manages unit files. A sample unit file looks like the following:

```
[Unit]
Description=Docker Application Container Engine
Documentation=http://docs.docker.com
After=network.target docker.socket
Requires=docker.socket

[Service]
Type=notify
EnvironmentFile=-/etc/sysconfig/docker
EnvironmentFile=-/etc/sysconfig/docker-storage
ExecStart=/usr/bin/docker -d -H fd:// $OPTIONS
$DOCKER_STORAGE_OPTIONS
LimitNOFILE=1048576
LimitNPROC=1048576

[Install]
WantedBy=multi-user.target
```

This unit file starts the Docker daemon with the command mentioned in `ExecStart` on Fedora 21. The Docker daemon will start after the `network target` and `docker socket` services. `docker socket` is a prerequisite for the Docker daemon to start. Systemd targets are ways to group processes so that they can start at the same time. `multi-user` is one of the targets with which the preceding unit file is registered. For more details, you can look at the upstream documentation of Systemd at `http://www.freedesktop.org/wiki/Software/systemd/`.

- **Fleet**: Fleet (`https://coreos.com/using-coreos/clustering/`) is the cluster manager that controls `systemd` at the cluster level. systemd unit files are combined with some Fleet-specific properties to achieve the goal. From the Fleet documentation (`https://github.com/coreos/fleet/blob/master/Documentation/architecture.md`):

  > *"Every system in the fleet cluster runs a single `fleetd` daemon. Each daemon encapsulates two roles: the engine and the agent. An engine primarily makes scheduling decisions while an agent executes units. Both the engine and agent use the reconciliation model, periodically generating a snapshot of 'current state' and 'desired state' and doing the necessary work to mutate the former towards the latter."*

  `etcd` is the sole datastore in a `fleet` cluster. All persistent and ephemeral data is stored in `etcd`; unit files, cluster presence, unit state, and so on. `etcd` is also used for all internal communication between fleet engines and agents.

Now we know of all the building blocks of CoreOS. Let's try out CoreOS on our local system/laptop. To keep things simple, we will use Vagrant to set up the environment.

Getting ready

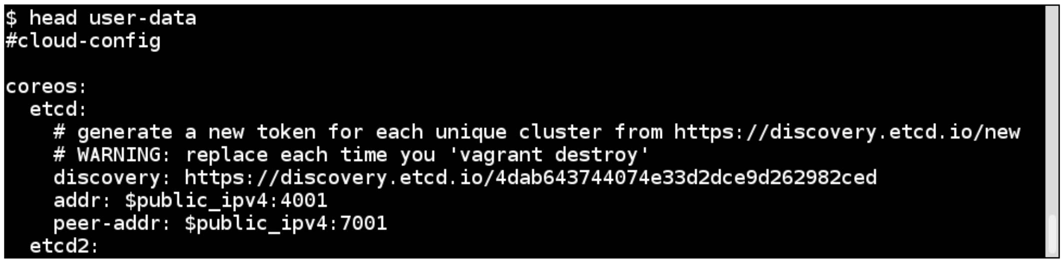
1. Install VirtualBox on the system (`https://www.virtualbox.org/`) and Vagrant (`https://www.vagrantup.com/`). The instructions to configure both of these things are outside the scope of this book.
2. Clone the `coreos-vagrant` repository:

   ```
   $ git clone https://github.com/coreos/coreos-vagrant.git
   $ cd coreos-vagrant
   ```

3. Copy the sample file `user-data.sample` to `user-data` and set up the token to bootstrap the cluster:

   ```
   $ cp user-data.sample user-data
   ```

4. When we configure the CoreOS cluster with more than one node, we need a token to bootstrap the cluster to select the initial etcd leader. This service is provided free by the CoreOS team. We just need to open `https://discovery.etcd.io/new` in the browser to get the token and update it within the `user-data` file as follows:



```
$ head user-data
#cloud-config

coreos:
  etcd:
    # generate a new token for each unique cluster from https://discovery.etcd.io/new
    # WARNING: replace each time you 'vagrant destroy'
    discovery: https://discovery.etcd.io/4dab643744074e33d2dce9d262982ced
    addr: $public_ipv4:4001
    peer-addr: $public_ipv4:7001
  etcd2:
```

5. Copy `config.rb.sample` to `config.rb` and make changes to the following line:

   `$num_instances=1`

   It should now look like this:

   `$num_instances=3`

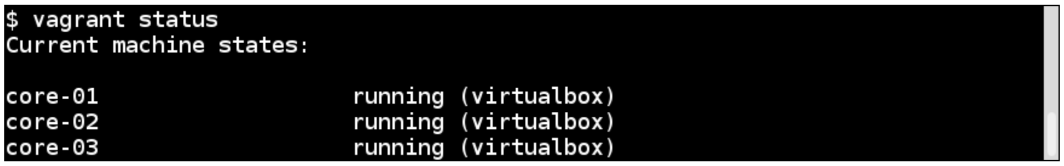
This will ask Vagrant to set up three node clusters. By default, Vagrant is configured to get the VM images from the alpha release. We can change it to beta or stable by updating the `$update_channel` parameter in Vagrantfile. For this recipe, I chose stable.

How to do it...

1. Run the following command to set up the cluster:

   `$ vagrant up`

   Now, check the status, using the command shown in the following screenshot:



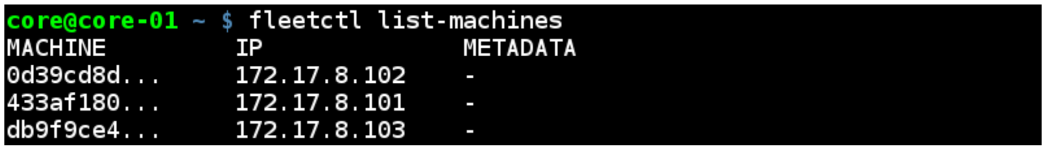
```
$ vagrant status
Current machine states:

core-01                   running (virtualbox)
core-02                   running (virtualbox)
core-03                   running (virtualbox)
```

2. Log in to one of the VMs using SSH, look at the status of services, and list the machines in the cluster:

   `$ vagrant ssh core-01`

   `$ systemctl status etcd fleet`

   `$ fleetctl list-machines`

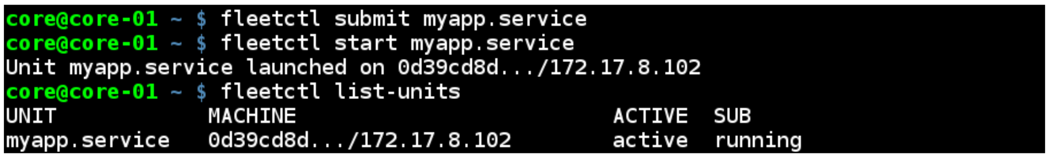

```
core@core-01 ~ $ fleetctl list-machines
MACHINE         IP              METADATA
0d39cd8d...     172.17.8.102    -
433af180...     172.17.8.101    -
db9f9ce4...     172.17.8.103    -
```

3. Create a service unit file called `myapp.service` with the following content:

```
[Unit]
Description=MyApp
After=docker.service
Requires=docker.service

[Service]
TimeoutStartSec=0
ExecStartPre=-/usr/bin/docker kill busybox1
ExecStartPre=-/usr/bin/docker rm busybox1
ExecStartPre=/usr/bin/docker pull busybox
ExecStart=/usr/bin/docker run --name busybox1 busybox /bin/sh -c
"while true; do echo Hello World; sleep 1; done"
ExecStop=/usr/bin/docker stop busybox1
```

4. Let's now submit the service for scheduling and start the service:

```
$ fleetctl submit myapp.service
$ fleetctl start myapp.service
$ fleetctl list-units
```


```
core@core-01 ~ $ fleetctl submit myapp.service
core@core-01 ~ $ fleetctl start myapp.service
Unit myapp.service launched on 0d39cd8d.../172.17.8.102
core@core-01 ~ $ fleetctl list-units
UNIT            MACHINE                         ACTIVE  SUB
myapp.service   0d39cd8d.../172.17.8.102        active  running
```

As we can see, our service has started on one of the nodes in the cluster.

How it works...

Vagrant uses the cloud configuration file (`user-data`) to boot the VMs. As they have the same token to bootstrap the cluster, they select the leader and start operating. Then, with `fleetctl`, which is the fleet cluster management tool, we submit the unit file for scheduling, which starts on one of the nodes.

There's more...

- Using the cloud configuration file in this recipe, we can start `etcd` and `fleet` on all the VMs. We can choose to run `etcd` just on selected nodes and then configure worker nodes running `fleet` to connect to etcd servers. This can be done by setting the cloud configuration file accordingly. For more information, please visit `https://coreos.com/docs/cluster-management/setup/cluster-architectures/`.
- With `fleet`, we can configure services for high availability. For more information, take a look at `https://coreos.com/docs/launching-containers/launching/fleet-unit-files/`.
- Though your service is running on the host, you will not be able to reach it from the outside world. You will need to add some kind of router and wildcard DNS configuration to reach your service from the outside world.

See also

- The CoreOS documentation for more details at `https://coreos.com/docs/`
- The visualization of RAFT consensus algorithm at `http://thesecretlivesofdata.com/raft`
- How to configure the cloud config file at `https://coreos.com/docs/cluster-management/setup/cloudinit-cloud-config/` and `https://coreos.com/validate/`
- Documentation on systemd at `https://coreos.com/docs/launching-containers/launching/getting-started-with-systemd/`
- How to launch containers with fleet at `https://coreos.com/docs/launching-containers/launching/launching-containers-fleet/`

Setting up a Project Atomic host

Project Atomic facilitates application-centric IT architecture by providing an end-to-end solution to deploy containerized applications quickly and reliably, with atomic update and rollback for the application and host alike.

This is achieved by running applications in containers on a Project Atomic host, which is a lightweight operating system specially designed to run containers. The hosts can be based on Fedora, CentOS, or Red Hat Enterprise Linux.

Next, we will elaborate on the building blocks of the Project Atomic host.

- **OSTree and rpm-OSTree**: OSTree (`https://wiki.gnome.org/action/show/Projects/OSTree`) is a tool to manage bootable, immutable, and versioned filesystem trees. Using this, we can build client-server architecture in which the server hosts an OSTree repository and the client subscribed to it can incrementally replicate the content.

  rpm-OSTree is a system to decompose RPMs on the server side into the OSTree repository to which the client can subscribe and perform updates. With each update, a new root is created, which is used for the next reboot. During updates, `/etc` is rebased and `/var` is untouched.

- **Container runtime**: As of now Project Atomic only supports Docker as container runtime.
- **systemd**: As we saw in earlier recipes, systemd is a new init system. It also helps to set up SELinux policies to containers for complete multitenant security and to control Cgroups policies, which we looked in at *Chapter 1*, *Introduction and Installation*.

  Project Atomic uses Kubernetes (`http://kubernetes.io/`) for application deployment over clusters of container hosts. Project Atomic can be installed on bare metal, cloud providers, VMs, and so on. In this recipe, let's see how we can install it on a VM using virt-manager on Fedora.

Getting ready

1. Download the image:

   ```
   $ wget
   http://download.fedoraproject.org/pub/fedora/linux/releases/te
   st/22_Beta/Cloud/x86_64/Images/Fedora-Cloud-Atomic-22_Beta-
   20150415.x86_64.raw.xz
   ```

   I have downloaded the beta image for Fedora 22 Cloud image *For Containers*. You should look for the latest cloud image *For Containers* at `https://getfedora.org/en/cloud/download/`.

2. Uncompress this image by using the following command:

   ```
   $ xz -d Fedora-Cloud-Atomic-22_Beta-20150415.x86_64.raw.xz
   ```

How to do it...

1. We downloaded the cloud image that does not have any password set for the default user `fedora`. While booting the VM, we have to provide a cloud configuration file through which we can customize the VM. To do this, we need to create two files, `meta-data` and `user-data`, as follows:

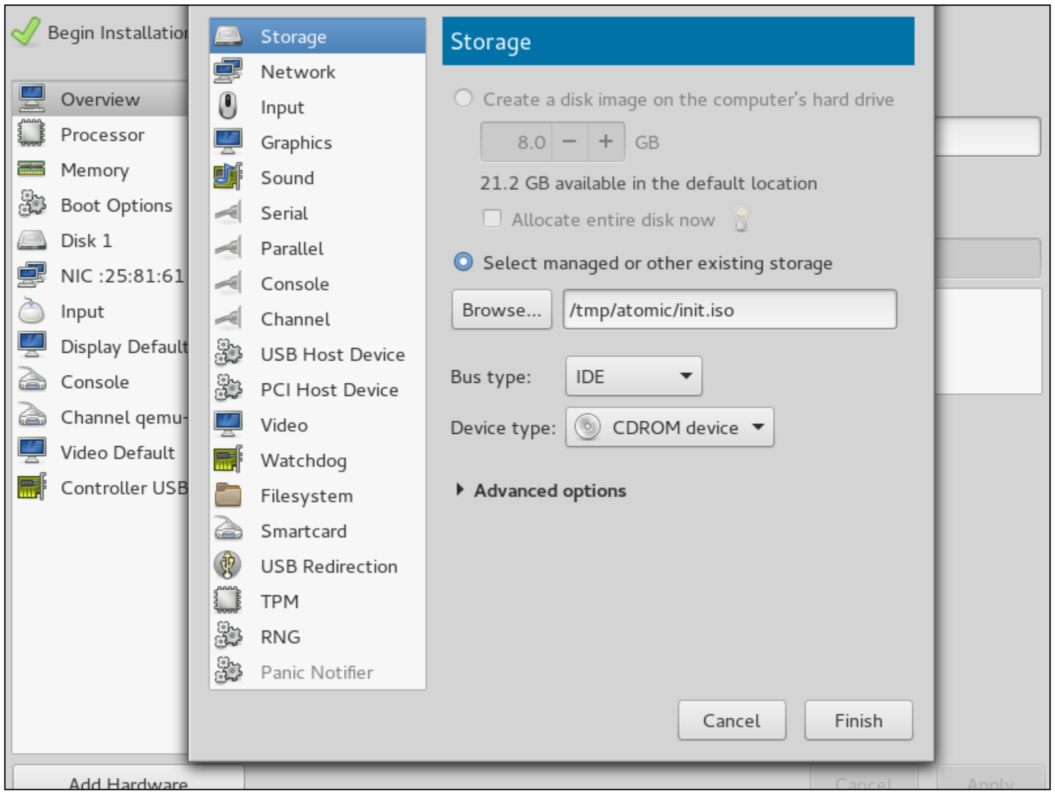
   ```
   $ cat  meta-data
   instance-id: iid-local01
   local-hostname: atomichost

   $ cat user-data
   #cloud-config
   password: atomic
   ssh_pwauth: True
   chpasswd: { expire: False }

   ssh_authorized_keys:
   - ssh-rsa AAAAB3NzaC1yc.........
   ```

   In the preceding code, we need to provide the complete SSH public key. We then need to create an ISO image consisting of these files, which we will use to boot to the VM. As we are using a cloud image, our setting will be applied to the VM during the boot process. This means the hostname will be set to `atomichost`, the password will be set to `atomic`, and so on. To create the ISO, run the following command:

   ```
   $ genisoimage -output init.iso -volid cidata -joliet -rock
   user-data meta-data
   ```

2. Start virt-manager.
3. Select **New Virtual Machine** and then import the existing disk image. Enter the image path of the Project Atomic image we downloaded earlier. Select **OS type** as **Linux** and **Version** as **Fedora 20/Fedora 21 (or later)**, and click on **Forward**. Next, assign CPU and Memory and click on **Forward**. Then, give a name to the VM and select **Customize configuration** before install. Finally, click on **Finish** and review the details.

4. Next, click on **Add Hardware**, and after selecting **Storage**, attach the ISO (`init.iso`) file we created to the VM and select **Begin Installation**:



Storage

Create a disk image on the computer's hard drive

8.0 GB

21.2 GB available in the default location

Allocate entire disk now

Select managed or other existing storage

Browse... /tmp/atomic/init.iso

Bus type: IDE

Device type: CDROM device

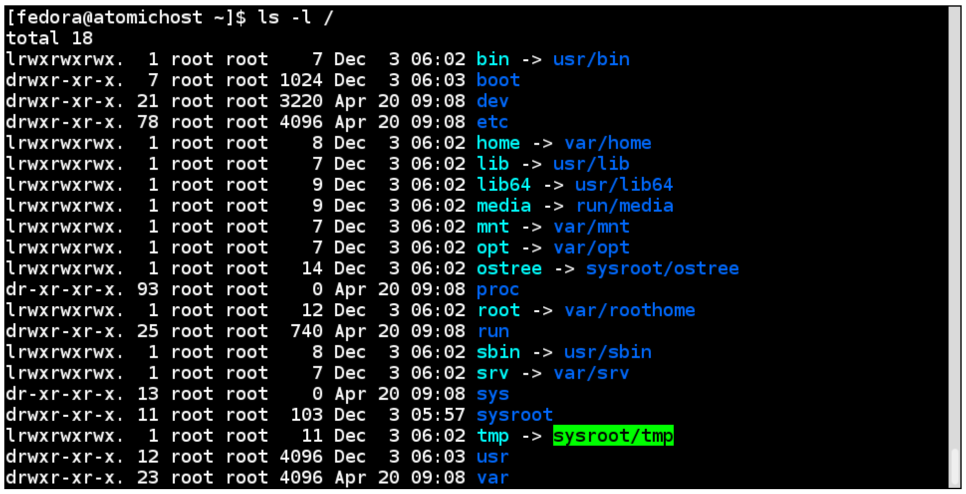
Once booted, you can see that its hostname is correctly set and you will be able to log in through the password given in the cloud init file. The default user is `fedora` and password is `atomic` as set in the `user-data` file.

How it works...

In this recipe, we took a Project Atomic Fedora cloud image and booted it using `virt-manager` after supplying the cloud init file.

There's more...

- After logging in, if you do file listing at `/`, you will see that most of the traditional directories are linked to `/var` because it is preserved across upgrades.



```
[fedora@atomichost ~]$ ls -l /
total 18
lrwxrwxrwx.  1 root root    7 Dec  3 06:02 bin -> usr/bin
drwxr-xr-x.  7 root root 1024 Dec  3 06:03 boot
drwxr-xr-x. 21 root root 3220 Apr 20 09:08 dev
drwxr-xr-x. 78 root root 4096 Apr 20 09:08 etc
lrwxrwxrwx.  1 root root    8 Dec  3 06:02 home -> var/home
lrwxrwxrwx.  1 root root    7 Dec  3 06:02 lib -> usr/lib
lrwxrwxrwx.  1 root root    9 Dec  3 06:02 lib64 -> usr/lib64
lrwxrwxrwx.  1 root root    9 Dec  3 06:02 media -> run/media
lrwxrwxrwx.  1 root root    7 Dec  3 06:02 mnt -> var/mnt
lrwxrwxrwx.  1 root root    7 Dec  3 06:02 opt -> var/opt
lrwxrwxrwx.  1 root root   14 Dec  3 06:02 ostree -> sysroot/ostree
dr-xr-xr-x. 93 root root    0 Apr 20 09:08 proc
lrwxrwxrwx.  1 root root   12 Dec  3 06:02 root -> var/roothome
drwxr-xr-x. 25 root root  740 Apr 20 09:08 run
lrwxrwxrwx.  1 root root    8 Dec  3 06:02 sbin -> usr/sbin
lrwxrwxrwx.  1 root root    7 Dec  3 06:02 srv -> var/srv
dr-xr-xr-x. 13 root root    0 Apr 20 09:08 sys
drwxr-xr-x. 11 root root  103 Dec  3 05:57 sysroot
lrwxrwxrwx.  1 root root   11 Dec  3 06:02 tmp -> sysroot/tmp
drwxr-xr-x. 12 root root 4096 Dec  3 06:03 usr
drwxr-xr-x. 23 root root 4096 Apr 20 09:08 var
```

- After logging in, you can run the Docker command as usual:

  ```
  $sudo docker run -it fedora bash
  ```

See also

- The virtual manager documentation at `https://virt-manager.org/documentation/`
- More information on package systems, image systems, and RPM-OSTree at `https://github.com/projectatomic/rpm-ostree/blob/master/doc/background.md`
- The quick-start guide on the Project Atomic website at `http://www.projectatomic.io/docs/quickstart/`
- The resources on cloud images at `https://www.technovelty.org//linux/running-cloud-images-locally.html` and `http://cloudinit.readthedocs.org/en/latest/`
- How to set up Kubernetes with an Atomic host at `http://www.projectatomic.io/blog/2014/11/testing-kubernetes-with-an-atomic-host/` and `https://github.com/cgwalters/vagrant-atomic-cluster`

Doing atomic update/rollback with Project Atomic

To get to the latest version or to roll back to the older version of Project Atomic, we use the `atomic host` command, which internally calls rpm-ostree.

Getting ready

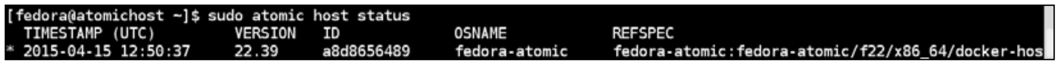
Boot and log in to the Atomic host.

How to do it...

1. Just after the boot, run the following command:

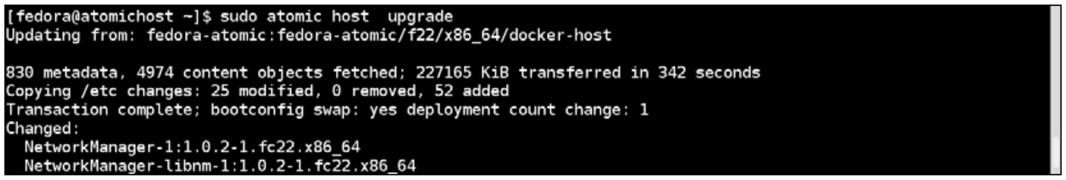
   `$ atomic host status`

   You will see details about one deployment that is in use now.



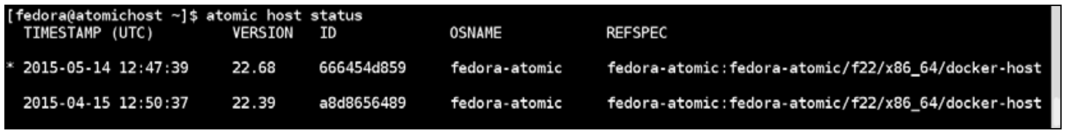
```
[fedora@atomichost ~]$ sudo atomic host status
  TIMESTAMP (UTC)         VERSION   ID            OSNAME           REFSPEC
* 2015-04-15 12:50:37     22.39     a8d8656489    fedora-atomic    fedora-atomic:fedora-atomic/f22/x86_64/docker-hos
```

   To upgrade, run the following command:



```
[fedora@atomichost ~]$ sudo atomic host  upgrade
Updating from: fedora-atomic:fedora-atomic/f22/x86_64/docker-host

830 metadata, 4974 content objects fetched; 227165 KiB transferred in 342 seconds
Copying /etc changes: 25 modified, 0 removed, 52 added
Transaction complete; bootconfig swap: yes deployment count change: 1
Changed:
  NetworkManager-1:1.0.2-1.fc22.x86_64
  NetworkManager-libnm-1:1.0.2-1.fc22.x86_64
```

2. This changes and/or adds new packages. After the upgrade, we will need to reboot the system to use the new update. Let's reboot and see the outcome:



```
[fedora@atomichost ~]$ atomic host status
  TIMESTAMP (UTC)         VERSION   ID            OSNAME           REFSPEC

* 2015-05-14 12:47:39     22.68     666454d859    fedora-atomic    fedora-atomic:fedora-atomic/f22/x86_64/docker-host

  2015-04-15 12:50:37     22.39     a8d8656489    fedora-atomic    fedora-atomic:fedora-atomic/f22/x86_64/docker-host
```

   As we can see, the system is now booted with the new update. The `*`, which is at the beginning of the first line, specifies the active build.

3. To roll back, run the following command:

   ```
   $ sudo atomic host rollback
   ```

   We will have to reboot again if we want to use older bits.

How it works...

For updates, the Atomic host connects to the remote repository hosting the newer build, which is downloaded and used from the next reboot onwards until the user upgrades or rolls back. In the case rollback older build available on the system used after the reboot.

See also

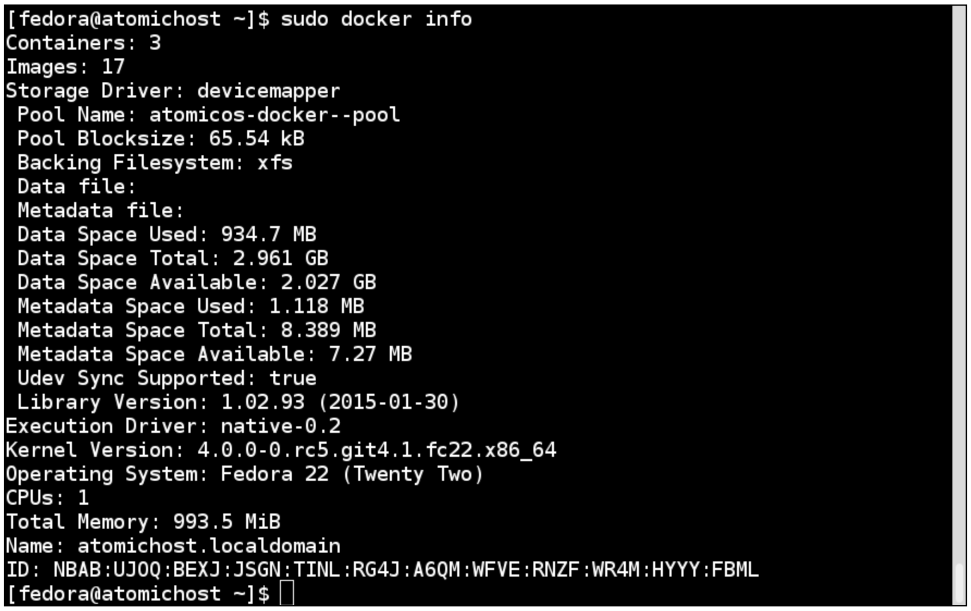
- The documentation Project Atomic website, which can be found at `http://www.projectatomic.io/docs/os-updates/`

Adding more storage for Docker in Project Atomic

The Atomic host is a minimal distribution and, as such, is distributed on a 6 GB image to keep the footprint small. This is very less amount of storage to build and store lots of Docker images, so it is recommended to attach external storage for those operations.

By default, Docker uses `/var/lib/docker` as the default directory where all Docker-related files, including images, are stored. In Project Atomic, we use direct LVM volumes via the devicemapper backend to store Docker images and metadata in `/dev/atomicos/docker-data` and `/dev/atomicos/docker-meta` respectively.

So, to add more storage, Project Atomic provides a helper script called `docker-storage-helper` to add an external disk into the existing LVM thin pool. Let's look at the current available storage to Docker with the `docker info` command:



```
[fedora@atomichost ~]$ sudo docker info
Containers: 3
Images: 17
Storage Driver: devicemapper
 Pool Name: atomicos-docker--pool
 Pool Blocksize: 65.54 kB
 Backing Filesystem: xfs
 Data file:
 Metadata file:
 Data Space Used: 934.7 MB
 Data Space Total: 2.961 GB
 Data Space Available: 2.027 GB
 Metadata Space Used: 1.118 MB
 Metadata Space Total: 8.389 MB
 Metadata Space Available: 7.27 MB
 Udev Sync Supported: true
 Library Version: 1.02.93 (2015-01-30)
Execution Driver: native-0.2
Kernel Version: 4.0.0-0.rc5.git4.1.fc22.x86_64
Operating System: Fedora 22 (Twenty Two)
CPUs: 1
Total Memory: 993.5 MiB
Name: atomichost.localdomain
ID: NBAB:UJOQ:BEXJ:JSGN:TINL:RG4J:A6QM:WFVE:RNZF:WR4M:HYYY:FBML
[fedora@atomichost ~]$
```

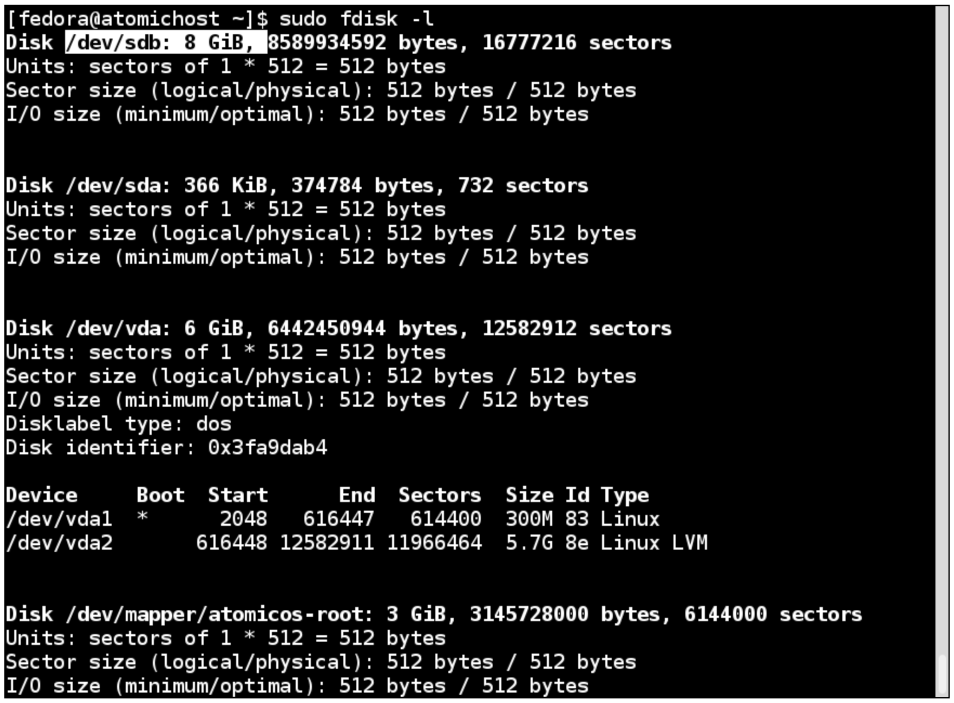
As we can see, the total data space is 2.96 GB and the total metadata space is 8.38 MB.

Getting ready

1. Stop the VM, if it is running.
2. Add an additional disk of the size you want to the Project Atomic VM. I have added 8 GB.
3. Boot the VM.
4. Check whether the newly attached disk is visible to the VM or not.

How to do it...

1. Check if the additional disk is available to the Atomic host VM:



```
[fedora@atomichost ~]$ sudo fdisk -l
Disk /dev/sdb: 8 GiB, 8589934592 bytes, 16777216 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes


Disk /dev/sda: 366 KiB, 374784 bytes, 732 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes


Disk /dev/vda: 6 GiB, 6442450944 bytes, 12582912 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x3fa9dab4

Device     Boot  Start      End  Sectors  Size Id Type
/dev/vda1  *      2048   616447   614400  300M 83 Linux
/dev/vda2       616448 12582911 11966464  5.7G 8e Linux LVM


Disk /dev/mapper/atomicos-root: 3 GiB, 3145728000 bytes, 6144000 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
```

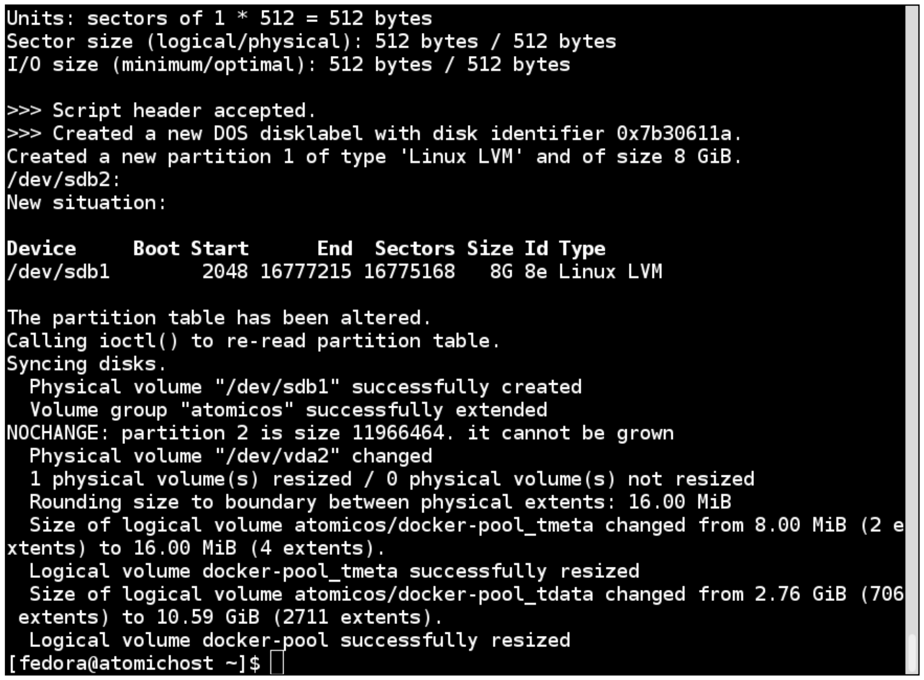
   As we can see, the newly created 8 GB disk is available to the VM.

2. As the newly attached disk is `/dev/sdb`, create a file called `/etc/sysconfig/docker-storage-setup` with the following content:

```
DEVS="/dev/sdb"
[fedora@atomichost ~]$ cat /etc/sysconfig/docker-storage-setup
DEVS="/dev/sdb"
```

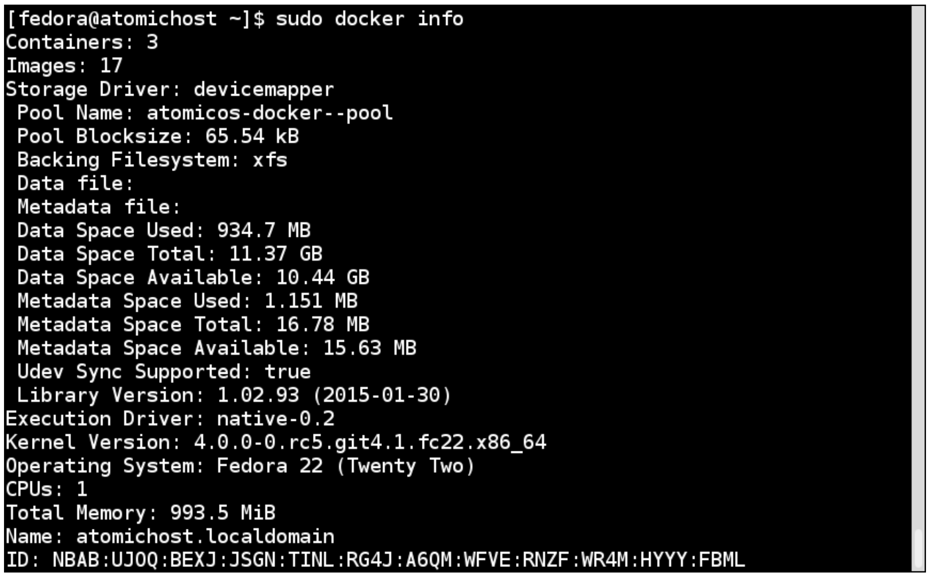
3. Run the `docker-storage-setup` command to add `/dev/sdb` to the existing volume:

```
$ sudo docker-storage-setup
```



```
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes

>>> Script header accepted.
>>> Created a new DOS disklabel with disk identifier 0x7b30611a.
Created a new partition 1 of type 'Linux LVM' and of size 8 GiB.
/dev/sdb2:
New situation:

Device     Boot Start      End  Sectors Size Id Type
/dev/sdb1        2048 16777215 16775168   8G 8e Linux LVM

The partition table has been altered.
Calling ioctl() to re-read partition table.
Syncing disks.
  Physical volume "/dev/sdb1" successfully created
  Volume group "atomicos" successfully extended
NOCHANGE: partition 2 is size 11966464. it cannot be grown
  Physical volume "/dev/vda2" changed
  1 physical volume(s) resized / 0 physical volume(s) not resized
  Rounding size to boundary between physical extents: 16.00 MiB
  Size of logical volume atomicos/docker-pool_tmeta changed from 8.00 MiB (2 e
xtents) to 16.00 MiB (4 extents).
  Logical volume docker-pool_tmeta successfully resized
  Size of logical volume atomicos/docker-pool_tdata changed from 2.76 GiB (706
 extents) to 10.59 GiB (2711 extents).
  Logical volume docker-pool successfully resized
[fedora@atomichost ~]$ 
```

4. Now, let's look at the current available storage to Docker once again with the `docker info` command:



```
[fedora@atomichost ~]$ sudo docker info
Containers: 3
Images: 17
Storage Driver: devicemapper
 Pool Name: atomicos-docker--pool
 Pool Blocksize: 65.54 kB
 Backing Filesystem: xfs
 Data file:
 Metadata file:
 Data Space Used: 934.7 MB
 Data Space Total: 11.37 GB
 Data Space Available: 10.44 GB
 Metadata Space Used: 1.151 MB
 Metadata Space Total: 16.78 MB
 Metadata Space Available: 15.63 MB
 Udev Sync Supported: true
 Library Version: 1.02.93 (2015-01-30)
Execution Driver: native-0.2
Kernel Version: 4.0.0-0.rc5.git4.1.fc22.x86_64
Operating System: Fedora 22 (Twenty Two)
CPUs: 1
Total Memory: 993.5 MiB
Name: atomichost.localdomain
ID: NBAB:UJOQ:BEXJ:JSGN:TINL:RG4J:A6QM:WFVE:RNZF:WR4M:HYYY:FBML
```

As we can see, both the total data space and metadata space have increased.

How it works...

The procedure is the same as extending any other LVM volume. We create a physical volume on the added disk, add that physical volume to the volume group, and then extend the LVM volumes. Since we are directly accessing the thin pool within Docker, we won't need to create or extend a filesystem or mount the LVM volumes.

There's more...

- In addition to the `DEVS` option, you can also add the `VG` option to the `/etc/sysconfig/docker-storage-setup` file to use a different volume group.
- You can add more than one disk with the `DEVS` option.
- If a disk that is already part of the Volume Group has been mentioned with the `DEVS` option, then the `docker-storage-setup` script will exit, as the existing device has a partition and physical volume already created.
- The `docker-storage-setup` script reserves 0.1 percent of the size for `meta-data`. This is why we saw an increase in the Metadata Space as well.

See also

- The documentation on the Project Atomic website at `http://www.projectatomic.io/docs/docker-storage-recommendation/`
- Supported filesystems with Project Atomic at `http://www.projectatomic.io/docs/filesystems/`

Setting up Cockpit for Project Atomic

Cockpit (`http://cockpit-project.org/`) is a server manager that makes it easy to administer your GNU/Linux servers via a web browser. It can be used to manage the Project Atomic host as well. More than one host can be managed through one Cockpit instance. Cockpit does not come by default with the latest Project Atomic, and you will need to start it as a **Super Privileged Container** (**SPC**). SPCs are specially built containers that run with security turned off (`--privileged`); they turn off one or more of the namespaces or "volume mounts in" parts of the host OS into the container. For more details on SPC, refer to `https://developerblog.redhat.com/2014/11/06/introducing-a-super-privileged-container-concept/` and `https://www.youtube.com/watch?v=eJIeGnHtIYg`.

Because Cockpit runs as an SPC, it can access the resources needed to manage the Atomic host within the container.

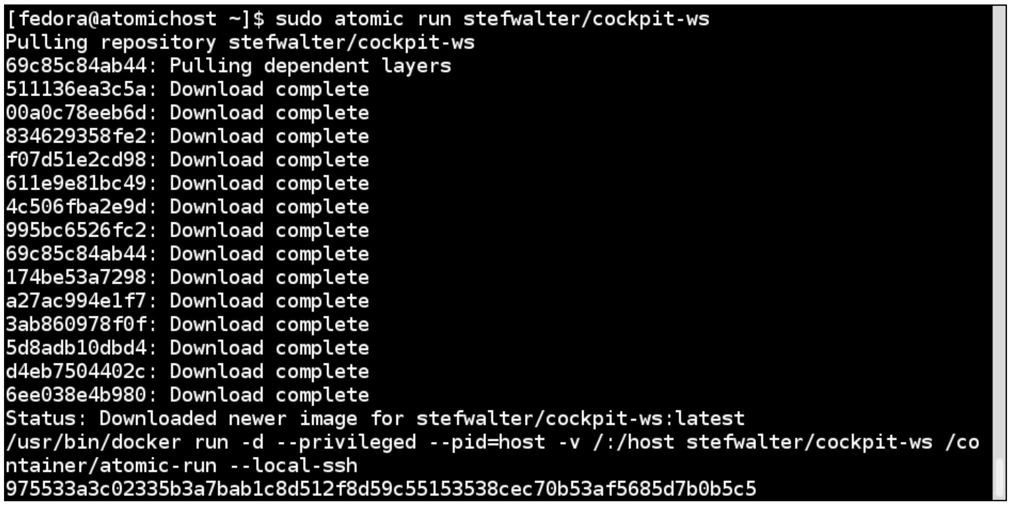
Getting ready

Set up the Project Atomic host and log in to it.

How to do it...

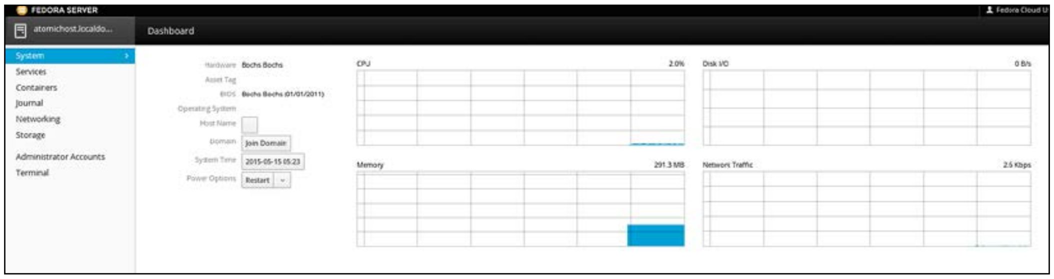
1. Run the following command to start the Cockpit container:

   ```
   [fedora@atomichost ~]$ sudo atomic run stefwalter/cockpit-ws
   ```



```
[fedora@atomichost ~]$ sudo atomic run stefwalter/cockpit-ws
Pulling repository stefwalter/cockpit-ws
69c85c84ab44: Pulling dependent layers
511136ea3c5a: Download complete
00a0c78eeb6d: Download complete
834629358fe2: Download complete
f07d51e2cd98: Download complete
611e9e81bc49: Download complete
4c506fba2e9d: Download complete
995bc6526fc2: Download complete
69c85c84ab44: Download complete
174be53a7298: Download complete
a27ac994e1f7: Download complete
3ab860978f0f: Download complete
5d8adb10dbd4: Download complete
d4eb7504402c: Download complete
6ee038e4b980: Download complete
Status: Downloaded newer image for stefwalter/cockpit-ws:latest
/usr/bin/docker run -d --privileged --pid=host -v /:/host stefwalter/cockpit-ws /co
ntainer/atomic-run --local-ssh
975533a3c02335b3a7bab1c8d512f8d59c55153538cec70b53af5685d7b0b5c5
```

2. Open the browser (`http://<VM IP>:9090`) and log in with the default user/password `fedora/atomic`. Once logged in, you can select the current host to manage. You will see a screen as shown here:



Hardware Bochs Bochs
Asset Tag
BIOS Bochs Bochs (01/01/2011)
Operating System
Host Name
Domain Join Domain
System Time 2015-05-15 05:23
Power Options Restart

How it works...

Here, we used the `atomic` command instead of the `docker` command to start the container. Let's look at the Cockpit Dockerfile (`https://github.com/fedora-cloud/Fedora-Dockerfiles/blob/master/cockpit-ws/Dockerfile`) to see why we did that. In the Dockerfile you will see some instructions:

```
LABEL INSTALL /usr/bin/docker run -ti --rm --privileged -v /:/host
IMAGE /container/atomic-install
LABEL UNINSTALL /usr/bin/docker run -ti --rm --privileged -v
/:/host IMAGE /cockpit/atomic-uninstall
LABEL RUN /usr/bin/docker run -d --privileged --pid=host -v
/:/host IMAGE /container/atomic-run --local-ssh
```

If you recall from *Chapter 2*, *Working with Docker Containers* and *Chapter 3*, *Working with Docker Images*, we could assign metadata to images and containers using labels. `INSTALL`, `UNINSTALL`, and `RUN` are labels here. The `atomic` command is a command specific to Project Atomic, which reads those labels and performs operations. As the container is running as an SPC, it does not need port forwarding from host to container. For more details on the `atomic` command, please visit `https://developerblog.redhat.com/2015/04/21/introducing-the-atomic-command/`.

There's more...

You can perform almost all administrator tasks from the GUI for the given system. You can manage Docker images/containers through this. You can perform operations such as:

- Pulling an image
- Starting/stopping the containers

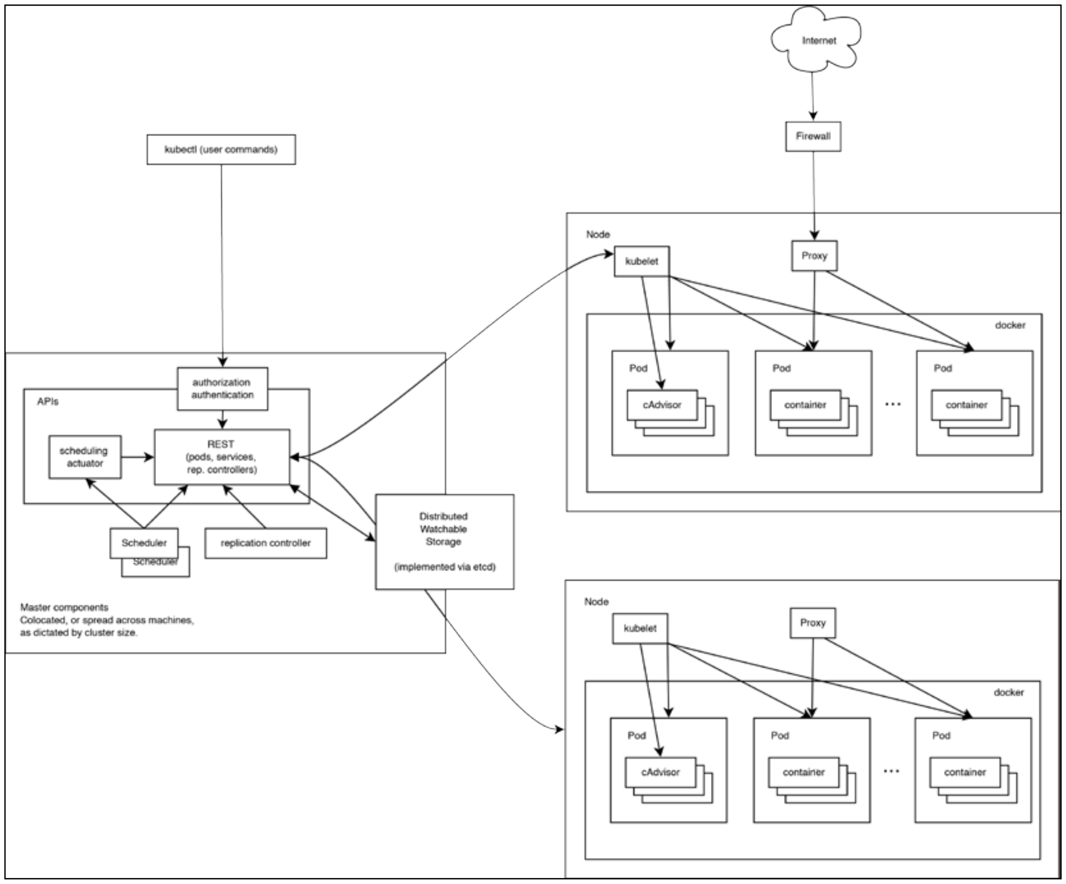
You can also add other machines to the same Cockpit instance so that you manage them from one central location.

See also

- The Cockpit documentation at `http://files.cockpit-project.org/guide/`

Setting up a Kubernetes cluster

Kubernetes is an open source container orchestration tool across multiple nodes in the cluster. Currently, it only supports Docker. It was started by Google, and now developers from other companies are contributing to it. It provides mechanisms for application deployment, scheduling, updating, maintenance, and scaling. Kubernetes' auto-placement, auto-restart, auto-replication features make sure that the desired state of the application is maintained, which is defined by the user. Users define applications through YAML or JSON files, which we'll see later in the recipe. These YAML and JSON files also contain the API Version (the `apiVersion` field) to identify the schema. The following is the architectural diagram of Kubernetes:



https://raw.githubusercontent.com/GoogleCloudPlatform/kubernetes/master/docs/architecture.png

Let's look at some of the key components and concepts of Kubernetes.

- **Pods**: A pod, which consists of one or more containers, is the deployment unit of Kubernetes. Each container in a pod shares different namespaces with other containers in the same pod. For example, each container in a pod shares the same network namespace, which means they can all communicate through localhost.
- **Node/Minion**: A node, which was previously known as a minion, is a worker node in the Kubernetes cluster and is managed through master. Pods are deployed on a node, which has the necessary services to run them:
    - docker, to run containers
    - kubelet, to interact with master
    - proxy (kube-proxy), which connects the service to the corresponding pod
- **Master**: Master hosts cluster-level control services such as the following:
    - **API server**: This has RESTful APIs to interact with master and nodes. This is the only component that talks to the etcd instance.
    - **Scheduler**: This schedules jobs in clusters, such as creating pods on nodes.
    - **Replication controller**: This ensures that the user-specified number of pod replicas is running at any given time. To manage replicas with replication controller, we have to define a configuration file with the replica count for a pod.

  Master also communicates with etcd, which is a distributed key-value pair. etcd is used to store the configuration information, which is used by both master and nodes. The watch functionality of etcd is used to notify the changes in the cluster. etcd can be hosted on master or on a different set of systems.
- **Services**: In Kubernetes, each pod gets its own IP address, and pods are created and destroyed every now and then based on the replication controller configuration. So, we cannot rely on a pod's IP address to cater an app. To overcome this problem, Kubernetes defines an abstraction, which defines a logical set of pods and policies to access them. This abstraction is called a service. Labels are used to define the logical set, which a service manages.
- **Labels**: Labels are key-value pairs that can be attached to objects like, using which we select a subset of objects. For example, a service can select all pods with the label `mysql`.
- **Volumes**: A volume is a directory that is accessible to the containers in a pod. It is similar to Docker volumes but not the same. Different types of volumes are supported in Kubernetes, some of which are EmptyDir (ephemeral), HostDir, GCEPersistentDisk, and NFS. Active development is happening to support more types of volumes. More details can be found at `https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/volumes.md`.

Kubernetes can be installed on VMs, physical machines, and the cloud. For the complete matrix, take a look at `https://github.com/GoogleCloudPlatform/kubernetes/tree/master/docs/getting-started-guides`. In this recipe, we'll see how to install it on VMs, using Vagrant with VirtualBox provider. This recipe and the following recipes on Kubernetes, were tried on v0.17.0 of Kubernetes.

Getting ready

1. Install latest Vagrant >= 1.6.2 from `http://www.vagrantup.com/downloads.html`.
2. Install the latest VirtualBox from `https://www.virtualbox.org/wiki/Downloads`. Detailed instructions on how to set this up are outside the scope of this book.

How to do it...

1. Run the following command to set up Kubernetes on Vagrant VMs:

```
$ export KUBERNETES_PROVIDER=vagrant
$ export VAGRANT_DEFAULT_PROVIDER=virtualbox
$ curl -sS https://get.k8s.io | bash
```

How it works...

The bash script downloaded from the `curl` command, first downloads the latest Kubernetes release and then runs the `./kubernetes/cluster/kube-up.sh` bash script to set up the Kubernetes environment. As we have specified Vagrant as `KUBERNETES_PROVIDER`, the script first downloads the Vagrant images and then, using Salt (`http://saltstack.com/`), configures one master and one node (minion) VM. Initial setup takes a few minutes to run.

Vagrant creates a credential file in `~/.kubernetes_vagrant_auth` for authentication.

There's more...

Similar to `./cluster/kube-up.sh`, there are other helper scripts to perform different operations from the host machine itself. Make sure you are in the `kubernetes` directory, which was created with the preceding installation, while running the following commands:

- Get the list of nodes:

```
$ ./cluster/kubectl.sh get nodes
```

- Get the list of pods:

```
$ ./cluster/kubectl.sh get pods
```

- Get the list of services:

  ```
  $ ./cluster/kubectl.sh get services
  ```

- Get the list of replication controllers:

  ```
  $ ./cluster/kubectl.sh get replicationControllers
  ```

- Destroy the vagrant cluster:

  ```
  $ ./cluster/kube-down.sh
  ```

- Then bring back the vagrant cluster:

  ```
  $ ./cluster/kube-up.sh
  ```

You will see some `pods`, `services`, and `replicationControllers` listed, as Kubernetes creates them for internal use.

See also

- Setting up the Vagrant environment at `https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/getting-started-guides/vagrant.md`
- The Kubernetes user guide at `https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/user-guide.md`
- Kubernetes API conventions at `https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/api-conventions.md`

Scaling up and down in a Kubernetes cluster

In the previous section, we mentioned that the replication controller ensures that the user-specified number of pod replicas is running at any given time. To manage replicas with the replication controller, we have to define a configuration file with the replica count for a pod. This configuration can be changed at runtime.

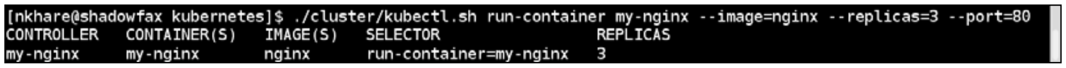
Getting ready

Make sure the Kubernetes setup is running as described in the preceding recipe and that you are in the `kubernetes` directory, which was created with the preceding installation.

How to do it...

1. Start the `nginx` container with a replica count of 3:

   ```
   $ ./cluster/kubectl.sh run-container my-nginx --image=nginx
   --replicas=3 --port=80
   ```



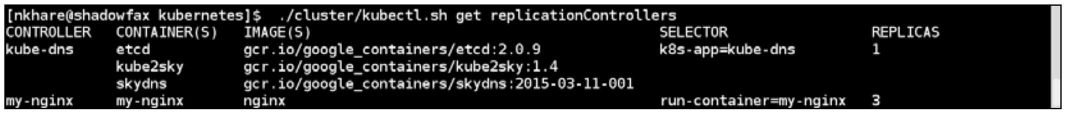
```
[nkhare@shadowfax kubernetes]$ ./cluster/kubectl.sh run-container my-nginx --image=nginx --replicas=3 --port=80
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR                 REPLICAS
my-nginx     my-nginx       nginx      run-container=my-nginx   3
```

   This will start three replicas of the `nginx` container. List the pods to get the status:

   ```
   $  ./cluster/kubectl.sh get pods
   ```

2. Get the replication controller configuration:

   ```
   $ ./cluster/kubectl.sh get replicationControllers
   ```

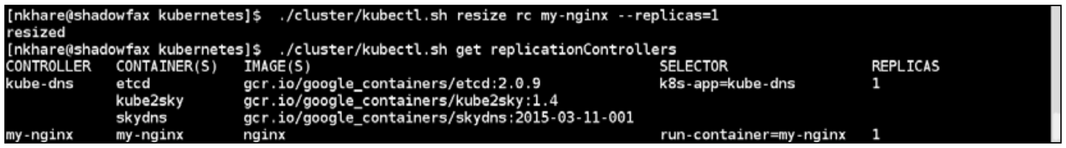


```
[nkhare@shadowfax kubernetes]$  ./cluster/kubectl.sh get replicationControllers
CONTROLLER   CONTAINER(S)   IMAGE(S)                                         SELECTOR                 REPLICAS
kube-dns     etcd           gcr.io/google_containers/etcd:2.0.9              k8s-app=kube-dns         1
             kube2sky       gcr.io/google_containers/kube2sky:1.4
             skydns         gcr.io/google_containers/skydns:2015-03-11-001
my-nginx     my-nginx       nginx                                            run-container=my-nginx   3
```

   As you can see, we have a `my-nginx` controller, which has a replica count of 3. There is a replication controller for `kube-dns`, which we will explore in next recipe.

3. Request the replication controller service to scale down to replica of 1 and update the replication controller:

   ```
   $ ./cluster/kubectl.sh resize rc my-nginx –replicas=1
   $ ./cluster/kubectl.sh get rc
   ```



```
[nkhare@shadowfax kubernetes]$  ./cluster/kubectl.sh resize rc my-nginx --replicas=1
resized
[nkhare@shadowfax kubernetes]$  ./cluster/kubectl.sh get replicationControllers
CONTROLLER   CONTAINER(S)   IMAGE(S)                                         SELECTOR                 REPLICAS
kube-dns     etcd           gcr.io/google_containers/etcd:2.0.9              k8s-app=kube-dns         1
             kube2sky       gcr.io/google_containers/kube2sky:1.4
             skydns         gcr.io/google_containers/skydns:2015-03-11-001
my-nginx     my-nginx       nginx                                            run-container=my-nginx   1
```

4. Get the list of pods to verify; you should see only one pod for `nginx`:

   ```
   $  ./cluster/kubectl.sh get pods
   ```

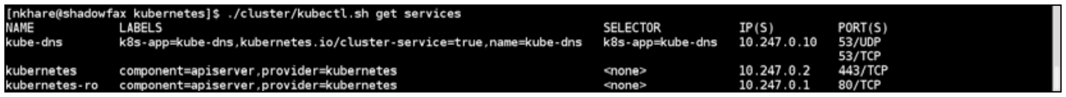
How it works...

We request the replication controller service running on master to update the replicas for a pod, which updates the configuration and requests nodes/minions to act accordingly to honor the resizing.

There's more...

Get the services:

```
$ ./cluster/kubectl.sh get services
```



```
[nkhare@shadowfax kubernetes]$ ./cluster/kubectl.sh get services
NAME            LABELS                                                        SELECTOR            IP(S)          PORT(S)
kube-dns        k8s-app=kube-dns,kubernetes.io/cluster-service=true,name=kube-dns  k8s-app=kube-dns    10.247.0.10    53/UDP
                                                                                                                 53/TCP
kubernetes      component=apiserver,provider=kubernetes                       <none>              10.247.0.2     443/TCP
kubernetes-ro   component=apiserver,provider=kubernetes                       <none>              10.247.0.1     80/TCP
```

As you can see, we don't have any service defined for our `nginx` containers started earlier. This means that though we have a container running, we cannot access them from outside because the corresponding service is not defined.

See also

- Setting up the Vagrant environment at `https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/getting-started-guides/vagrant.md`
- The Kubernetes user guide at `https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/user-guide.md`

Setting up WordPress with a Kubernetes cluster

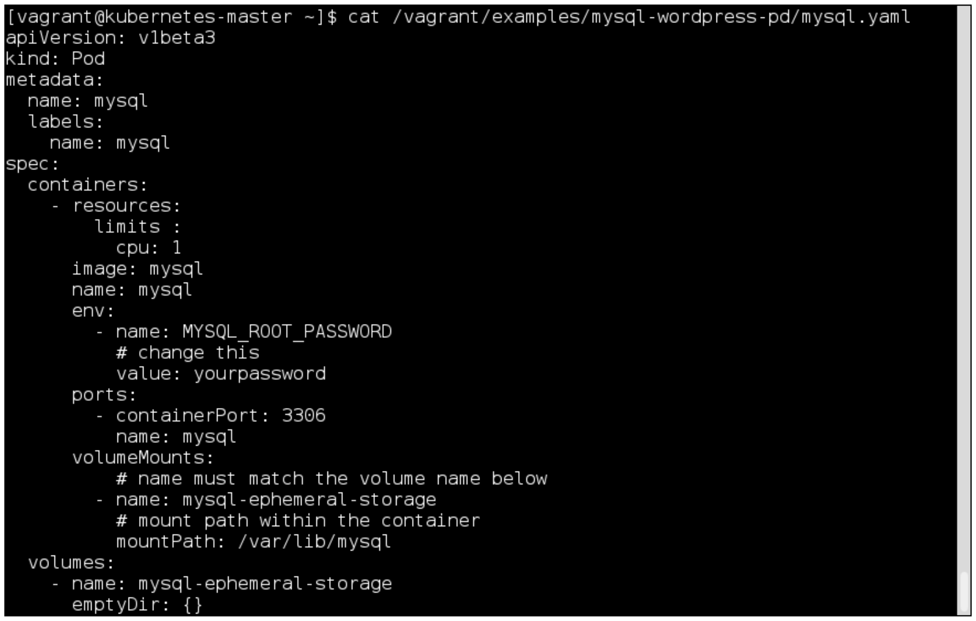
In this recipe, we will use the WordPress example given in the Kubernetes GitHub (`https://github.com/GoogleCloudPlatform/kubernetes/tree/master/examples/mysql-wordpress-pd`). The given example requires some changes, as we'll be running it on the Vagrant environment instead of the default Google Compute engine. Also, instead of using the helper functions (for example, `<kubernetes>/cluster/kubectl.sh`), we'll log in to master and use the `kubectl` binary.

Getting ready

- Make sure the Kubernetes cluster has been set up as described in the previous recipe.
- In the kubernetes directory that was downloaded during the setup, you will find an examples directory that contains many examples. Let's go to the mysql-wordpress-pd directory:

  ```
  $ cd kubernetes/examples/mysql-wordpress-pd
  $  ls *.yaml
  mysql-service.yaml mysql.yaml wordpress-service.yaml  wordpress.yaml
  ```

- These .yaml files describe pods and services for mysql and wordpress respectively.
- In the pods files (mysql.yaml and wordpress.yaml), you will find the section on volumes and the corresponding volumeMount file. The original example assumes that you have access to Google Compute Engine and that you have the corresponding storage setup. For simplicity, we will not set up that and instead use ephemeral storage with the EmptyDir volume option. For reference, our mysql.yaml will look like the following:



```
[vagrant@kubernetes-master ~]$ cat /vagrant/examples/mysql-wordpress-pd/mysql.yaml
apiVersion: v1beta3
kind: Pod
metadata:
  name: mysql
  labels:
    name: mysql
spec:
  containers:
    - resources:
        limits :
          cpu: 1
      image: mysql
      name: mysql
      env:
        - name: MYSQL_ROOT_PASSWORD
          # change this
          value: yourpassword
      ports:
        - containerPort: 3306
          name: mysql
      volumeMounts:
          # name must match the volume name below
        - name: mysql-ephemeral-storage
          # mount path within the container
          mountPath: /var/lib/mysql
  volumes:
    - name: mysql-ephemeral-storage
      emptyDir: {}
```

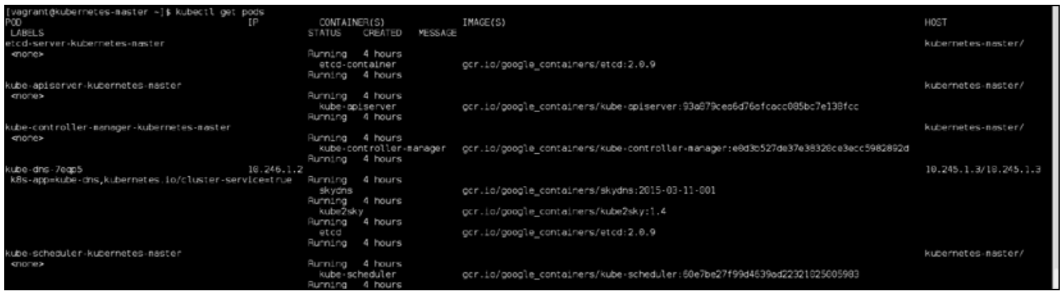
- Make the similar change to wordpress.yaml.

How to do it...

1. With SSH, log in to the master node and look at the running pods:

   $ vagrant ssh master

   $ kubectl get pods



```
[vagrant@kubernetes-master ~]$ kubectl get pods
POD                                              IP              CONTAINER(S)              IMAGE(S)                                                                       HOST
 LABELS                                                         STATUS     CREATED   MESSAGE
etcd-server-kubernetes-master                                                                                                                                              kubernetes-master/
 <none>                                                         Running    4 hours
                                                                  etcd-container           gcr.io/google_containers/etcd:2.0.9
                                                                Running    4 hours
kube-apiserver-kubernetes-master                                                                                                                                           kubernetes-master/
 <none>                                                         Running    4 hours
                                                                  kube-apiserver           gcr.io/google_containers/kube-apiserver:93a879cea6d76afcacc085bc7e138fcc
                                                                Running    4 hours
kube-controller-manager-kubernetes-master                                                                                                                                  kubernetes-master/
 <none>                                                         Running    4 hours
                                                                  kube-controller-manager  gcr.io/google_containers/kube-controller-manager:e0d3b527de37e38328ce3ecc5982892d
                                                                Running    4 hours
kube-dns-7eqp5                                   10.246.1.2                                                                                                                10.245.1.3/10.245.1.3
 k8s-app=kube-dns,kubernetes.io/cluster-service=true            Running    4 hours
                                                                  skydns                   gcr.io/google_containers/skydns:2015-03-11-001
                                                                Running    4 hours
                                                                  kube2sky                 gcr.io/google_containers/kube2sky:1.4
                                                                Running    4 hours
                                                                  etcd                     gcr.io/google_containers/etcd:2.0.9
                                                                Running    4 hours
kube-scheduler-kubernetes-master                                                                                                                                           kubernetes-master/
 <none>                                                         Running    4 hours
                                                                  kube-scheduler           gcr.io/google_containers/kube-scheduler:60e7be27f99d4639ad22321025005983
                                                                Running    4 hours
```

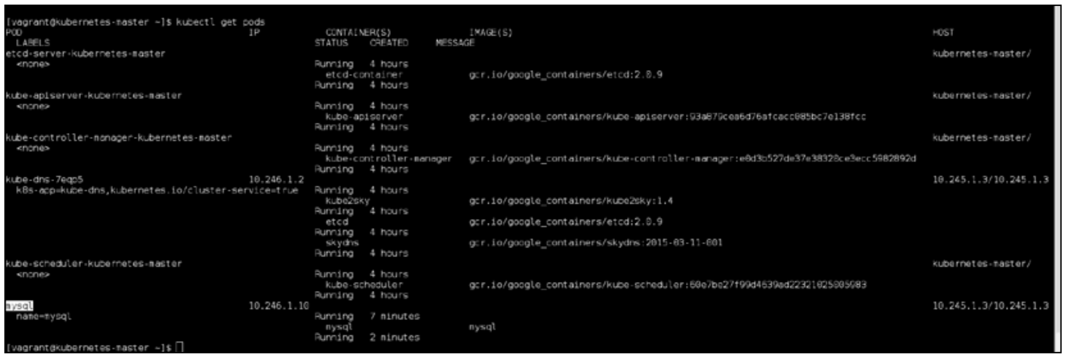
   The `kube-dns-7eqp5` pod consists of three containers: `etcd`, `kube2sky`, and `skydns`, which are used to configure an internal DNS server for service name to IP resolution. We'll see it in action later in this recipe.

   The Vagrantfile used in this example is created so that the `kubernetes` directory that we created earlier is shared under `/vagrant` on VM, which means that the changes we made to the host system will be visible here as well.

2. From the master node, create the `mysql` pod and check the running pods:

   $ kubectl create -f /vagrant/examples/mysql-wordpress-pd/mysql.yaml

   $ kubectl get pods



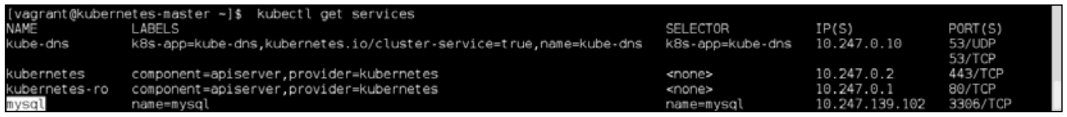
```
[vagrant@kubernetes-master ~]$ kubectl get pods
POD                                              IP              CONTAINER(S)              IMAGE(S)                                                                       HOST
  LABELS                                                        STATUS     CREATED    MESSAGE
etcd-server-kubernetes-master                                                                                                                                              kubernetes-master/
  <none>                                                        Running    4 hours
                                                                  etcd-container           gcr.io/google_containers/etcd:2.0.9
                                                                Running    4 hours
kube-apiserver-kubernetes-master                                                                                                                                           kubernetes-master/
  <none>                                                        Running    4 hours
                                                                  kube-apiserver           gcr.io/google_containers/kube-apiserver:93a879cea6d76afcacc085bc7e138fcc
                                                                Running    4 hours
kube-controller-manager-kubernetes-master                                                                                                                                  kubernetes-master/
  <none>                                                        Running    4 hours
                                                                  kube-controller-manager  gcr.io/google_containers/kube-controller-manager:e0d3b527de37e38328ce3ecc5982892d
                                                                Running    4 hours
kube-dns-7eqp5                                   10.246.1.2                                                                                                                10.245.1.3/10.245.1.3
  k8s-app=kube-dns,kubernetes.io/cluster-service=true           Running    4 hours
                                                                  kube2sky                 gcr.io/google_containers/kube2sky:1.4
                                                                Running    4 hours
                                                                  etcd                     gcr.io/google_containers/etcd:2.0.9
                                                                Running    4 hours
                                                                  skydns                   gcr.io/google_containers/skydns:2015-03-11-001
                                                                Running    4 hours
kube-scheduler-kubernetes-master                                                                                                                                           kubernetes-master/
  <none>                                                        Running    4 hours
                                                                  kube-scheduler           gcr.io/google_containers/kube-scheduler:60e7be27f99d4639ad22321025005983
                                                                Running    4 hours
mysql                                            10.246.1.10                                                                                                               10.245.1.3/10.245.1.3
  name=mysql                                                    Running    7 minutes
                                                                  mysql                    mysql
                                                                Running    2 minutes
[vagrant@kubernetes-master ~]$ 
```

   As we can see, a new pod with the `mysql` name has been created and it is running on host `10.245.1.3`, which is our node (minion).

3. Now let's create the service for `mysql` and look at all the services:

   `$ kubectl create -f /vagrant/examples/mysql-wordpress-pd/mysql-service.yaml`

   `$ kubectl get services`



```
[vagrant@kubernetes-master ~]$  kubectl get services
NAME            LABELS                                                              SELECTOR            IP(S)            PORT(S)
kube-dns        k8s-app=kube-dns,kubernetes.io/cluster-service=true,name=kube-dns   k8s-app=kube-dns    10.247.0.10      53/UDP
                                                                                                                         53/TCP
kubernetes      component=apiserver,provider=kubernetes                             <none>              10.247.0.2       443/TCP
kubernetes-ro   component=apiserver,provider=kubernetes                             <none>              10.247.0.1       80/TCP
mysql           name=mysql                                                          name=mysql          10.247.139.102   3306/TCP
```

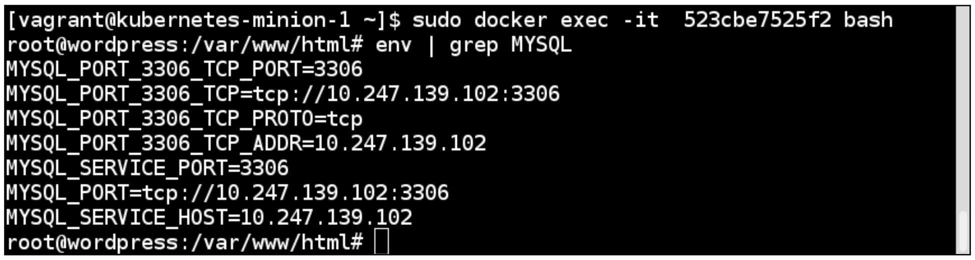
   As we can see, a service named `mysql` has been created. Each service has a Virtual IP. Other than the `kubernetes` services, we see a service named `kube-dns`, which is used as the service name for the `kube-dns` pod we saw earlier.

4. Similar to `mysql`, let's create a pod for `wordpress`:

   `$ kubectl create -f /vagrant/examples/mysql-wordpress-pd/wordpress.yaml`

   With this command, there are a few things happening in the background:

   - The `wordpress` image gets downloaded from the official Docker registry and the container runs.
   - By default, whenever a pod starts, information about all the existing services is exported as environment variables. For example, if we log in to the `wordpress` pod and look for `MYSQL`-specific environment variables, we will see something like the following:



```
[vagrant@kubernetes-minion-1 ~]$ sudo docker exec -it  523cbe7525f2 bash
root@wordpress:/var/www/html# env | grep MYSQL
MYSQL_PORT_3306_TCP_PORT=3306
MYSQL_PORT_3306_TCP=tcp://10.247.139.102:3306
MYSQL_PORT_3306_TCP_PROTO=tcp
MYSQL_PORT_3306_TCP_ADDR=10.247.139.102
MYSQL_SERVICE_PORT=3306
MYSQL_PORT=tcp://10.247.139.102:3306
MYSQL_SERVICE_HOST=10.247.139.102
root@wordpress:/var/www/html# 
```

   - When the WordPress container starts, it runs the `/entrypoint.sh` script, which looks for the environment variables mentioned earlier to start the service. `https://github.com/docker-library/wordpress/blob/master/docker-entrypoint.sh`.
   - With the `kube-dns` service, PHP scripts of `wordpress` are able to the reserve lookup to proceed forward.

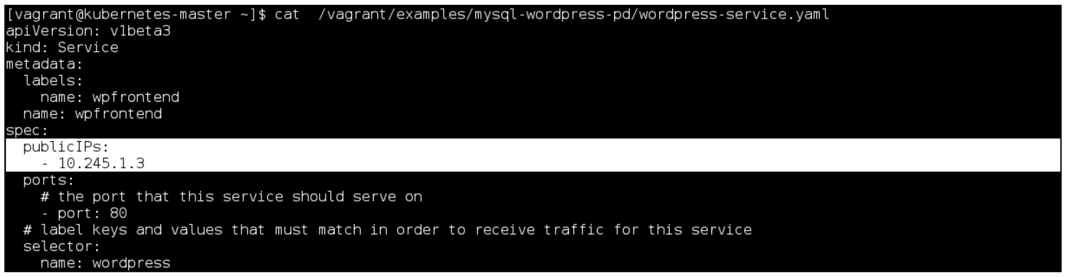
5. After starting the pod, the last step here is to set up the `wordpress` service. In the default example, you will see an entry like the following in the service file (`/vagrant/examples/mysql-wordpress-pd/mysql-service.yaml`):

   ```
   createExternalLoadBalancer: true
   ```

   This has been written to keep in mind that this example will run on the Google Compute Engine. So it is not valid here. In place of that, we will need to make an entry like the following:

   ```
   publicIPs:
       - 10.245.1.3
   ```

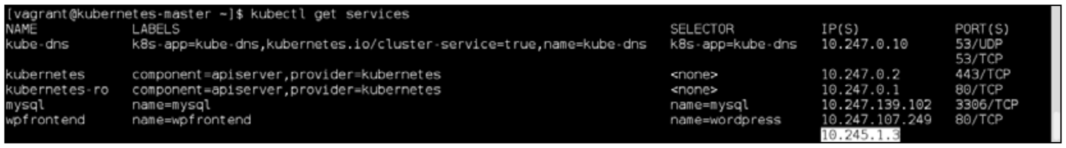
   We have replaced the load-balancer entry with the public IP of the node, which in our case is the IP address of the node (minion). So, the `wordpress` file would look like the following:



```
[vagrant@kubernetes-master ~]$ cat  /vagrant/examples/mysql-wordpress-pd/wordpress-service.yaml
apiVersion: v1beta3
kind: Service
metadata:
  labels:
    name: wpfrontend
  name: wpfrontend
spec:
  publicIPs:
    - 10.245.1.3
  ports:
    # the port that this service should serve on
    - port: 80
  # label keys and values that must match in order to receive traffic for this service
  selector:
    name: wordpress
```

6. To start the `wordpress` service, run the following command from the master node:

   `$ kubectl create -f /vagrant/examples/mysql-wordpress-pd/wordpress-service.yaml`



```
[vagrant@kubernetes-master ~]$ kubectl get services
NAME            LABELS                                                                 SELECTOR            IP(S)            PORT(S)
kube-dns        k8s-app=kube-dns,kubernetes.io/cluster-service=true,name=kube-dns    k8s-app=kube-dns    10.247.0.10      53/UDP
                                                                                                                            53/TCP
kubernetes      component=apiserver,provider=kubernetes                                <none>              10.247.0.2       443/TCP
kubernetes-ro   component=apiserver,provider=kubernetes                                <none>              10.247.0.1       80/TCP
mysql           name=mysql                                                             name=mysql          10.247.139.102   3306/TCP
wpfrontend      name=wpfrontend                                                        name=wordpress      10.247.107.249   80/TCP
                                                                                                           10.245.1.3
```

   We can see here that our service is also available through the node (minion) IP.

7. To verify if everything works fine, we can install the links package on master by which we can browse a URL through the command line and connect to the public IP we mentioned:

   `$ sudo yum install links -y`

   `$ links 10.245.1.3`

   With this, you should see the `wordpress` installation page.

How it works...

In this recipe, we first created a `mysql` pod and service. Later, we connected it to a `wordpress` pod, and to access it, we created a `wordpress` service. Each YAML file has a `kind` key that defines the type of object it is. For example, in pod files, the `kind` is set to pod and in service files, it is set to service.

There's more...

- In this example setup, we have only one Node (minion). If you log in to it, you will see all the running containers:

  ```
  $ vagrant ssh minion-1
  $ sudo docker ps
  ```

- In this example, we have not configured replication controllers. We can extend this example by creating them.

See also

- Setting up the Vagrant environment at `https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/getting-started-guides/vagrant.md`
- The Kubernetes User Guide at `https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/user-guide.md`
- The documentation on kube-dns at `https://github.com/GoogleCloudPlatform/kubernetes/tree/master/cluster/addons/dns`

9
Docker Security

In this chapter, we will cover the following recipes:

- Setting Mandatory Access Control (MAC) with SELinux
- Allowing writes to volume mounted from the host with SELinux ON
- Removing capabilities to breakdown the power of a root user inside the container
- Sharing namespaces between the host and the container

Introduction

Docker containers are not actually Sandbox applications, which means they are not recommended to run random applications on the system as root with Docker. You should always treat a container running a service/process as a service/process running on the host system and put all the security measures inside the container you put on the host system.

We saw in *Chapter 1*, *Introduction and Installation*, how Docker uses namespaces for isolation. The six namespaces that Docker uses are Process, Network, Mount, Hostname, Shared Memory, and User. Not everything in Linux is namespaced, for example, SELinux, Cgroups, Devices (`/dev/mem`, `/dev/sd*`), and Kernel Modules. Filesystems under `/sys`, `/proc/sys`, `/proc/sysrq-trigger`, `/proc/irq`, `/proc/bus` are also not namespaced but they are mounted as read only by default with the libcontainer execution driver.

To make Docker a secure environment, a lot of work has been done in the recent past and more work is underway.

- As Docker images are the basic building blocks, it is very important that we choose the right base image to start with. Docker has the concept of official images, which are maintained by either Docker, the vendor or someone else. If you recall from *Chapter 2*, *Working with Docker Containers*, we can search images on Docker Hub using the following syntax:

  ```
  $ docker search <image name>
  ```

  For example, consider the following command :

  ```
  $ docker search fedora
  ```

  We will see a column `OFFICIAL`, and if the images are official, you will see `[OK]` against that image in that column. There is an experimental feature added in Docker 1.3 (`http://blog.docker.com/2014/10/docker-1-3-signed-images-process-injection-security-options-mac-shared-directories/`), which does Digital Signal Verification of official images after pulling the image. If the image has been tampered with, the user will be notified, but it will not prevent the user from running it. At present, this feature works only with official images. More details about official images can be found at `https://github.com/docker-library/official-images`. The image signing and verification feature is not ready, so as of now, don't completely rely on it.

- In *Chapter 6*, *Docker APIs and Language Bindings*, we saw how we can secure Docker remote API, when Docker daemon access is configured over TCP.
- We can also consider turning off the default intercontainer communication over the network with `--icc=false` on the Docker host. Though containers can still communicate through links, which overrides the default DROP policy of iptables, they get set with the `--icc=false` option.
- We can also set Cgroups resource restrictions through, which we can prevent **Denial of Service** (**DoS**) attacks through system resource constraints.
- Docker takes advantage of the special device, Cgroups that allows us to specify which device nodes can be used within the container. It blocks the processes from creating and using device nodes that could be used to attack the host.
- Any device node precreated on the image cannot be used to talk to kernel because images are mounted with the `nodev` option.

The following are some guidelines (may not be complete), which one can follow to have a secure Docker environment:

- Run services as nonroot and treat the root in the container, as well as outside the container, as root.
- Use images from trusted parties to run the container; avoid using the `-insecure-registry=[]` option.
- Don't run the random container from the Docker registry or anywhere else. Red Hat carries patches to add and block registries to give more control to enterprises (`http://rhelblog.redhat.com/2015/04/15/understanding-the-changes-to-docker-search-and-docker-pull-in-red-hat-enterprise-linux-7-1/`).
- Have your host kernel up to date.
- Avoid using `--privileged` whenever possible and drop container privileges as soon as possible.
- Configure **Mandatory Access Control** (**MAC**) through SELinux or AppArmor.
- Collect logs for auditing.
- Do regular auditing.
- Run containers on hosts, which are specially designed to run containers only. Consider using Project Atomic, CoreOS, or similar solutions.
- Mount the devices with the `--device` option rather than using the `--privileged` option to use devices inside the container.
- Prohibit SUID and SGID inside the container.

Recently, Docker and the Center for Internet Security (`http://www.cisecurity.org/`) released a best practices guide for Docker security, which covers most of the preceding guidelines and more guidelines at `https://blog.docker.com/2015/05/understanding-docker-security-and-best-practices/`.

To set the context for some of the recipes in this chapter, let's try an experiment on the default installation on Fedora 21 with Docker installed.

1. Disable SELinux using the following command:

   ```
   $ sudo setenforce 0
   ```

2. Create a user and add it to the default Docker group so that the user can run Docker commands without `sudo`:

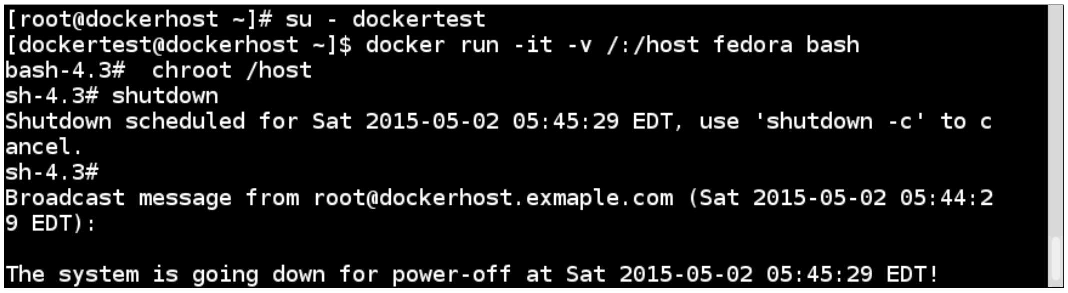
   ```
   $ sudo useradd dockertest
   $ sudo passwd dockertest
   $ sudo groupadd docker
   $ sudo gpasswd -a dockertest docker
   ```

3. Log in using the user we created earlier, start a container as follows:

   ```
   $ su - dockertest
   $ docker run -it -v /:/host fedora bash
   ```

4. From the container chroot to `/host` and run the `shutdown` command:

   ```
   $ chroot /host
   $ shutdown
   ```



```
[root@dockerhost ~]# su - dockertest
[dockertest@dockerhost ~]$ docker run -it -v /:/host fedora bash
bash-4.3#  chroot /host
sh-4.3# shutdown
Shutdown scheduled for Sat 2015-05-02 05:45:29 EDT, use 'shutdown -c' to c
ancel.
sh-4.3#
Broadcast message from root@dockerhost.exmaple.com (Sat 2015-05-02 05:44:2
9 EDT):

The system is going down for power-off at Sat 2015-05-02 05:45:29 EDT!
```

As we can see, a user in the Docker group can shut down the host system. Docker currently does not have authorization control, so if you can communicate to the Docker socket, you are allowed to run any Docker command. It is similar to `/etc/sudoers`.

```
USERNAME ALL=(ALL) NOPASSWD: ALL
```

This is really not good. Let's see how we can guard against this and more in the rest of the chapter.

Setting Mandatory Access Control (MAC) with SELinux

It is recommended that you set up some form of MAC on the Docker host either through SELinux or AppArmor, depending on the Linux distribution. In this recipe, we'll see how to set up SELinux on a Fedora/RHEL/CentOS installed system. Let's first look at what SELinux is:

- SELinux is a labeling system
- Every process has a label
- Every file, directory, and system object has a label
- Policy rules control access between labeled processes and labeled objects
- The kernel enforces the rules

With Docker containers, we use two types of SELinux enforcement:

- **Type enforcement**: This is used to protect the host system from container processes. Each container process is labeled `svirt_lxc_net_t` and each container file is labeled `svirt_sandbox_file_t`. The `svirt_lxc_net_t` type is allowed to manage any content labeled with `svirt_sandbox_file_t`. Container processes can only access/write container files.
- **Multi Category Security enforcement**: By setting type enforcement, all container processes will run with the `svirt_lxc_net_t` label and all content will be labeled with `svirt_sandbox_file_t`. However, only with these settings, we are not protecting one container from another because their labels are the same.

  We use **Multi Category Security** (**MCS**) enforcement to protect one container from another, which is based on **Multi Level Security** (**MLS**). When a container is launched, the Docker daemon picks a random MCS label, for example, `s0:c41,c717` and saves it with the container metadata. When any container process starts, the Docker daemon tells the kernel to apply the correct MCS label. As the MCS label is saved in the metadata, if the container restarts, it gets the same MCS label.

Getting ready

A Fedora/RHEL/CentOS host with the latest version of Docker installed, which can be accessed through a Docker client.

How to do it...

Fedora/RHEL/CentOS gets installed by default with SELinux in enforcing mode and the Docker daemon is set to start with SELinux. To check whether these conditions are being met, perform the following steps.

1. Run the following command to make sure SELinux is enabled:

   ```
   $ getenforce
   ```

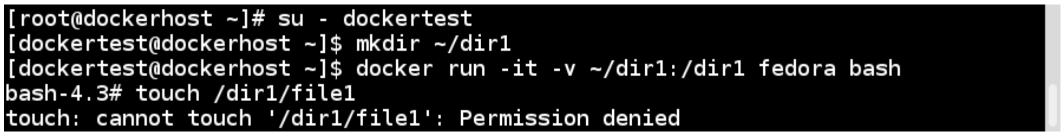
   If the preceding command returns `enforcing`, then it's all good, else we need to change it by updating SELinux configuration file (`/etc/selinux/config`) and rebooting the system.

2. Docker should be running with the `--selinux-enabled` option. You can check the `OPTIONS` section in the Docker daemon configuration (`/etc/sysconfig/docker`) file. Also, cross-check whether the Docker service has started with the SELinux option:

   ```
   $ systemctl status docker
   ```

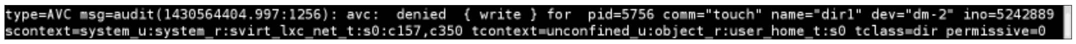
   The preceding command assumes that you are not starting Docker in daemon mode manually.

Let's start a container (without the privileged option) after mounting a host directory as volume and try to create a file in that:



```
[root@dockerhost ~]# su - dockertest
[dockertest@dockerhost ~]$ mkdir ~/dir1
[dockertest@dockerhost ~]$ docker run -it -v ~/dir1:/dir1 fedora bash
bash-4.3# touch /dir1/file1
touch: cannot touch '/dir1/file1': Permission denied
```

As expected, we see `Permission denied` because a container process with the `svirt_lxc_net_t` label cannot create files on the host's filesystem. If we look at the SELinux logs (`/var/log/audit.log`) on the host, we will see messages similar to the following:



```
type=AVC msg=audit(1430564404.997:1256): avc:  denied  { write } for  pid=5756 comm="touch" name="dir1" dev="dm-2" ino=5242889
scontext=system_u:system_r:svirt_lxc_net_t:s0:c157,c350 tcontext=unconfined_u:object_r:user_home_t:s0 tclass=dir permissive=0
```

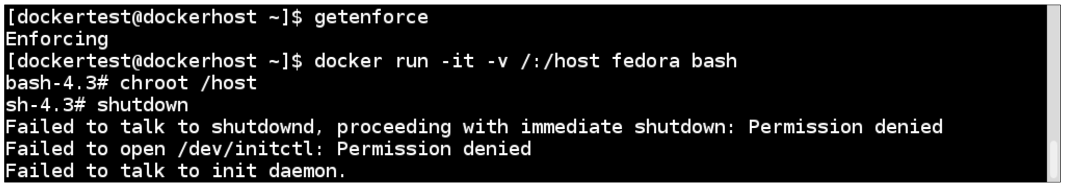
The `s0:c157,c350` label is the MCS label on the container.

How it works…

SELinux sets both Type and Multi Category Security enforcement when the right options are set for SELinux and Docker. The Linux kernel enforces these enforcements.

There's more…

- If SELinux is in enforcing mode and the Docker daemon is configured to use SELinux, then we will not be able to shut down the host from the container, like we did earlier in this chapter:



```
[dockertest@dockerhost ~]$ getenforce
Enforcing
[dockertest@dockerhost ~]$ docker run -it -v /:/host fedora bash
bash-4.3# chroot /host
sh-4.3# shutdown
Failed to talk to shutdownd, proceeding with immediate shutdown: Permission denied
Failed to open /dev/initctl: Permission denied
Failed to talk to init daemon.
```

- As we know, by default, all the containers will run with the `svirt_lxc_net_t` label, but we can also adjust SELinux labels for custom requirements. Visit the *Adjusting SELinux labels* section of `http://opensource.com/business/15/3/docker-security-tuning`.
- Setting up MLS with Docker containers is also possible. Visit the *Multi Level Security mode* section of `http://opensource.com/business/15/3/docker-security-tuning`.

See also

- *The SELinux Coloring Book*; visit `https://people.redhat.com/duffy/selinux/selinux-coloring-book_A4-Stapled.pdf`

Allowing writes to volume mounted from the host with SELinux ON

As we saw in the earlier recipe, when SELinux is configured, a nonprivileged container cannot access files on the volume created after mounting the directory from the host system. However, sometimes it is needed to allow access to host files from the container. In this recipe, we'll see how to allow access in such cases.

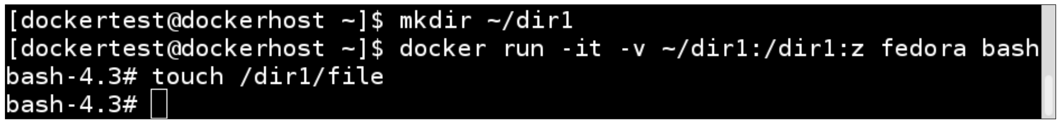
Getting ready

A Fedora/RHEL/CentOS host with the latest version of Docker installed, which can be accessed through a Docker client. Also, SELinux is set to enforcing mode and the Docker daemon is configured to use SELinux.

How to do it...

1. Mount the volume with the `z` or `Z` option as follows:

   ```
   $ docker run -it -v /tmp/:/tmp/host:z docker.io/fedora bash
   $ docker run -it -v /tmp/:/tmp/host:Z docker.io/fedora bash
   ```



```
[dockertest@dockerhost ~]$ mkdir ~/dir1
[dockertest@dockerhost ~]$ docker run -it -v ~/dir1:/dir1:z fedora bash
bash-4.3# touch /dir1/file
bash-4.3# 
```

How it works...

While mounting the volume, Docker will relabel to the volume to allow access. From the man page of Docker run.

The `z` option tells Docker that the volume content will be shared between containers. Docker will label the content with a shared content label. The shared volume labels allow all containers to read/write content. The `Z` option tells Docker to label the content with a private unshared label. Private volumes can only be used by the current container.

See also

- The *Volume mounts* section at `http://opensource.com/business/14/9/security-for-docker`

Removing capabilities to breakdown the power of a root user inside a container

In simple terms, with capabilities, we can breakdown the power of a root user. From the man page for *capabilities*:

> *For the purpose of performing permission checks, traditional UNIX implementations distinguish two categories of processes: privileged processes (whose effective user ID is 0, referred to as superuser or root), and unprivileged processes (whose effective UID is nonzero). Privileged processes bypass all kernel permission checks, while unprivileged processes are subject to full permission checking based on the process's credentials (usually: effective UID, effective GID, and supplementary group list).*
>
> *Starting with kernel 2.2, Linux divides the privileges traditionally associated with superuser into distinct units, known as capabilities, which can be independently enabled and disabled. Capabilities are a per-thread attribute.*

Some example capabilities are:

- `CAP_SYSLOG`: This modifies kernel printk behavior
- `CAP_NET_ADMIN`: This configures the network
- `CAP_SYS_ADMIN`: This helps you to catch all the capabilities

There are only 32 slots available for capabilities in the kernel. There is one capability, `CAP_SYS_ADMIN`, that catches all capabilities; this is used whenever in doubt.

In version 1.2, Docker added some features to add or remove the capabilities for a container. It uses the `chown`, `dac_override`, `fowner`, `kill`, `setgid`, `setuid`, `setpcap`, `net_bind_service`, `net_raw`, `sys_chroot`, `mknod`, `setfcap`, and `audit_write` capabilities by default and removes the following capabilities for a container by default.

- `CAP_SETPCAP`: This modifies the process capabilities
- `CAP_SYS_MODULE`: This inserts/removes the kernel modules
- `CAP_SYS_RAWIO`: This modifies the kernel memory
- `CAP_SYS_PACCT`: This configures process accounting
- `CAP_SYS_NICE`: This modifies the priority of processes

- `CAP_SYS_RESOURCE`: This overrides the resource limits
- `CAP_SYS_TIME`: This modifies the system clock
- `CAP_SYS_TTY_CONFIG`: This configures `tty` devices
- `CAP_AUDIT_WRITE`: This writes the audit log
- `CAP_AUDIT_CONTROL`: This configures the audit subsystem
- `CAP_MAC_OVERRIDE`: This ignores the kernel MAC policy
- `CAP_MAC_ADMIN`: This configures MAC configuration
- `CAP_SYSLOG`: This modifies kernel printk behavior
- `CAP_NET_ADMIN`: This configures the network
- `CAP_SYS_ADMIN`: This helps you catch all the containers

We need to be very careful what capabilities we remove, as applications can break if they don't have enough capabilities to run. To add and remove the capabilities for the container, you can use the `--cap-add` and `--cap-drop` options respectively.

Getting ready

A host with the latest version of Docker installed, which can be accessed through a Docker client.

How to do it...

1. To drop capabilities, run a command similar to the following:

   ```
   $ docker run --cap-drop <CAPABILITY> <image> <command>
   ```

   To remove the `setuid` and `setgid` capabilities from the container so that it cannot run binaries, which have these bits set, run the following command:

   ```
   $ docker run -it --cap-drop  setuid --cap-drop setgid fedora
   bash
   ```

2. Similarly, to add capabilities, run a command similar to the following:

   ```
   $ docker run --cap-add <CAPABILITY> <image> <command>
   ```

   To add all the capabilities and just drop `sys-admin`, run the following command:

   ```
   $ docker run -it --cap-add all --cap-drop sys-admin fedora
   bash
   ```

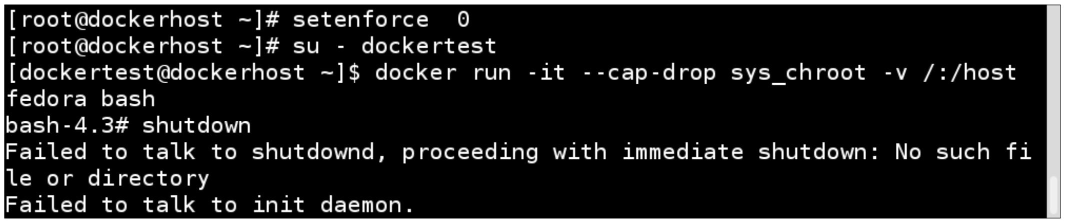
How it works...

Before starting the container, Docker sets up the capabilities for the root user inside the container, which affects the command execution for the container process.

There's more...

Let's revisit the example we saw at the beginning of this chapter, through which we saw the host system shut down through a container. Let SELinux be disabled on the host system; however, while starting the container, drop the `sys_choot` capability:

```
$ docker run -it --cap-drop  sys_chroot -v /:/host  fedora bash
$ shutdown
```



```
[root@dockerhost ~]# setenforce  0
[root@dockerhost ~]# su - dockertest
[dockertest@dockerhost ~]$ docker run -it --cap-drop sys_chroot -v /:/host
fedora bash
bash-4.3# shutdown
Failed to talk to shutdownd, proceeding with immediate shutdown: No such fi
le or directory
Failed to talk to init daemon.
```

See also

- Dan Walsh's articles on opensource.com at `http://opensource.com/business/14/9/security-for-docker`.
- The Docker 1.2 release notes at `http://blog.docker.com/2014/08/announcing-docker-1-2-0/`.
- There are efforts on to selectively disable system calls from container processes to provide tighter security. Visit the *Seccomp* section of `http://opensource.com/business/15/3/docker-security-future`.
- Similar to custom namespaces and capabilities with version 1.6, Docker supports the `--cgroup-parent` flag to pass specific Cgroup to run containers. `https://docs.docker.com/v1.6/release-notes/`.

Sharing namespaces between the host and the container

As we know, while starting the container, by default, Docker creates six different namespaces—Process, Network, Mount, Hostname, Shared Memory, and User for a container. In some cases, we might want to share a namespace between two or more containers. For example, in Kubernetes, all containers in a pod share the same network namespace.

In some cases, we would want to share the namespaces of the host system with the containers. For example, we share the same network namespace between the host and the container to get near line speed inside the container. In this recipe, we will see how to share namespaces between the host and the container.

Getting ready

A host with the latest version of Docker installed, which can be accessed through a Docker client.

How to do it...

1. To share the host network namespace with the container, run the following command:

   ```
   $ docker run -it  --net=host fedora bash
   ```

   If you see the network details inside the container, run the following command:

   ```
   $ ip a
   ```

   You will see a result same as the host.

2. To share the host network, PID, and IPC namespaces with the container, run the following command:

   ```
   $ docker run -it --net=host --pid=host --ipc=host fedora bash
   ```

How it works...

Docker does not create separate namespaces for containers when such arguments are passed to the container.

There's more...

For hosts that are built to run just containers, such as Project Atomic (`http://www.projectatomic.io/`), which we saw in *Chapter 8*, *Docker Orchestration and Hosting Platforms*, doesn't have debugging tools such as `tcpdump` and `sysstat` on the host system. So we have created containers with those tools and have access to host resources. In such cases, sharing namespaces between the host and the container becomes handy. You can read more about it at the following links:

- `http://developerblog.redhat.com/2014/11/06/introducing-a-super-privileged-container-concept/`
- `http://developerblog.redhat.com/2015/03/11/introducing-the-rhel-container-for-rhel-atomic-host/`

See also

- Dan Walsh's documentation on Docker Security at `http://opensource.com/business/15/3/docker-security-tuning`

10
Getting Help and Tips and Tricks

In this chapter, we will see the following recipes:

- Starting Docker in debug mode
- Building a Docker binary from the source
- Building images without using cached layers
- Building your own bridge for container communication
- Changing the default execution driver of Docker
- Selecting the logging driver for containers
- Getting real-time Docker events for containers

Introduction

We'll become more curious as we learn more about Docker. Mailing lists and IRC channels are the best places to get help, learn, and share knowledge about Docker. Docker has a few IRC channels on the free node, such as `#docker` and `#docker-dev`, to discuss Docker in general and dev-related stuff respectively. Similarly, Docker has two mailing lists:

- The Docker user list available at `https://groups.google.com/forum/#!forum/docker-user`
- The Docker dev list available at `https://groups.google.com/forum/#!forum/docker-dev`

While working on Docker, if you find any bugs, you can report them on GitHub at `https://github.com/docker/docker/issues`.

Similarly, if you have fixed a bug, you can send the pull request, which will get reviewed and then get merged to the code base.

Docker also has a forum and a YouTube channel, which are great learning resources and can be found at `https://forums.docker.com/` and `https://www.youtube.com/user/dockerrun` respectively.

There are many Docker meet up groups around the world, where you meet like-minded individuals and learn by sharing experiences at `https://www.docker.com/community/meetups/`.

In this chapter, I'll also put down a few tips and tricks, which will help you to work better with Docker.

Starting Docker in debug mode

We can start Docker in debug mode to debug logs.

Getting ready

Install Docker on the system.

How to do it…

1. Start the Docker daemon with the debug option `-D`. To start from the command line, you can run the following command:

   ```
   $ docker -d -D
   ```

2. You can also add the `--debug/-D` option in the Docker configuration file to start in debug mode.

How it works…

The preceding command would start the Docker in the daemon mode. You will see lots of useful messages as you start the daemon, such as loading up existing images, settings for firewalls (iptables), and so on. If you start a container, you will see messages like the following:

```
[info] POST /v1.15/containers/create
[99430521] +job create()
......
......
```

Building a Docker binary from the source

Sometimes it becomes necessary to build a Docker binary from the source for testing a patch. It is very easy to build the Docker binary from the source.

Getting ready

1. Download the Docker source code using `git`:

   ```
   $ git clone https://github.com/docker/docker.git
   ```

2. Install `make` on Fedora:

   ```
   $ yum install -y make
   ```

3. Make sure Docker is running on the host on which you are building the code and you can access it through the Docker client, as the build we are discussing here happens inside a container.

How to do it...

1. Go inside the cloned directory:

   ```
   $ cd docker
   ```

2. Run the `make` command:

   ```
   $ sudo make
   ```

How it works...

This will create a container and compile the code inside that from the master branch. Once finished, it will spit out the binary inside `bundles/<version>/binary`.

There's more...

- Similar to the source code, you can build the documentation as well:

  ```
  $ sudo make docs
  ```

- You can also run tests with the following command:

  ```
   $ sudo make test
  ```

See also

- Look at the documentation on the Docker website `https://docs.docker.com/contributing/devenvironment/`

Building images without using cached layers

By default, when we build an image, Docker will try to use the cached layers so that it takes less time to build. However, at times it is necessary to build from scratch. For example, you will need to force a system update such as `yum -y update`. Let's see how we can do that in this recipe.

Getting ready

Get a Dockerfile to build the image.

How to do it...

1. While building the image, pass the `--no-cache` option as follows:

```
$ docker build -t test --no-cache - < Dockerfile
```

How it works...

The `--no-cache` option will discard any cached layer and build one Dockerfile by following the instructions.

There's more...

Sometimes, we also want to discard the cache after only a few instructions. In such cases, we can add any arbitrary command which doesn't affect the image, such as creation or setting up an environment variable.

Building your own bridge for container communication

As we know, when the Docker daemon starts, it creates a bridge called `docker0` and all the containers would get the IP from it. Sometimes we might want to customize those settings. Let's see how we can do that in this recipe.

Getting ready

I am assuming you already have a Docker set up. On the Docker host, stop the Docker daemon. On Fedora, use the following command:

```
$ systemctl stop docker
```

How to do it...

1. To remove the default `docker0` bridge, use the following command:

   ```
   $ sudo ip link set dev docker0 down
   $ sudo brctl delbr docker0
   ```

2. To create the custom bridge, use the following command:

   ```
   $ sudo brctl addbr br0
   $ sudo ip addr add 192.168.2.1/24 dev br0
   $ sudo ip link set dev bridge0 up
   ```

3. Update the Docker configuration file to start with the bridge we created earlier. On Fedora, you can update the configuration file as follows:

   ```
   $ sed -i '/^OPTIONS/ s/$/ --bridge br0/' /etc/sysconfig/docker
   ```

4. To start the Docker daemon, use the following command:

   ```
   $ systemctl start docker
   ```

How it works...

The preceding steps will create a new bridge and it will assign the IP from 192.168.2.0 subnet to the containers.

There's more...

You can even add an interface to the bridge.

See also

- The documentation on the Docker website at `https://docs.docker.com/articles/networking/`

Changing the default execution driver of Docker

As we know, libcontainer is the default execution driver. There is legacy support for LXC userspace tools (`https://linuxcontainers.org/`). Keep in mind that LXC is not the primary development environment.

Getting ready

Install Docker on the system.

How to do it...

1. Start the Docker daemon with the `-e lxc` option, as follows:

```
$ docker -d -e lxc
```

You can also add this option in the configuration file of Docker, depending on the distribution.

How it works...

Docker uses LXC tools to access kernel features, such as Namespaces and Cgroups to run containers.

See also

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#docker-exec-driver-option`

Selecting the logging driver for containers

With the release of Docker 1.6, a new feature has been added to select the logging driver while starting the Docker daemon. Currently, three types of logging drivers are supported:

- none
- json-file (default)
- syslog

Getting ready

Install Docker 1.6 or above on the system.

How to do it...

1. Start the Docker daemon with the required logging driver as follows:

```
$ docker -d --log-driver=none
$ docker -d --log-driver=syslog
```

You can also add this option in the configuration file of Docker, depending on the distribution.

The `docker logs` command will just support the default logging driver JSON file.

How it works...

Depending on the log driver configuration, Docker daemon selects the corresponding logging driver.

There's more...

There is work in progress to add `journald` as one of the logging drivers. It will be available from Docker 1.7 at `http://www.projectatomic.io/blog/2015/04/logging-docker-container-output-to-journald/`.

See also

- The documentation on the Docker website `http://docs.docker.com/reference/run/#logging-drivers-log-driver`

Getting real-time Docker events for containers

As we will be running many containers in production, it will helpful if we can watch the real-time container events for monitoring and debugging purposes. Docker containers can report events, such as create, destroy, die, export, kill, oom, pause, restart, start, stop, and unpause. In this recipe, we will see how to enable event logging and then use filters to select specific event types, images or containers.

Getting ready

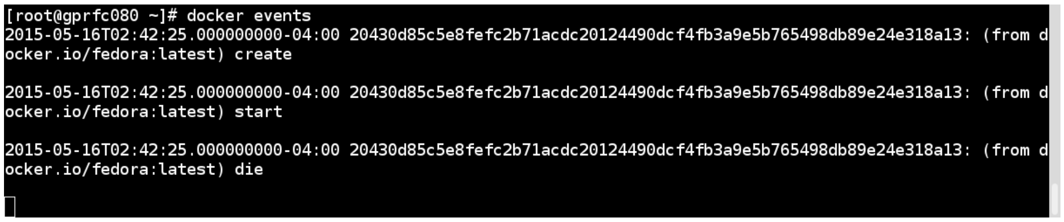
Make sure the Docker daemon is running on the host and you can connect through the Docker client.

How to do it...

1. Start the Docker events logging with the following command:

   ```
   $ docker events
   ```

2. From the other terminal, do some container/image-related operation and you will see a result similar to the following screenshot on the first terminal:



```
[root@gprfc080 ~]# docker events
2015-05-16T02:42:25.000000000-04:00 20430d85c5e8fefc2b71acdc20124490dcf4fb3a9e5b765498db89e24e318a13: (from d
ocker.io/fedora:latest) create

2015-05-16T02:42:25.000000000-04:00 20430d85c5e8fefc2b71acdc20124490dcf4fb3a9e5b765498db89e24e318a13: (from d
ocker.io/fedora:latest) start

2015-05-16T02:42:25.000000000-04:00 20430d85c5e8fefc2b71acdc20124490dcf4fb3a9e5b765498db89e24e318a13: (from d
ocker.io/fedora:latest) die
```

After the events collection started, I created a container to just echo something. As you can see from the preceding screenshot, a container got created, started, and died.

How it works...

With Docker events, Docker starts listing different events.

There's more...

- You can use the `--since` or `--until` option with Docker events to narrow down results for a selected timestamp:

  ```
  --since=""          Show all events created since timestamp
  --until=""          Stream events until this timestamp
  ```

  Consider the following example:

  ```
  $ docker events --since '2015-01-01'
  ```

- With filters, we can further narrow down the events log based on the event, container, and image as follows:

  - To list only the start event, use the following command:

    ```
    $ docker events --filter 'event=start'
    ```

  - To list events only from image CentOS, use the following command:

    ```
    $ docker events --filter 'image=docker.io/centos:centos7'
    ```

- To list events from the specific container, use the following command:

  ```
  docker events --filter
  'container=b3619441cb444b87b4d79a8c30616ca70da4b5aa8fdc5d8a4
  8d23a2082052174'
  ```

See also

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#events`

Module 3

Kubernetes Cookbook

Learn how to automate and manage your Linux containers and improve the overall performance of your system

1
Building Your Own Kubernetes

In this chapter, we will cover the following topics:

- Exploring architecture
- Preparing your environment
- Building datastore
- Creating an overlay network
- Configuring master
- Configuring nodes
- Running your first container in Kubernetes

Introduction

Welcome to the journey of Kubernetes! In this very first section, you will learn how to build your own Kubernetes cluster. Along with understanding each component and connecting them together, you will learn how to run your first container on Kubernetes. Holding a Kubernetes cluster will help you continue the study in the chapters ahead.

Exploring architecture

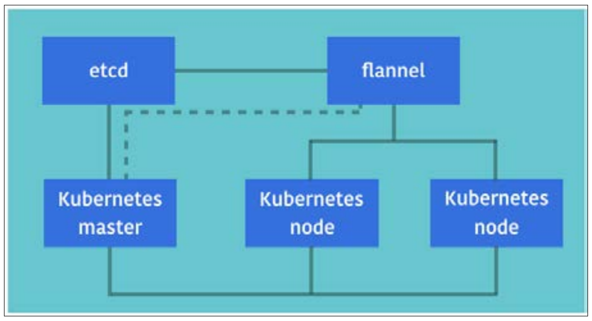
Kubernetes is an open source container management tool. It is a Go-Lang based (`https://golang.org`), lightweight, and portable application. You can set up a Kubernetes cluster on a Linux-based OS to deploy, manage, and scale the Docker container applications on multiple hosts.

Getting ready

Kubernetes is constructed using several components, as follows:

- Kubernetes master
- Kubernetes nodes
- etcd
- Overlay network (flannel)

These components are connected via network, as shown in the following screenshot:



The preceding image can be summarized as follows:

- **Kubernetes master** connects to **etcd** via HTTP or HTTPS to store the data. It also connects **flannel** to access the container application.
- Kubernetes nodes connect to the **Kubernetes master** via HTTP or HTTPS to get a command and report the status.
- Kubernetes nodes use an overlay network (for example, **flannel**) to make a connection of their container applications.

How to do it...

In this section, we are going to explain the features of Kubernetes master and nodes; both of them realize the main functions of the Kubernetes system.

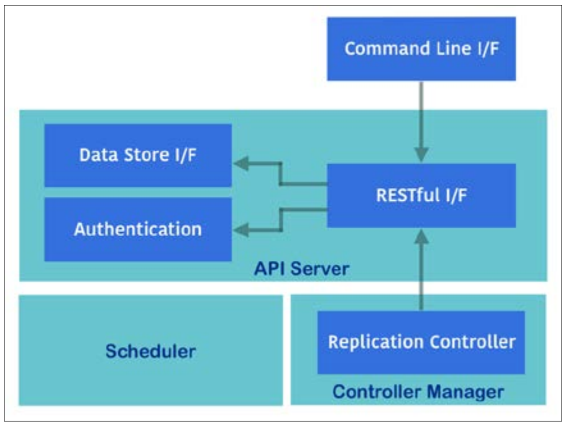
Kubernetes master

Kubernetes master is the main component of Kubernetes cluster. It serves several functionalities, such as the following items:

- Authorization and authentication
- RESTful API entry point

- Container deployment scheduler to the Kubernetes nodes
- Scaling and replicating the controller
- Read and store the configuration
- Command Line Interface

The next image shows how master daemons worked together to fulfill the mentioned functionalities:



There are several daemon processes that make the Kubernetes master's functionality, such as **kube-apiserver**, **kube-scheduler**, and **kube-controller-manager**. Hypercube wrapper launched all of them.

In addition, the Kubernetes Command Line Interface kubectl can control the Kubernetes master functionality.

API server (kube-apiserver)

The API server provides an HTTP- or HTTPS-based RESTful API, which is the hub between Kubernetes components, such as kubectl, scheduler, replication controller, etcd datastore, and kubelet and kube-proxy, which runs on Kubernetes nodes and so on.

Scheduler (kube-scheduler)

Scheduler helps to choose which container runs by which nodes. It is a simple algorithm that defines the priority to dispatch and bind containers to nodes, for example:

- CPU
- Memory
- How many containers are running?

Controller manager (kube-controller-manager)

Controller manager performs cluster operations. For example:

- Manages Kubernetes nodes
- Creates and updates the Kubernetes internal information
- Attempts to change the current status to the desired status

Command Line Interface (kubectl)

After you install Kubernetes master, you can use the Kubernetes Command Line Interface kubectl to control the Kubernetes cluster. For example, `kubectl get cs` returns the status of each component. Also, `kubectl get nodes` returns a list of Kubernetes nodes:

```
//see the ComponentStatuses
# kubectl get cs
NAME                 STATUS    MESSAGE              ERROR
controller-manager   Healthy   ok                   nil
scheduler            Healthy   ok                   nil
etcd-0               Healthy   {"health": "true"}   nil

//see the nodes
# kubectl get nodes
NAME          LABELS                             STATUS    AGE
kub-node1   kubernetes.io/hostname=kub-node1   Ready     26d
kub-node2   kubernetes.io/hostname=kub-node2   Ready     26d
```

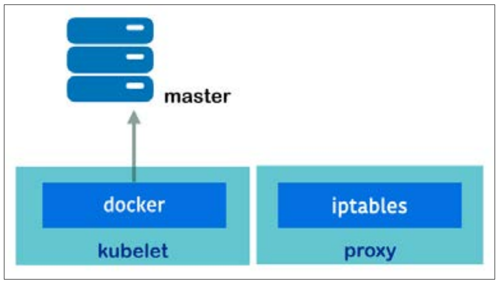
Kubernetes node

Kubernetes node is a slave node in the Kubernetes cluster. It is controlled by Kubernetes master to run the container application using Docker (`http://docker.com`) or rkt (`http://coreos.com/rkt/docs/latest/`) in this book; we will use the Docker container runtime as the default engine.

Node or slave?

The terminology of slave is used in the computer industry to represent the cluster worker node; however, it is also associated with discrimination. The Kubernetes project uses node instead.

The following image displays the role and tasks of daemon processes in node:



Node also has multiple daemon processes, named kubelet and kube-proxy, to support its functionalities.

kubelet

kubelet is the main process on Kubernetes node that communicates with Kubernetes master to handle the following operations:

- Periodically access the API Controller to check and report
- Perform container operations
- Runs the HTTP server to provide simple APIs

Proxy (kube-proxy)

Proxy handles the network proxy and load balancer for each container. It performs to change the Linux iptables rules (nat table) to control TCP and UDP packets across the containers.

After starting the `kube-proxy` daemon, it will configure iptables rules; you can see `iptables -t nat -L` or `iptables -t nat -S` to check the nat table rules, as follows:

```
 //the result will be vary and dynamically changed by kube-proxy
# sudo iptables -t nat -S
-P PREROUTING ACCEPT
-P INPUT ACCEPT
-P OUTPUT ACCEPT
-P POSTROUTING ACCEPT
-N DOCKER
-N FLANNEL
-N KUBE-NODEPORT-CONTAINER
-N KUBE-NODEPORT-HOST
-N KUBE-PORTALS-CONTAINER
-N KUBE-PORTALS-HOST
```

```
-A PREROUTING -m comment --comment "handle ClusterIPs; NOTE: this must be
before the NodePort rules" -j KUBE-PORTALS-CONTAINER
-A PREROUTING -m addrtype --dst-type LOCAL -m comment --comment "handle
service NodePorts; NOTE: this must be the last rule in the chain" -j
KUBE-NODEPORT-CONTAINER
-A PREROUTING -m addrtype --dst-type LOCAL -j DOCKER
-A OUTPUT -m comment --comment "handle ClusterIPs; NOTE: this must be
before the NodePort rules" -j KUBE-PORTALS-HOST
-A OUTPUT -m addrtype --dst-type LOCAL -m comment --comment "handle
service NodePorts; NOTE: this must be the last rule in the chain" -j
KUBE-NODEPORT-HOST
-A OUTPUT ! -d 127.0.0.0/8 -m addrtype --dst-type LOCAL -j DOCKER
-A POSTROUTING -s 192.168.90.0/24 ! -o docker0 -j MASQUERADE
-A POSTROUTING -s 192.168.0.0/16 -j FLANNEL
-A FLANNEL -d 192.168.0.0/16 -j ACCEPT
-A FLANNEL ! -d 224.0.0.0/4 -j MASQUERADE
```

How it works...

There are two more components to complement the Kubernetes nodes' functionalities, the datastore etcd and the overlay network flannel. You can learn how they support the Kubernetes system in the following paragraphs.

etcd

The etcd (`https://coreos.com/etcd/`) is the distributed key-value datastore. It can be accessed via the RESTful API to perform the CRUD operation over the network. Kubernetes uses etcd as the main datastore.

You can explore the Kubernetes configuration and status in etcd (`/registry`) using the `curl` command as follows:

```
//example: etcd server is 10.0.0.1 and default port is 2379
# curl -L "http://10.0.0.1:2379/v2/keys/registry"


{"action":"get","node":{"key":"/registry","dir":true,"nodes":[{"key":"/
registry/namespaces","dir":true,"modifiedIndex":15,"createdIndex":15},{"k
ey":"/registry/serviceaccounts","dir":true,"modifiedIndex":16,"createdInd
ex":16},{"key":"/registry/services","dir":true,"modifiedIndex":17,"create
dIndex":17},{"key":"/registry/ranges","dir":true,"modifiedIndex":76,"crea
tedIndex":76},{"key":"/registry/nodes","dir":true,"modifiedIndex":740,"cr
eatedIndex":740},{"key":"/registry/pods","dir":true,"modifiedIndex":794,"
createdIndex":794},{"key":"/registry/controllers","dir":true,"modifiedInd
ex":810,"createdIndex":810},{"key":"/registry/events","dir":true,"modifie
dIndex":6,"createdIndex":6}],"modifiedIndex":6,"createdIndex":6}}
```

Overlay network

Network communication between containers is the most difficult part. Because when you start to run the Docker, an IP address will be assigned dynamically; the container application needs to know the peer's IP address and port number.

If the container's network communication is only within the single host, you can use the Docker link to generate the environment variable to discover the peer. However, Kubernetes usually works as a cluster and ambassador pattern or overlay network could help to connect every node. Kubernetes uses overlay network to manage multiple containers' communication.

For overlay network, Kubernetes has several options, but using flannel is the easier solution.

Flannel

Flannel also uses etcd to configure the settings and store the status. You can also perform the `curl` command to explore the configuration (`/coreos.com/network`) and status, as follows:

```
//overlay network CIDR is 192.168.0.0/16
# curl -L "http://10.0.0.1:2379/v2/keys/coreos.com/network/config"

{"action":"get","node":{"key":"/coreos.com/network/config","value":"{
\"Network\": \"192.168.0.0/16\" }","modifiedIndex":144913,"createdInd
ex":144913}}

//Kubernetes assigns some subnets to containers
# curl -L "http://10.0.0.1:2379/v2/keys/coreos.com/network/subnets"

{"action":"get","node":{"key":"/coreos.com/network/subnets","dir":true,"n
odes":[{"key":"/coreos.com/network/subnets/192.168.90.0-24","value":"{\"
PublicIP\":\"10.97.217.158\"}","expiration":"2015-11-05T08:16:21.9957499
71Z","ttl":38993,"modifiedIndex":388599,"createdIndex":388599},{"key":"/
coreos.com/network/subnets/192.168.76.0-24","value":"{\"PublicIP\":\"10.9
7.217.148\"}","expiration":"2015-11-05T04:32:45.528111606Z","ttl":25576,"
modifiedIndex":385909,"createdIndex":385909},{"key":"/coreos.com/network/
subnets/192.168.40.0-24","value":"{\"PublicIP\":\"10.97.217.51\"}","expir
ation":"2015-11-05T15:18:27.335916533Z","ttl":64318,"modifiedIndex":3936-
75,"createdIndex":393675}],"modifiedIndex":79,"createdIndex":79}}
```

See also

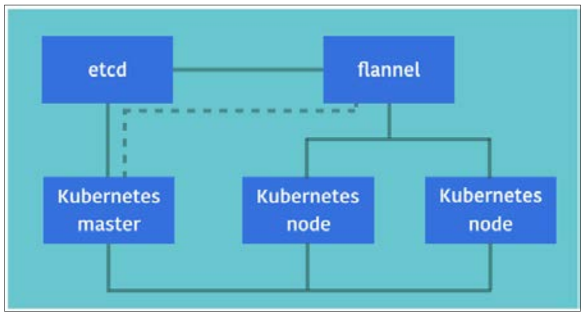
This section describes the basic architecture and methodology of Kubernetes and related components. Understanding Kubernetes is not easy, but a step-by-step lesson on how to setup, configure, and manage Kubernetes is really fun.

The following recipes describe how to install and configure related components:

- *Building datastore*
- *Creating an overlay network*
- *Configuring master*
- *Configuring nodes*

Preparing your environment

Before heading to the journey of building our own cluster, we have to prepare the environment in order to build the following components:



There are different solutions of creating such a Kubernetes cluster, for example:

- Local-machine solutions that include:
    - Docker-based
    - Vagrant
    - Linux machine
- Hosted solution that includes:
    - Google Container Engine
- Custom solutions

A local-machine solution is suitable if we just want to build a development environment or do the proof of concept quickly. By using **Docker** (`https://www.docker.com`) or **Vagrant** (`https://www.vagrantup.com`), we could easily build the desired environment in one single machine; however, it is not practical if we want to build a production environment. A hosted solution is the easiest starting point if we want to build it in the cloud.

Google Container Engine, which has been used by Google for many years, has the comprehensive support naturally and we do not need to care much about the installation and setting. Kubernetes can also run on different cloud and on-premises VMs by custom solutions. We will build the Kubernetes clusters from scratch on Linux-based virtual machines (CentOS 7.1) in the following chapters. The solution is suitable for any Linux machines in both cloud and on-premises environments.

Getting ready

It is recommended if you have at least four Linux servers for master, etcd, and two nodes. If you want to build it as a high availability cluster, more servers for each component are preferred. We will build three types of servers in the following sections:

- Kubernetes master
- Kubernetes node
- etcd

Flannel will not be located in one machine, which is required in all the nodes. Communication between containers and services are powered by flannel, which is an etcd backend overlay network for containers.

Hardware resource

The hardware spec of each component is suggested in the following table. Please note that it might cause a longer response time when manipulating the cluster if the amount of requests between the API server and etcd is large. In a normal situation, increasing resources can resolve this problem:

Component Spec	**Kubernetes master**	**etcd**
CPU Count	1	1
Memory GB	2G	2G

For the nodes, the default maximum number of pods in one node is 40. However, a node capacity is configurable when adding a node. You have to measure how many resources you might need for the hosted services and applications to decide how many nodes should be there with a certain spec and with proper monitoring in production workload.

Check out your node capacity in node

In your master, you could install `jq` by `yum install jq` and use `kubectl get nodes -o json | jq '.items[] | {name: .metadata.name, capacity: .status.capacity}'` to check the capacity of each node, including CPU, memory, and the maximum capacity of pods:

```
// check out your node capacity
$ kubectl get nodes -o json | jq '.items[] | {name:
.metadata.name, capacity: .status.capacity}'
{
  "name": "kub-node1",
  "capacity": {
    "cpu": "1",
    "memory": "1021536Ki",
    "pods": "40"
  }
}
{
  "name": "kub-node2",
  "capacity": {
    "cpu": "1",
    "memory": "1021536Ki",
    "pods": "40"
  }
}
```

Operating system

The OS of nodes could be various, but the kernel version must be 3.10 or later. Following are the OSs that are using kernel 3.10+:

- CentOS 7 or later
- RHEL 7 or later
- Ubuntu Vivid 15.04 / Ubuntu Trusty 14.04 (LTS) / Ubuntu Saucy 13.10

Beware of the Linux kernel version

Docker requires that your kernel must be 3.10 at minimum on CentOS or Red Hat Enterprise Linux, 3.13 kernel version on Ubuntu Precise 12.04 (LTS). It will cause data loss or kernel panic sometimes if using unsupported kernels. It is recommended you fully update the system before building Kubernetes. You can use `uname -r` to check the kernel you're currently using. For more information on checking the kernel version, please refer to `http://www.linfo.org/find_kernel_version.html`.

How to do it…

To ensure each component works perfectly in Kubernetes cluster, we must install the correct packages on each machine of master, node, and etcd.

Kubernetes master

Kubernetes master should be installed on a Linux-based OS. For the examples listed in this book, we will use CentOS 7.1 as an OS. There are two packages required in master:

- Kubernetes
- Flannel (optional)
- iptables (at least 1.4.11+ is preferred)

Kubernetes (`https://github.com/kubernetes/kubernetes/releases`) has a couple of fast-paced releases. Flannel daemon is optional in master; if you would like to launch Kubernetes UI, flannel (`https://github.com/coreos/flannel/releases`) is required. Otherwise, Kubernetes UI will be failed to access via `https://<kubernetes-master>/ui`.

Beware of iptables version

Kubernetes uses iptables to implement service proxy. iptables with version 1.4.11+ is recommended on Kubernetes. Otherwise, iptables rules might be out of control and keep increasing. You can use `yum info iptables` to check the current version of iptables.

Kubernetes nodes

On Kubernetes nodes, we have to prepare the following:

- Kubernetes
- Flannel daemon
- Docker (at least 1.6.2+ is preferred)
- iptables (at least 1.4.11+ is preferred)

Beware of Docker version and dependencies

Sometimes, you'll get an unknown error when using the incompatible Docker version, such as target image is not found. You can always use the `docker version` command to check the current version you've installed. The recommended versions we tested are at least 1.7.1+. Before building the cluster, you can start the service by using the `service docker start` command and make sure it can be contacted using `docker ps`.

Docker has different package names and dependency packages in Linux distributions. In Ubuntu, you could use `curl -sSL https://get.docker.com/ | sh`. For more information, check out the Docker installation document (`http://docs.docker.com/v1.8/installation`) to find your preferred Linux OS.

etcd

etcd, which is a distributed reliable key-value store for shared configurations and service discovery, is powered by CoreOS. The release page is `https://github.com/coreos/etcd/releases`. The prerequisite we need is just the etcd package.

See also

After preparing the environment, it is time to build up your Kubernetes. Check out the following recipes for that:

- *Building datastore*
- *Creating an overlay network*
- *Configuring master*
- *Configuring nodes*
- The *Setting resource in nodes* recipe in *Chapter 7*, *Advanced Cluster Administration*
- The *Monitoring master and node* recipe in *Chapter 8*, *Logging and Monitoring*

Building datastore

In order to persist the Kubernetes cluster information, we need to set up datastore. Kubernetes uses etcd as a standard datastore. This section will guide you to build the etcd server.

How to do it...

The etcd database requires Linux OS; some Linux distributions provide the etcd package and some don't. This section describes how to install etcd.

Red Hat Enterprise Linux 7 or CentOS 7

Red Hat Enterprise Linux (RHEL) 7, CentOS 7 or later has an official package for etcd. You can install via the `yum` command, as follows:

```
//it will perform to install etcd package on RHEL/CentOS Linux
sudo yum update -y
sudo yum install etcd
```

Ubuntu Linux 15.10 Wily Werewolf

Ubuntu 15.10 or later has an official package for etcd as well. You can install via the `apt-get` command as follows:

```
//it will perform to install etcd package on Ubuntu Linux
sudo apt-get update -y
sudo apt-get install etcd
```

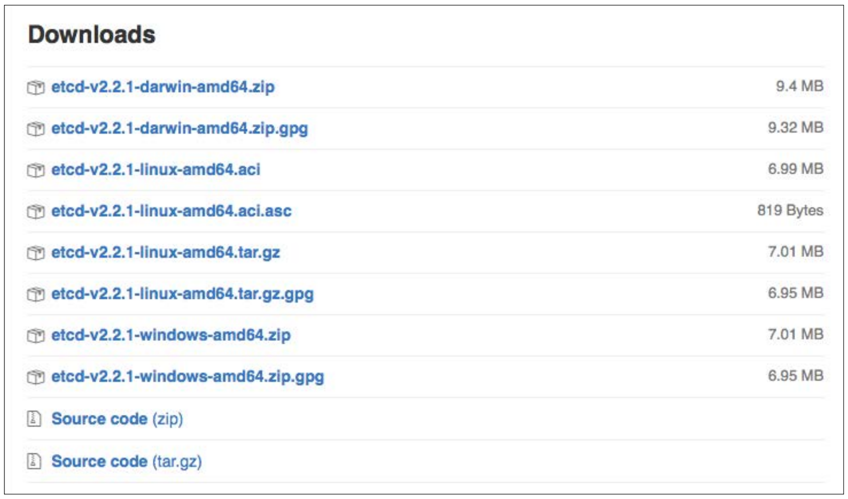
Other Linux

If you are using a different Linux version, such as Amazon Linux, you can download a binary from the official website and install it as follows.

Download a binary

etcd is provided via `https://github.com/coreos/etcd/releases`. OS X (darwin-amd64), Linux, Windows binary, and source code are available for download.

> Note that there are no 32-bit binaries provided due to the Go runtime issue. You must prepare a 64-bit Linux OS.

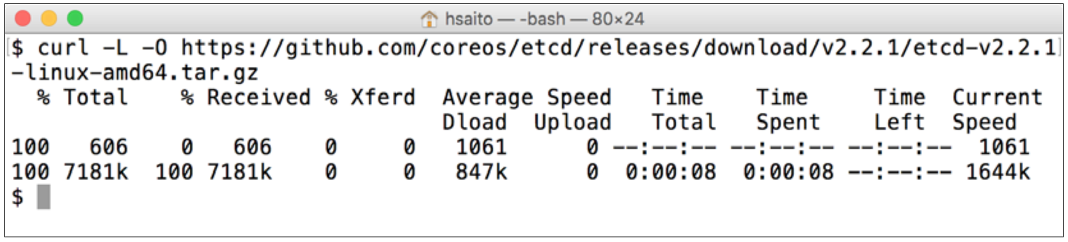


Downloads

etcd-v2.2.1-darwin-amd64.zip	9.4 MB
etcd-v2.2.1-darwin-amd64.zip.gpg	9.32 MB
etcd-v2.2.1-linux-amd64.aci	6.99 MB
etcd-v2.2.1-linux-amd64.aci.asc	819 Bytes
etcd-v2.2.1-linux-amd64.tar.gz	7.01 MB
etcd-v2.2.1-linux-amd64.tar.gz.gpg	6.95 MB
etcd-v2.2.1-windows-amd64.zip	7.01 MB
etcd-v2.2.1-windows-amd64.zip.gpg	6.95 MB
Source code (zip)	
Source code (tar.gz)	

On your Linux machine, use the `curl` command to download the `etcd-v2.2.1-linux-amd64.tar.gz` binary:

```
// follow redirection(-L) and use remote name (-O)
curl -L -O https://github.com/coreos/etcd/releases/download/v2.2.1/etcd-v2.2.1-linux-amd64.tar.gz
```



```
$ curl -L -O https://github.com/coreos/etcd/releases/download/v2.2.1/etcd-v2.2.1
-linux-amd64.tar.gz
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   606    0   606    0     0   1061      0 --:--:-- --:--:-- --:--:--  1061
100 7181k  100 7181k    0     0   847k      0  0:00:08  0:00:08 --:--:-- 1644k
$ 
```

Creating a user

Due to security reasons, create a local user and group that can own etcd packages:

1. Run the following `useradd` command:

   ```
   //options
   //    create group(-U), home directory(-d), and create it(-m)
   //    name in GCOS field (-c), login shell(-s)
   $ sudo useradd -U -d /var/lib/etcd -m -c "etcd user" -s /sbin/
   nologin etcd
   ```

2. You can check `/etc/passwd` to see whether creating `etcd user` has created a user or not:

   ```
   //search etcd user on /etc/passwd, uid and gid is vary
   $ grep etcd /etc/passwd
   etcd:x:997:995:etcd user:/var/lib/etcd:/sbin/nologin
   ```

[You can delete a user any time; type `sudo userdel -r etcd` to delete `etcd user`.]

Install etcd

1. After downloading an etcd binary, use the `tar` command to extract files:

   ```
   $ tar xf etcd-v2.2.1-linux-amd64.tar.gz
   $ cd etcd-v2.2.1-linux-amd64

   //use ls command to see that there are documentation and binaries
   $ ls
   Documentation  README-etcdctl.md  README.md  etcd  etcdctl
   ```

2. There are `etcd` daemon and `etcdctl` command that need to be copied to `/usr/local/bin`. Also, create `/etc/etcd/etcd.conf` as a setting file:

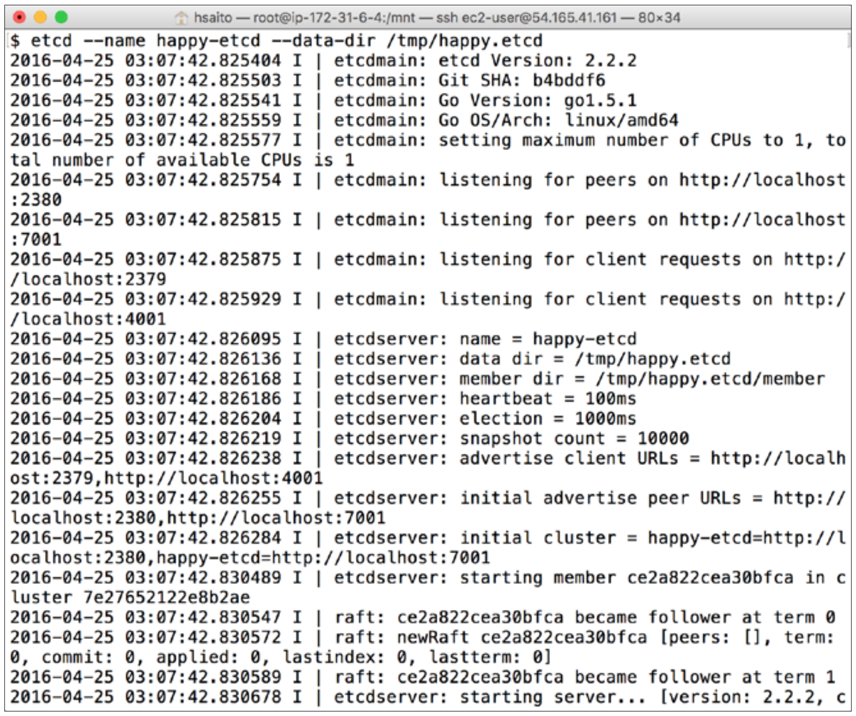
   ```
   $ sudo cp etcd etcdctl /usr/local/bin/

   //create etcd.conf
   $ sudo mkdir -p /etc/etcd/
   $ sudo touch /etc/etcd/etcd.conf
   $ sudo chown -R etcd:etcd /etc/etcd
   ```

How it works...

Let's test run the `etcd` daemon to explorer the etcd functionalities. Type the `etcd` command with the `name` and `data-dir` argument as follows:

```
//for the testing purpose, create data file under /tmp
$ etcd --name happy-etcd --data-dir /tmp/happy.etcd &
```

Then, you will see several output logs as follows:



hsaito — root@ip-172-31-6-4:/mnt — ssh ec2-user@54.165.41.161 — 80×34

```
$ etcd --name happy-etcd --data-dir /tmp/happy.etcd
2016-04-25 03:07:42.825404 I | etcdmain: etcd Version: 2.2.2
2016-04-25 03:07:42.825503 I | etcdmain: Git SHA: b4bddf6
2016-04-25 03:07:42.825541 I | etcdmain: Go Version: go1.5.1
2016-04-25 03:07:42.825559 I | etcdmain: Go OS/Arch: linux/amd64
2016-04-25 03:07:42.825577 I | etcdmain: setting maximum number of CPUs to 1, to
tal number of available CPUs is 1
2016-04-25 03:07:42.825754 I | etcdmain: listening for peers on http://localhost
:2380
2016-04-25 03:07:42.825815 I | etcdmain: listening for peers on http://localhost
:7001
2016-04-25 03:07:42.825875 I | etcdmain: listening for client requests on http:/
/localhost:2379
2016-04-25 03:07:42.825929 I | etcdmain: listening for client requests on http:/
/localhost:4001
2016-04-25 03:07:42.826095 I | etcdserver: name = happy-etcd
2016-04-25 03:07:42.826136 I | etcdserver: data dir = /tmp/happy.etcd
2016-04-25 03:07:42.826168 I | etcdserver: member dir = /tmp/happy.etcd/member
2016-04-25 03:07:42.826186 I | etcdserver: heartbeat = 100ms
2016-04-25 03:07:42.826204 I | etcdserver: election = 1000ms
2016-04-25 03:07:42.826219 I | etcdserver: snapshot count = 10000
2016-04-25 03:07:42.826238 I | etcdserver: advertise client URLs = http://localh
ost:2379,http://localhost:4001
2016-04-25 03:07:42.826255 I | etcdserver: initial advertise peer URLs = http://
localhost:2380,http://localhost:7001
2016-04-25 03:07:42.826284 I | etcdserver: initial cluster = happy-etcd=http://l
ocalhost:2380,happy-etcd=http://localhost:7001
2016-04-25 03:07:42.830489 I | etcdserver: starting member ce2a822cea30bfca in c
luster 7e27652122e8b2ae
2016-04-25 03:07:42.830547 I | raft: ce2a822cea30bfca became follower at term 0
2016-04-25 03:07:42.830572 I | raft: newRaft ce2a822cea30bfca [peers: [], term:
0, commit: 0, applied: 0, lastindex: 0, lastterm: 0]
2016-04-25 03:07:42.830589 I | raft: ce2a822cea30bfca became follower at term 1
2016-04-25 03:07:42.830678 I | etcdserver: starting server... [version: 2.2.2, c
```

Now, you can try to use the `etcdctl` command to access etcd and to load and store the data as follows:

```
//set value "hello world" to the key /my/happy/data
$ etcdctl set /my/happy/data "hello world"


//get value for key /my/happy/data
$ etcdctl get /my/happy/data
hello world
```

In addition, by default, etcd opens TCP port `2379` to access the RESTful API, so you may also try to use an HTTP client, such as the `curl` command to access data as follows:

```
//get value for key /my/happy/data using cURL
$ curl -L http://localhost:2379/v2/keys/my/happy/data
{"action":"get","node":{"key":"/my/happy/data","value":"hello world","mod
ifiedIndex":4,"createdIndex":4}}

//set value "My Happy world" to the key /my/happy/data using cURL
$ curl http://127.0.0.1:2379/v2/keys/my/happy/data -XPUT -d value="My
Happy world"

//get value for key /my/happy/data using etcdctl
$ etcdctl get /my/happy/data
My Happy world
```

Okay! Now, you can delete the key using the `curl` command as follows:

```
$ curl http://127.0.0.1:2379/v2/keys/my?recursive=true -XDELETE

//no more data returned afterword
$ curl http://127.0.0.1:2379/v2/keys/my/happy/data
{"errorCode":100,"message":"Key not found","cause":"/my","index":10}

$ curl http://127.0.0.1:2379/v2/keys/my/happy
{"errorCode":100,"message":"Key not found","cause":"/my","index":10}

$ curl http://127.0.0.1:2379/v2/keys/my
{"errorCode":100,"message":"Key not found","cause":"/my","index":10}
```

Auto startup script

Based on your Linux, either systemd or init, there are different ways to make an auto startup script.

If you are not sure, check the process ID `1` on your system. Type `ps -P 1` to see the process name as follows:

```
//This Linux is systemd based
$ ps -P 1
  PID PSR TTY      STAT   TIME COMMAND
```

```
    1    0 ?         Ss     0:03 /usr/lib/systemd/systemd --switched-root -
system

//This Linux is init based

# ps -P 1

  PID PSR TTY        STAT   TIME COMMAND

    1    0 ?         Ss     0:01 /sbin/init
```

Startup script (systemd)

If you are using systemd-based Linux, such as RHEL 7, CentOS 7, Ubuntu 15.4 or later, you need to prepare the `/usr/lib/systemd/system/etcd.service` file as follows:

```
[Unit]
Description=Etcd Server
After=network.target

[Service]
Type=simple
WorkingDirectory=/var/lib/etcd/
EnvironmentFile=/etc/etcd/etcd.conf
User=etcd
ExecStart=/usr/local/bin/etcd

[Install]
WantedBy=multi-user.target
```

After that, register to `systemd` using the `systemctl` command as follows:

```
# sudo systemctl enable etcd
```

Then, you restart the system or type `sudo systemctl start etcd` to launch the `etcd` daemon. You may check the etcd service status using `sudo systemctl status -l etcd`.

Startup script (init)

If you are using the init-based Linux, such as Amazon Linux, use the traditional way to prepare the `/etc/init.d/etcd` script as follows:

```
#!/bin/bash
#
# etcd This shell script takes care of starting and stopping etcd
#
# chkconfig: - 60 74
# description: etcd

### BEGIN INIT INFO
# Provides: etcd
# Required-Start: $network $local_fs $remote_fs
# Required-Stop: $network $local_fs $remote_fs
# Should-Start: $syslog $named ntpdate
# Should-Stop: $syslog $named
# Short-Description: start and stop etcd
# Description: etcd
### END INIT INFO

# Source function library.
. /etc/init.d/functions

# Source networking configuration.
. /etc/sysconfig/network

prog=/usr/local/bin/etcd
etcd_conf=/etc/etcd/etcd.conf
lockfile=/var/lock/subsys/`basename $prog`
hostname=`hostname`

start() {
  # Start daemon.
. $etcd_conf
  echo -n $"Starting $prog: "
  daemon --user=etcd $prog > /var/log/etcd.log 2>&1 &
  RETVAL=$?
  echo
  [ $RETVAL -eq 0 ] && touch $lockfile
  return $RETVAL
}
stop() {
  [ "$EUID" != "0" ] && exit 4
        echo -n $"Shutting down $prog: "
  killproc $prog
  RETVAL=$?
        echo
  [ $RETVAL -eq 0 ] && rm -f $lockfile
  return $RETVAL
}

# See how we were called.
case "$1" in
  start)
  start
  ;;
```

```
      stop)
      stop
      ;;
      status)
      status $prog
      ;;
      restart)
      stop
      start
      ;;
      reload)
      exit 3
      ;;
      *)
      echo $"Usage: $0 {start|stop|status|restart|reload}"
      exit 2
    esac
```

After that, register to init script using the `chkconfig` command as follows:

```
//set file permission correctly
$ sudo chmod 755 /etc/init.d/etcd
$ sudo chown root:root /etc/init.d/etcd

//auto start when boot Linux
$ sudo chkconfig --add etcd
$ sudo chkconfig etcd on
```

Then, you restart the system or type `/etc/init.d/etcd start` to launch the `etcd` daemon.

Configuration

There is the file `/etc/etcd/etcd.conf` to change the configuration of etcd, such as data file path and TCP port number.

The minimal configuration is as follows:

NAME	Mean	Example	Note
`ETCD_NAME`	Instance name	`myhappy-etcd`	
`ETCD_DATA_DIR`	Data file path	`/var/lib/etcd/myhappy.etcd`	File path must be owned by etcd user

NAME	Mean	Example	Note
`ETCD_LISTEN_CLIENT_URLS`	TCP port number	`http://0.0.0.0:8080`	Specifying `0.0.0.0`, binds all IP address, otherwise use localhost to accept only same machine
`ETCD_ADVERTISE_CLIENT_URLS`	Advertise this etcd URL to other cluster instances	`http://localhost:8080`	Use for clustering configuration

Note that you need to use the `export` directive if you want to use the init-based Linux in order to set environment variables as follows:

```
$ cat /etc/etcd/etcd.conf


export ETCD_NAME=myhappy-etcd
export ETCD_DATA_DIR="/var/lib/etcd/myhappy.etcd"
export ETCD_LISTEN_CLIENT_URLS="http://0.0.0.0:8080"
export ETCD_ADVERTISE_CLIENT_URLS="http://localhost:8080"
```

On the other hand, systemd-based Linux doesn't need the export directive as follows:

```
$ cat /etc/etcd/etcd.conf
ETCD_NAME=myhappy-etcd
ETCD_DATA_DIR="/var/lib/etcd/myhappy.etcd"
ETCD_LISTEN_CLIENT_URLS="http://0.0.0.0:8080"
ETCD_ADVERTISE_CLIENT_URLS="http://localhost:8080"
```

See also

This section described how to configure etcd. It is easy and simple to operate via the RESTful API, but powerful. However, there's a need to be aware of its security and availability. The following recipes will describe how to ensure that etcd is secure and robust:

- *Exploring architecture*
- The *Clustering etcd* recipe in *Chapter 4*, *Building a High Availability Cluster*
- The *Authentication and authorization* recipe in *Chapter 7*, *Advanced Cluster Administration*
- The *Working with etcd log* recipe in *Chapter 8*, *Logging and Monitoring*

Creating an overlay network

Kubernetes abstracts the networking to enable communication between containers across nodes. The basic unit to make it possible is named pod, which is the smallest deployment unit in Kubernetes with a shared context in a containerized environment. Containers within a pod can communicate with others by port with the localhost. Kubernetes will deploy the pods across the nodes.

Then, how do pods talk to each other?

Kubernetes allocates each pod an IP address in a shared networking namespace so that pods can communicate with other pods across the network. There are a couple of ways to achieve the implementation. The easiest and across the platform way will be using flannel.

Flannel gives each host an IP subnet, which can be accepted by Docker and allocate the IPs to individual containers. Flannel uses etcd to store the IP mapping information, and has a couple of backend choices for forwarding the packets. The easiest backend choice would be using TUN device to encapsulate IP fragment in a UDP packet. The port is `8285` by default.

Flannel also supports in-kernel VXLAN as backend to encapsulate the packets. It might provide better performance than UDP backend while it is not running in user space. Another popular choice is using the advanced routing rule upon Google Cloud Engine (`https://cloud.google.com/compute/docs/networking#routing`). We'll use both UDP and VXLAN as examples in this section.

Flanneld is the agent of flannel used to watch the information from etcd, allocate the subnet lease on each host, and route the packets. What we will do in this section is let flanneld be up and running and allocate a subnet for each host.

If you're struggling to find out which backend should be used, here is a simple performance test between UDP and VXLAN. We use qperf (`http://linux.die.net/man/1/qperf`) to measure packet transfer performance between containers. TCP streaming one way bandwidth through UDP is 0.3x slower than VXLAN when there are some loads on the hosts. If you prefer building Kubernetes on the cloud, GCP is the easiest choice.

Getting ready

Before installing flannel, be sure you have the etcd endpoint. Flannel needs etcd as its datastore. If Docker is running, stop the Docker service first and delete `docker0`, which is a virtual bridge created by Docker:

```
# Stop docker service
$ service docker stop
```

```
# delete docker0
$ ip link delete docker0
```

Installation

Using the `etcdctl` command we learned in the previous section on the etcd instance, insert the desired configuration into etcd with the key `/coreos.com/network/config`:

Configuration Key	Description
`Network`	IPv4 network for flannel to allocate to entire virtual network
`SubnetLen`	The subnet prefix length to each host, default is `24`.
`SubnetMin`	The beginning of IP range for flannel subnet allocation
`SubnetMax`	The end of IP range for flannel subnet allocation
`Backend`	Backend choices for forwarding the packets. Default is `udp`.

```
# insert desired CIDR for the overlay network Flannel creates
$ etcdctl set /coreos.com/network/config '{ "Network": "192.168.0.0/16"
}'
```

Flannel will assign the IP address within `192.168.0.0/16` for overlay network with `/24` for each host by default, but you could also overwrite its default setting and insert into etcd:

```
$ cat flannel-config-udp.json
{
    "Network": "192.168.0.0/16",
    "SubnetLen": 28,
    "SubnetMin": "192.168.10.0",
    "SubnetMax": "192.168.99.0",
    "Backend": {
        "Type": "udp",
        "Port": 7890
    }
}
```

Use the `etcdctl` command to insert the `flannel-config-udp.json` configuration:

```
# insert the key by json file
$ etcdctl set /coreos.com/network/config < flannel-config-udp.json
```

Then, flannel will allocate to each host with `/28` subnet and only issue the subnets within `192.168.10.0` and `192.168.99.0`. Backend will be still `udp` and the default port will be changed from `8285` to `7890`.

We could also use VXLAN to encapsulate the packets and use `etcdctl` to insert the configuration:

```
$ cat flannel-config-vxlan.json
{
    "Network": "192.168.0.0/16",
    "SubnetLen": 24,
    "Backend": {
        "Type": "vxlan",
        "VNI": 1
    }
}

# insert the key by json file
$ etcdctl set /coreos.com/network/config < flannel-config-vxlan.json
```

You might be able to see the configuration you get using `etcdctl`:

```
$ etcdctl get /coreos.com/network/config
{
    "Network": "192.168.0.0/16",
    "SubnetLen": 24,
    "Backend": {
        "Type": "vxlan",
        "VNI": 1
     }
}
```

CentOS 7 or Red Hat Enterprise Linux 7

RHEL 7, CentOS 7, or later have an official package for flannel. You can install them via the `yum` command:

```
# install flannel package
$ sudo yum install flannel
```

After the installation, we have to configure the etcd server in order to use the flannel service:

```
$ cat /etc/sysconfig/flanneld

# Flanneld configuration options

# etcd url location.  Point this to the server where etcd runs
FLANNEL_ETCD="<your etcd server>"

# etcd config key.  This is the configuration key that flannel queries
# For address range assignment
FLANNEL_ETCD_KEY="/coreos.com/network"

# Any additional options that you want to pass
#FLANNEL_OPTIONS=""
```

We should always keep `flanneld` up and running all the time when we boot up the server. Using `systemctl` could do the trick:

```
# Enable flanneld service by default
$ sudo systemctl enable flanneld

# start flanneld
$ sudo service flanneld start

# check if the service is running
$ sudo service flannel status
```

Other Linux options

You can always download a binary as an alternative. The CoreOS flannel official release page is here: `https://github.com/coreos/flannel/releases`. Choose the packages with the **Latest release** tag; it will always include the latest bug fixes:

```
# download flannel package
$ curl -L -O https://github.com/coreos/flannel/releases/download/v0.5.5/
flannel-0.5.5-linux-amd64.tar.gz

# extract the package
$ tar zxvf flannel-0.5.5-linux-amd64.tar.gz

# copy flanneld to $PATH
$ sudo cp flannel-0.5.5/flanneld /usr/local/bin
```

If you use a startup script (`systemd`) in the etcd section, you might probably choose the same way to describe `flanneld`:

```
$ cat /usr/lib/systemd/system/flanneld.service
[Unit]
Description=Flanneld overlay address etcd agent
Wants=etcd.service
After=etcd.service
Before=docker.service

[Service]
Type=notify
EnvironmentFile=/etc/sysconfig/flanneld
EnvironmentFile=-/etc/sysconfig/docker-network
ExecStart=/usr/bin/flanneld -etcd-endpoints=${FLANNEL_ETCD} -etcd-prefix=${FLANNEL_ETCD_KEY} $FLANNEL_OPTIONS
Restart=on-failure

RestartSec=5s

[Install]
WantedBy=multi-user.target
```

Then, enable the service on bootup using `sudo systemctl enable flanneld`.

Alternatively, you could use a startup script (init) under `/etc/init.d/flanneld` if you're using an init-based Linux:

```
#!/bin/bash

# flanneld  This shell script takes care of starting and stopping flanneld
#

# Source function library.
. /etc/init.d/functions

# Source networking configuration.
. /etc/sysconfig/network

prog=/usr/local/bin/flanneld
lockfile=/var/lock/subsys/`basename $prog`
```

After you have sourced and set the variables, you should implement start, stop status, and restart for the service. The only thing you need to take care of is to ensure to add the `etcd` endpoint into the configuration when the daemon starts:

```
start() {
  # Start daemon.
  echo -n $"Starting $prog: "
  daemon $prog \
    --etcd-endpoints=="<your etcd server>" \
    -ip-masq=true \
    > /var/log/flanneld.log 2>&1 &
  RETVAL=$?
  echo
  [ $RETVAL -eq 0 ] && touch $lockfile
  return $RETVAL
}

stop() {
  [ "$EUID" != "0" ] && exit 4
        echo -n $"Shutting down $prog: "
  killproc $prog
  RETVAL=$?
        echo
  [ $RETVAL -eq 0 ] && rm -f $lockfile
  return $RETVAL
}

case "$1" in
  start)
  start
  ;;
  stop)
  stop
  ;;
  status)
  status $prog
  ;;
  restart|force-reload)
  stop
  start
  ;;
  try-restart|condrestart)
  if status $prog > /dev/null; then
      stop
```

```
        start
    fi
    ;;
    reload)
    exit 3
    ;;
    *)
    echo $"Usage: $0 {start|stop|status|restart|try-restart|force-
  reload}"
    exit 2
  esac
```

If flannel gets stuck when starting up

Check out your etcd endpoint is accessible and the key listed in `FLANNEL_ETCD_KEY` exists:

```
# FLANNEL_ETCD_KEY="/coreos.com/network/config"
$ curl -L http://<etcd endpoint>:2379/v2/keys/coreos.com/
network/config
```

You could also check out flannel logs using `sudo journalctl -u flanneld`.

After the flannel service starts, you should be able to see a file in `/run/flannel/subnet.env` and the `flannel0` bridge in `ifconfig`.

How to do it...

To ensure flannel works well and transmits the packets from the Docker virtual interface, we need to integrate it with Docker.

Flannel networking configuration

1. After `flanneld` is up and running, use the `ifconfig` or `ip` commands to see whether there is a `flannel0` virtual bridge in the interface:

```
# check current ipv4 range
$ ip a | grep flannel | grep inet
    inet 192.168.50.0/16 scope global flannel0
```

   We can see from the preceding example, the subnet lease of `flannel0` is `192.168.50.0/16`.

2. Whenever your `flanneld` service starts, flannel will acquire the subnet lease and save in etcd and then write out the environment variable file in `/run/flannel/subnet.env` by default, or you could change the default path using the `--subnet-file` parameter when launching it:

```
# check out flannel subnet configuration on this host
$ cat /run/flannel/subnet.env
FLANNEL_SUBNET=192.168.50.1/24
FLANNEL_MTU=1472
FLANNEL_IPMASQ=true
```

Integrating with Docker

There are a couple of parameters that are supported by the Docker daemon. In `/run/flannel/subnet.env`, flannel already allocated one subnet with the suggested MTU and IPMASQ settings. The corresponding parameters in Docker are:

Parameters	Meaning
`--bip=""`	Specify network bridge IP (`docker0`)
`--mtu=0`	Set the container network MTU (for `docker0` and `veth`)
`--ip-masq=true`	(Optional) Enable IP masquerading

1. We could use the variables listed in `/run/flannel/subnet.env` into the Docker daemon:

```
# import the environment variables from subnet.env
$ . /run/flannel/subnet.env


# launch docker daemon with flannel information
$ docker -d --bip=${FLANNEL_SUBNET} --mtu=${FLANNEL_MTU}
# Or if your docker version is 1.8 or higher, use subcommand
daemon instead
$ docker daemon --bip=${FLANNEL_SUBNET} --mtu=${FLANNEL_MTU}
```

2. Alternatively, you can also specify them into `OPTIONS` of `/etc/sysconfig/docker`, which is the Docker configuration file in CentOS:

```
### in the file - /etc/sysconfig/docker
# set the variables into OPTIONS
$ OPTIONS="--bip=${FLANNEL_SUBNET} --mtu=${FLANNEL_MTU} --ip-
masq=${FLANNEL_IPMASQ}"
```

In the preceding example, specify `${FLANNEL_SUBNET}` is replaced by `192.168.50.1/24` and `${FLANNEL_MTU}` is `1472` in the `/etc/sysconfig/docker`.

3. Start Docker using `service docker start` and type `ifconfig`; you might be able to see the virtual network device `docker0` and its allocated IP address from flannel.

How it works...

There are two virtual bridges named `flannel0` and `docker0` that are created in the previous steps. Let's take a look at their IP range using the `ip` command:

```
# checkout IPv4 network in local
$ ip -4 a | grep inet
    inet 127.0.0.1/8 scope host lo
    inet 10.42.1.171/24 brd 10.42.21.255 scope global dynamic ens160
    inet 192.168.50.0/16 scope global flannel0
    inet 192.168.50.1/24 scope global docker0
```

Host IP address is `10.42.1.171/24`, `flannel0` is `192.168.50.0/16`, `docker0` is `192.168.50.1/24`, and the route is set for the full flat IP range:

```
# check the route
$ route -n
Destination     Gateway         Genmask         Flags Metric Ref    Use
Iface
0.0.0.0         10.42.1.1       0.0.0.0         UG    100    0        0
ens160
192.168.0.0     0.0.0.0         255.255.0.0     U     0      0        0
flannel0
192.168.50.0    0.0.0.0         255.255.255.0   U     0      0        0
docker0
```

Let's go a little bit deeper to see how etcd stores flannel subnet information. You could retrieve the network configuration by using the `etcdctl` command in etcd:

```
# get network config
$ etcdctl get /coreos.com/network/config
{ "Network": "192.168.0.0/16" }

# show all the subnet leases
$ etcdctl ls /coreos.com/network/subnets
/coreos.com/network/subnets/192.168.50.0-24
```

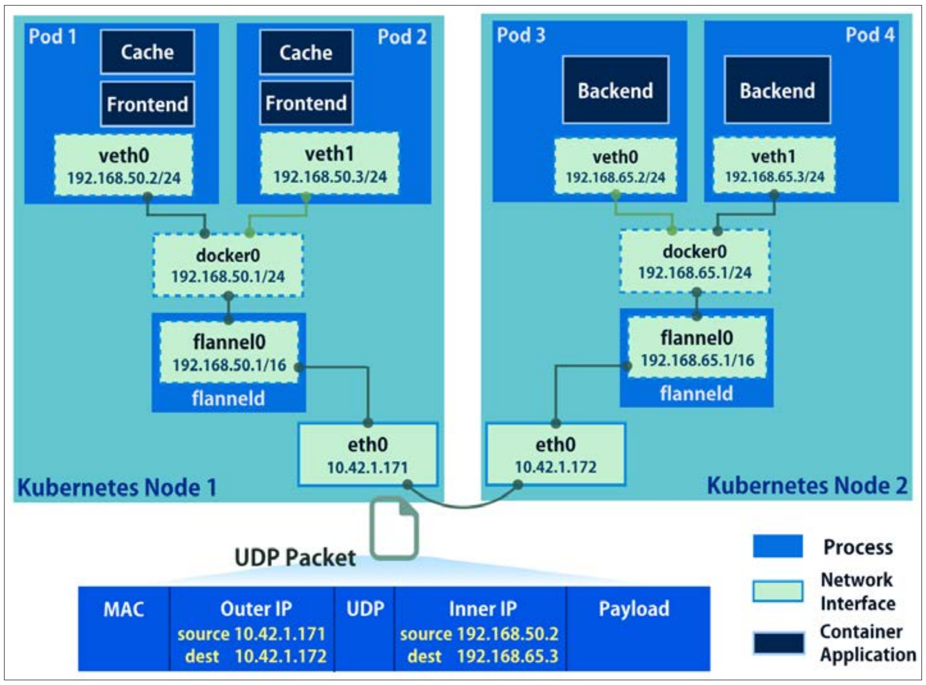
The preceding example shows that the network CIDR is `192.168.0.0/16`. There is one subnet lease. Check the value of the key; it's exactly the IP address of `eth0` on the host:

```
# show the value of the key of `/coreos.com/network/
subnets/192.168.50.0-24`
$ etcdctl get /coreos.com/network/subnets/192.168.50.0-24
{"PublicIP":"10.42.1.171"}
```

If you're using other backend solutions rather than simple UDP, you might see more configuration as follows:

```
# show the value when using different backend
$ etcdctl get /coreos.com/network/subnets/192.168.50.0-24
{"PublicIP":"10.97.1.171","BackendType":"vxlan","BackendData":{"VtepMAC":
"ee:ce:55:32:65:ce"}}
```

Following is an illustration about how a packet from **Pod1** goes through the overlay network to **Pod4**. As we discussed before, every pod will have its own IP address and the packet is encapsulated so that pod IPs are routable. The packet from **Pod1** will go through the **veth** (virtual network interface) device that connects to **docker0**, and routes to **flannel0**. The traffic is encapsulated by **flanneld** and sent to the host (**10.42.1.172**) of the target pod.



Let's perform a simple test by running two individual containers to see whether flannel works well. Assume we have two hosts (`10.42.1.171` and `10.42.1.172`) with different subnets, which are allocated by Flannel with the same etcd backend, and have launched Docker run by `docker run -it ubuntu /bin/bash` in each host:

Container 1 on host 1 (10.42.1.171)	Container 2 on host 2 (10.42.1.172)

`root@0cd2a2f73d8e:/# ifconfig eth0`
`eth0      Link encap:Ethernet HWaddr 02:42:c0:a8:3a:08`
`          inet addr:192.168.50.2 Bcast:0.0.0.0  Mask:255.255.255.0`
`          inet6 addr: fe80::42:c0ff:fea8:3a08/64 Scope:Link`
`          UP BROADCAST RUNNING MULTICAST  MTU:8951  Metric:1`
`          RX packets:8 errors:0 dropped:0 overruns:0 frame:0`
`          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0`
`          collisions:0 txqueuelen:0`
`          RX bytes:648 (648.0 B) TX bytes:648 (648.0 B)`
`root@0cd2a2f73d8e:/# ping 192.168.65.2`
`PING 192.168.4.10 (192.168.4.10) 56(84) bytes of data.`
`64 bytes from 192.168.4.10: icmp_seq=2 ttl=62 time=0.967 ms`
`64 bytes from 192.168.4.10: icmp_seq=3 ttl=62 time=1.00 ms`
	`root@619b3ae36d77:/# ifconfig eth0`
	`eth0      Link encap:Ethernet HWaddr 02:42:c0:a8:04:0a`
	`          inet addr:192.168.65.2 Bcast:0.0.0.0  Mask:255.255.255.0`
	`          inet6 addr: fe80::42:c0ff:fea8:40a/64 Scope:Link`
	`          UP BROADCAST RUNNING MULTICAST  MTU:8973  Metric:1`
	`          RX packets:8 errors:0 dropped:0 overruns:0 frame:0`
	`          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0`
	`          collisions:0 txqueuelen:0`
	`          RX bytes:648 (648.0 B) TX bytes:648 (648.0 B)`

We can see that two containers can communicate with each other using ping. Let's observe the packet using `tcpdump` in `host2`, which is a command-line tool that can help dump traffic on a network:

```
# install tcpdump in container
$ yum install -y tcpdump


# observe the UDP traffic from host2
$ tcpdump host 10.42.1.172 and udp
11:20:10.324392 IP 10.42.1.171.52293 > 10.42.1.172.6177: UDP, length 106
11:20:10.324468 IP 10.42.1.172.47081 > 10.42.1.171.6177: UDP, length 106
```

```
11:20:11.324639 IP 10.42.1.171.52293 > 10.42.1.172.6177: UDP, length 106
11:20:11.324717 IP 10.42.1.172.47081 > 10.42.1.171.6177: UDP, length 106
```

The traffic between the containers are encapsulated in UDP through port `6177` using flanneld.

See also

After setting up and understanding the overlay network, we have a good understanding of how flannel acts in Kubernetes. Check out the following recipes:

- The *Working with pods*, *Working with services* recipes in *Chapter 2*, *Walking through Kubernetes Concepts*
- The *Forwarding container ports* recipe in *Chapter 3*, *Playing with Containers*
- The *Authentication and authorization* recipe in *Chapter 7*, *Advanced Cluster Administration*

Configuring master

The master node of Kubernetes works as the control center of containers. The duties of which are taken charge by the master include serving as a portal to end users, assigning tasks to nodes, and gathering information. In this recipe, we will see how to set up Kubernetes master. There are three daemon processes on master:

- API Server
- Scheduler
- Controller Manager

We can either start them using the wrapper command, `hyperkube`, or individually start them as daemons. Both the solutions are covered in this section.

Getting ready

Before deploying the master node, make sure you have the etcd endpoint ready, which acts like the datastore of Kubernetes. You have to check whether it is accessible and also configured with the overlay network **Classless Inter-Domain Routing** (**CIDR** `https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing`). It is possible to check it using the following command line:

```
// Check both etcd connection and CIDR setting
$ curl -L <etcd endpoint URL>/v2/keys/coreos.com/network/config
```

If connection is successful, but the etcd configuration has no expected CIDR value, you can push value through `curl` as well:

```
$ curl -L <etcd endpoint URL>/v2/keys/coreos.com/network/config -XPUT -d
value="{ \"Network\": \"<CIDR of overlay network>\" }"
```

Besides this, please record the following items: the URL of `etcd endpoint`, the port exposed by `etcd endpoint`, and the CIDR of the overlay network. You will need them while configuring master's services.

How to do it...

In order to build up a master, we propose the following steps for installing the source code, starting with the daemons and then doing verification. Follow the procedure and you'll get a practical master eventually.

Installation

Here, we offer two kinds of installation procedures:

- One is a RHEL-based OS with package manager; master daemons are controlled by `systemd`
- The other one is for other Linux distributions; we build up master with binary files and service init scripts

CentOS 7 or Red Hat Enterprise Linux 7

1. RHEL 7, CentOS 7, or later have an official package for Kubernetes. You can install them via the `yum` command:

   ```
   // install Kubernetes master package
   # yum install kubernetes-master kubernetes-client
   ```

   The `kubernetes-master` package contains master daemons, while `kubernetes-client` installs a tool called `kubectl`, which is the Command Line Interface for communicating with the Kubernetes system. Since the master node is served as an endpoint for requests, with `kubectl` installed, users can easily control container applications and the environment through commands.

CentOS 7's RPM of Kubernetes

There are five Kubernetes RPMs (the `.rpm` files, `https://en.wikipedia.org/wiki/RPM_Package_Manager`) for different functionalities: `kubernetes`, `kubernetes-master`, `kubernetes-client`, `kubernetes-node`, and `kubernetes-unit-test`.

The first one, `kubernetes`, is just like a hyperlink to the following three items. You will install `kubernetes-master`, `kubernetes-client`, and `kubernetes-node` at once. The one named `kubernetes-node` is for node installation. And the last one, `kubernetes-unit-test` contains not only testing scripts, but also Kubernetes template examples.

2. Here are the files after `yum install`:

```
// profiles as environment variables for services
# ls /etc/kubernetes/
apiserver  config  controller-manager  scheduler
// systemd files
# ls /usr/lib/systemd/system/kube-*
/usr/lib/systemd/system/kube-apiserver.service           /usr/lib/
systemd/system/kube-scheduler.service
/usr/lib/systemd/system/kube-controller-manager.service
```

3. Next, we will leave the `systemd` files as the original ones and modify the values in the configuration files under the directory `/etc/kubernetes` to build a connection with etcd. The file named `config` is a shared environment file for several Kubernetes daemon processes. For basic settings, simply change items in `apiserver`:

```
# cat /etc/kubernetes/apiserver
###
# kubernetes system config
#
# The following values are used to configure the kube-apiserver
#

# The address on the local server to listen to.
KUBE_API_ADDRESS="--address=0.0.0.0"

# The port on the local server to listen on.
KUBE_API_PORT="--insecure-port=8080"
```

```
# Port nodes listen on
# KUBELET_PORT="--kubelet_port=10250"


# Comma separated list of nodes in the etcd cluster
KUBE_ETCD_SERVERS="--etcd_servers=<etcd endpoint URL>:<etcd
exposed port>"


# Address range to use for services
KUBE_SERVICE_ADDRESSES="--service-cluster-ip-range=<CIDR of
overlay network>"


# default admission control policies
KUBE_ADMISSION_CONTROL="--admission_control=NamespaceLifecycle,Nam
espaceExists,LimitRanger,SecurityContextDeny,ServiceAccount,Resour
ceQuota"


# Add your own!
KUBE_API_ARGS="--cluster_name=<your cluster name>"
```

4. Then, start the daemon `kube-apiserver`, `kube-scheduler`, and `kube-controller-manager` one by one; the command `systemctl` can help for management. Be aware that `kube-apiserver` should always start first, since `kube-scheduler` and `kube-controller-manager` connect to the Kubernetes API server when they start running:

```
// start services
# systemctl start kube-apiserver
# systemctl start kube-scheduler
# systemctl start kube-controller-manager
// enable services for starting automatically while server boots
up.
# systemctl enable kube-apiserver
# systemctl enable kube-scheduler
# systemctl enable kube-controller-manager
```

Adding daemon dependency

1. Although `systemd` does not return error messages without the API server running, both `kube-scheduler` and `kube-controller-manager` get connection errors and do not provide regular services:

```
$ sudo systemctl status kube-scheduler -l—output=cat kube-
scheduler.service - Kubernetes Scheduler Plugin
```

```
   Loaded: loaded (/usr/lib/systemd/system/kube-scheduler.service;
enabled)
   Active: active (running) since Thu 2015-11-19 07:21:57 UTC;
5min ago
     Docs: https://github.com/GoogleCloudPlatform/kubernetes
 Main PID: 2984 (kube-scheduler)
   CGroup: /system.slice/kube-scheduler.service
           └─2984 /usr/bin/kube-scheduler—logtostderr=true—v=0
--master=127.0.0.1:8080
E1119 07:27:05.471102    2984 reflector.go:136] Failed
to list *api.Node: Get http://127.0.0.1:8080/api/v1/
nodes?fieldSelector=spec.unschedulable%3Dfalse: dial tcp
127.0.0.1:8080: connection refused
```

2. Therefore, in order to prevent the starting order to affect performance, you can add two settings under the section of `systemd.unit` in `/usr/lib/systemd/system/kube-scheduler` and `/usr/lib/systemd/system/kube-controller-manager`:

   ```
   [Unit]
   Description=Kubernetes Controller Manager
   Documentation=https://github.com/GoogleCloudPlatform/kubernetes
   After=kube-apiserver.service
   Wants=kube-apiserver.service
   ```

   With the preceding settings, we can make sure `kube-apiserver` is the first started daemon.

3. Furthermore, if you expect the scheduler and the controller manager to always be running along with a healthy API server, which means if `kube-apiserver` is stopped, `kube-scheduler` and `kube-controller-manager` will be stopped as well; you can change `systemd.unit` item `Wants` to `Requires`, as follows:

   ```
   Requires=kube-apiserver.service
   ```

   `Requires` has more strict restrictions. In case the daemon `kube-apiserver` has crashed, `kube-scheduler` and `kube-controller-manager` would also be stopped. On the other hand, configuration with `Requires` is hard for debugging master installation. It is recommended that you enable this parameter once you make sure every setting is correct.

Other Linux options

It is also possible that we download a binary file for installation. The official website for the latest release is here: `https://github.com/kubernetes/kubernetes/releases`:

1. We are going to install the version tagged as **Latest release** and start all the daemons with the wrapper command `hyperkube`:

   ```
   // download Kubernetes package
   # curl -L -O https://github.com/GoogleCloudPlatform/kubernetes/
   releases/download/v1.1.2/kubernetes.tar.gz

   // extract the tarball to specific local, here we put it under /
   opt. the KUBE_HOME would be /opt/kubernetes
   # tar zxvf kubernetes.tar.gz -C /opt/

   // copy all binary files to system directory
   # cp /opt/kubernetes/server/bin/* /usr/local/bin/
   ```

2. The next step is to create a startup script (init), which would cover three master daemons and start them individually:

   ```
   # cat /etc/init.d/kubernetes-master
   #!/bin/bash
   #
   # This shell script takes care of starting and stopping kubernetes
   master

   # Source function library.
   . /etc/init.d/functions

   # Source networking configuration.
   . /etc/sysconfig/network

   prog=/usr/local/bin/hyperkube
   lockfile=/var/lock/subsys/`basename $prog`
   hostname=`hostname`
   logfile=/var/log/kubernetes.log

   CLUSTER_NAME="<your cluster name>"
   ```

```
ETCD_SERVERS="<etcd endpoint URL>:<etcd exposed port>"
CLUSTER_IP_RANGE="<CIDR of overlay network>"
MASTER="127.0.0.1:8080"
```

3. To manage your Kubernetes settings more easily and clearly, we will put the declaration of changeable variables at the beginning of this `init` script. Please double-check the etcd URL and overlay network CIDR to confirm that they are the same as your previous installation:

```
start() {

  # Start daemon.
  echo $"Starting apiserver: "
  daemon $prog apiserver \
  --service-cluster-ip-range=${CLUSTER_IP_RANGE} \
  --port=8080 \
  --address=0.0.0.0 \
  --etcd_servers=${ETCD_SERVERS} \
  --cluster_name=${CLUSTER_NAME} \
  > ${logfile}_apiserver 2>&1 &

  echo $"Starting controller-manager: "
  daemon $prog controller-manager \
  --master=${MASTER} \
  > ${logfile}_controller-manager 2>&1 &

  echo $"Starting scheduler: "
  daemon $prog scheduler \
  --master=${MASTER} \
  > ${logfile}_scheduler 2>&1 &

  RETVAL=$?
  [ $RETVAL -eq 0 ] && touch $lockfile
  return $RETVAL
}

stop() {
  [ "$EUID" != "0" ] && exit 4
        echo -n $"Shutting down $prog: "
  killproc $prog
  RETVAL=$?
        echo
  [ $RETVAL -eq 0 ] && rm -f $lockfile
  return $RETVAL
}
```

4. Next, feel free to attach the following lines as the last part in the script for general service usage:

   ```
   # See how we were called.
   case "$1" in
     start)
     start
     ;;
     stop)
     stop
     ;;
     status)
     status $prog
     ;;
     restart|force-reload)
     stop
     start
     ;;
     try-restart|condrestart)
     if status $prog > /dev/null; then
         stop
         start
     fi
     ;;
     reload)
     exit 3
     ;;
     *)
     echo $"Usage: $0 {start|stop|status|restart|try-restart|force-
   reload}"
     exit 2
   esac
   ```

5. Now, it is good to start the service named `kubernetes-master`:

   ```
   $sudo service kubernetes-master start
   ```

At the time of writing this book, the latest version of Kubernetes was 1.1.2. So, we will use 1.1.2 in the examples for most of the chapters.

Verification

1. After starting all the three daemons of the master node, you can verify whether they are running properly by checking the service status. Both the commands, `systemd` and `service`, are able to get the logs:

   ```
   # systemd status <service name>
   ```

2. For a more detailed log in history, you can use the command `journalctl`:

   ```
   # journalctl -u <service name> --no-pager --full
   ```

   Once you find a line showing `Started...` in the output, you can confirm that the service setup has passed the verification.

3. Additionally, the dominant command in Kubernetes, `kubectl`, can begin the operation:

   ```
   // check Kubernetes version
   # kubectl version
   Client Version: version.Info{Major:"1", Minor:"0.3", GitVersion:"v
   1.0.3.34+b9a88a7d0e357b", GitCommit:"b9a88a7d0e357be2174011dd2b127
   038c6ea8929", GitTreeState:"clean"}
   Server Version: version.Info{Major:"1", Minor:"0.3", GitVersion:"v
   1.0.3.34+b9a88a7d0e357b", GitCommit:"b9a88a7d0e357be2174011dd2b127
   038c6ea8929", GitTreeState:"clean"}
   ```

See also

From the recipe, you know how to create your own Kubernetes master. You can also check out the following recipes:

- *Exploring architecture*
- *Configuring nodes*
- The *Building multiple masters* recipe in *Chapter 4*, *Building a High Availability Cluster*
- The *Building the Kubernetes infrastructure in AWS* recipe in *Chapter 6*, *Building Kubernetes on AWS*
- The *Authentication and authorization* recipe in *Chapter 7*, *Advanced Cluster Administration*

Configuring nodes

Node is the slave in the Kubernetes cluster. In order to let master take a node under its supervision, node installs an agent called kubelet for registering itself to a specific master. After registering, daemon kubelet also handles container operations and reports resource utilities and container statuses to the master. The other daemon running on the node is kube-proxy, which manages TCP/UDP packets between containers. In this section, we will show you how to configure a node.

Getting ready

Since node is the worker of Kubernetes and the most important duty is running containers, you have to make sure that Docker and flanneld are installed at the beginning. Kubernetes relies on Docker helping applications to run in containers. And through flanneld, the pods on separated nodes can communicate with each other.

After you have installed both the daemons, according to the file `/run/flannel/subnet.env`, the network interface `docker0` should be underneath the same LAN as `flannel0`:

```
# cat /run/flannel/subnet.env
FLANNEL_SUBNET=192.168.31.1/24
FLANNEL_MTU=8973
FLANNEL_IPMASQ=true

// check the LAN of both flanneld0 and docker0
# ifconfig docker0 ; ifconfig flannel0
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 192.168.31.1  netmask 255.255.255.0  broadcast 0.0.0.0
        ether 02:42:6e:b9:a7:51  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
flannel0: flags=81<UP,POINTOPOINT,RUNNING>  mtu 8973
        inet 192.168.31.0  netmask 255.255.0.0  destination 192.168.11.0
        unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
txqueuelen 500  (UNSPEC)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

If `docker0` is in a different CIDR range, you may take the following service scripts as a reference for a reliable Docker service setup:

```
# cat /etc/sysconfig/docker
# /etc/sysconfig/docker
#
# Other arguments to pass to the docker daemon process
# These will be parsed by the sysv initscript and appended
# to the arguments list passed to docker -d, or docker daemon where
docker version is 1.8 or higher

. /run/flannel/subnet.env

other_args="--bip=${FLANNEL_SUBNET} --mtu=${FLANNEL_MTU}"
DOCKER_CERT_PATH=/etc/docker
```

Alternatively, by way of `systemd`, the configuration also originally handles the dependency:

```
$ cat /etc/systemd/system/docker.service.requires/flanneld.service
[Unit]
Description=Flanneld overlay address etcd agent
After=network.target
Before=docker.service

[Service]
Type=notify
EnvironmentFile=/etc/sysconfig/flanneld
EnvironmentFile=-/etc/sysconfig/docker-network
ExecStart=/usr/bin/flanneld -etcd-endpoints=${FLANNEL_ETCD} -etcd-
prefix=${FLANNEL_ETCD_KEY} $FLANNEL_OPTIONS
ExecStartPost=/usr/libexec/flannel/mk-docker-opts.sh -k DOCKER_NETWORK_
OPTIONS -d /run/flannel/docker

[Install]
RequiredBy=docker.service

$ cat /run/flannel/docker
DOCKER_OPT_BIP="--bip=192.168.31.1/24"
DOCKER_OPT_MTU="--mtu=8973"
DOCKER_NETWORK_OPTIONS=" --bip=192.168.31.1/24 --mtu=8973 "
```

Once you have modified the Docker service script to a correct one, stop the Docker service, clean its network interface, and start it again.

For more details on the flanneld setup and Docker integration, please refer to the recipe *Creating an overlay network*.

You can even configure a master to the node; just install the necessary daemons.

How to do it...

Once you verify that Docker and flanneld are good to go on your node host, continue to install the Kubernetes package for the node. We'll cover both RPM and tarball setup.

Installation

This will be the same as the Kubernetes master installation, Linux OS having the command line tool `yum`, the package management utility, can easily install the node package. On the other hand, we are also able to install the latest version through downloading a tarball file and copy binary files to the specified system directory, which is suitable for every Linux distribution. You can try either of the solutions for your deployment.

CentOS 7 or Red Hat Enterprise Linux 7

1. First, we will install the package `kubernetes-node`, which is what we need for the node:

   ```
   // install kubernetes node package
   $ yum install kubernetes-node
   ```

   The package `kubernetes-node` includes two daemon processes, `kubelet` and `kube-proxy`.

2. We need to modify two configuration files to access the master node:

   ```
   # cat /etc/kubernetes/config
   ###
   # kubernetes system config
   #
   # The following values are used to configure various aspects of
   all
   # kubernetes services, including
   #
   #   kube-apiserver.service
   #   kube-controller-manager.service
   #   kube-scheduler.service
   #   kubelet.service
   #   kube-proxy.service
   # logging to stderr means we get it in the systemd journal
   KUBE_LOGTOSTDERR="--logtostderr=true"
   ```

```
# journal message level, 0 is debug
KUBE_LOG_LEVEL="--v=0"

# Should this cluster be allowed to run privileged docker
containers
KUBE_ALLOW_PRIV="--allow_privileged=false"

# How the controller-manager, scheduler, and proxy find the
apiserver
KUBE_MASTER="--master=<master endpoint>:8080"
```

3. In the configuration file, we will change the master location argument to the machine's URL/IP, where you installed master. If you specified another exposed port for the API server, remember to update it as well, instead of port `8080`:

```
# cat /etc/kubernetes/kubelet
###
# kubernetes kubelet (node) config

# The address for the info server to serve on (set to 0.0.0.0 or
"" for all interfaces)
KUBELET_ADDRESS="--address=0.0.0.0"

# The port for the info server to serve on
# KUBELET_PORT="--port=10250"

# You may leave this blank to use the actual hostname
KUBELET_HOSTNAME="--hostname_override=127.0.0.1"

# location of the api-server
KUBELET_API_SERVER="--api_servers=<master endpoint>:8080"

# Add your own!
KUBELET_ARGS=""
```

We open the kubelet address for all the interfaces and attached master location.

4. Then, it is good to start services using the command `systemd`. There is no dependency between `kubelet` and `kube-proxy`:

   ```
   // start services
   # systemctl start kubelet
   # systemctl start kube-proxy
   // enable services for starting automatically while server boots
   up.
   # systemctl enable kubelet
   # systemctl enable kube-proxy
   // check the status of services
   # systemctl status kubelet
   # systemctl status kube-proxy
   ```

Other Linux options

1. We can also download the latest Kubernetes binary files and write a customized service init script for node configuration. The tarball of Kubernetes' latest updates will be released at `https://github.com/kubernetes/kubernetes/releases`:

   ```
   // download Kubernetes package
   # curl -L -O https://github.com/GoogleCloudPlatform/kubernetes/
   releases/download/v1.1.2/kubernetes.tar.gz

   // extract the tarball to specific local, here we put it under /
   opt. the KUBE_HOME would be /opt/kubernetes
   # tar zxvf kubernetes.tar.gz -C /opt/

   // copy all binary files to system directory
   # cp /opt/kubernetes/server/bin/* /usr/local/bin/
   ```

2. Next, a file named `kubernetes-node` is created under `/etc/init.d` with the following content:

   ```
   # cat /etc/init.d/kubernetes-node
   #!/bin/bash
   #
   # kubernetes    This shell script takes care of starting and
   stopping kubernetes

   # Source function library.
   . /etc/init.d/functions
   ```

```
# Source networking configuration.
. /etc/sysconfig/network

prog=/usr/local/bin/hyperkube
lockfile=/var/lock/subsys/`basename $prog`
MASTER_SERVER="<master endpoint>"
hostname=`hostname`
logfile=/var/log/kubernetes.log
```

3. Be sure to provide the master URL/IP for accessing the Kubernetes API server. If you're trying to install a node package on the master host as well, which means make master also work as a node, the API server should work on the local host. If so, you can attach `localhost` or `127.0.0.1` at `<master endpoint>`:

```
start() {
    # Start daemon.
    echo $"Starting kubelet: "
    daemon $prog kubelet \
        --api_servers=http://${MASTER_SERVER}:8080 \
        --v=2 \
        --address=0.0.0.0 \
        --enable_server \
        --hostname_override=${hostname} \
        > ${logfile}_kubelet 2>&1 &

    echo $"Starting proxy: "
    daemon $prog proxy \
        --master=http://${MASTER_SERVER}:8080 \
        --v=2 \
        > ${logfile}_proxy 2>&1 &

    RETVAL=$?
    [ $RETVAL -eq 0 ] && touch $lockfile
    return $RETVAL
}
stop() {
    [ "$EUID" != "0" ] && exit 4
        echo -n $"Shutting down $prog: "
    killproc $prog
    RETVAL=$?
        echo
    [ $RETVAL -eq 0 ] && rm -f $lockfile
    return $RETVAL
}
```

4. The following lines are for general daemon management, attaching them in the script to get the functionalities:

```
# See how we were called.
case "$1" in
  start)
    start
    ;;
  stop)
    stop
    ;;
  status)
    status $prog
    ;;
  restart|force-reload)
    stop
    start
    ;;
  try-restart|condrestart)
    if status $prog > /dev/null; then
        stop
        start
    fi
    ;;
  reload)
    exit 3
    ;;
  *)
    echo $"Usage: $0 {start|stop|status|restart|try-restart|force-
reload}"
    exit 2
esac
```

5. Now, you can start the service with the name of your `init` script:

```
# service kubernetes-node start
```

Verification

In order to check whether a node is well-configured, the straightforward way would be to check it from the master side:

```
// push command at master
# kubelet get nodes
NAME                                   LABELS
STATUS
ip-10-97-217-56.sdi.trendnet.org   kubernetes.io/
hostname=ip-10-97-217-56.sdi.trendnet.org   Ready
```

See also

It is also recommended to read the recipes about the architecture of the cluster and system environment. Since the Kubernetes node is like a worker, who receives tasks and listens to the others; they should be built after the other components. It is good for you to get more familiar with the whole system before building up nodes. Furthermore, you can also manage the resource in nodes. Please check the following recipes for more information:

- *Exploring architecture*
- *Preparing your environment*
- The *Setting resource in nodes* recipe in *Chapter 7*, *Advanced Cluster Administration*

Run your first container in Kubernetes

Congratulations! You've built your own Kubernetes cluster in the previous sections. Now, let's get on with running your very first container nginx (`http://nginx.org/`), which is an open source reverse proxy server, load balancer, and web server.

Getting ready

Before we start running the first container in Kubernetes, it's better to check whether every component works as expected. Please follow these steps on master to check whether the environment is ready to use:

1. Check whether the Kubernetes components are running:

```
# check component status are all healthy
$ kubectl get cs
NAME                 STATUS    MESSAGE              ERROR
controller-manager   Healthy   ok                   nil
scheduler            Healthy   ok                   nil
etcd-0               Healthy   {"health": "true"}   nil
```

> If any one of the components is not running, check out the settings in the previous sections. Restart the related services, such as `service kube-apiserver start`.

2. Check the master status:

```
# Check master is running
$ kubectl cluster-info
Kubernetes master is running at http://localhost:8080
```

If the Kubernetes master is not running, restart the service using `service kubernetes-master start` or `/etc/init.d/kubernetes-master start`.

3. Check whether all the nodes are ready:

```
# check nodes are all Ready
$ kubectl get nodes
NAME          LABELS                                    STATUS
kub-node1   kubernetes.io/hostname=kub-node1    Ready
kub-node2   kubernetes.io/hostname=kub-node2    Ready
```

If one node is expected as `Ready` but is `NotReady`, go to that node to restart Docker and the node service using `service docker start` and `service kubernetes-node start`.

Before we go to the next section, make sure the nodes are accessible to the Docker registry. We will use the nginx image from Docker Hub (`https://hub.docker.com/`) as an example. If you want to run your own application, be sure to dockerize it first! What you need to do for your custom application is to write a Dockerfile (`https://docs.docker.com/v1.8/reference/builder`), build, and push it into the public/private Docker registry.

Test your node connectivity with the public/private Docker registry

On your node, try `docker pull nginx` to test whether you can pull the image from Docker Hub. If you're behind a proxy, please add `HTTP_PROXY` into your Docker configuration file (normally, in `/etc/sysconfig/docker`). If you want to run the image from the private repository in Docker Hub, using `docker login` on the node to place your credential in `~/.docker/config.json`, copy the credentials into `/var/lib/kubelet/.dockercfg` in the json format and restart Docker:

```
# put the credential of docker registry
$ cat /var/lib/kubelet/.dockercfg

{
        "<docker registry endpoint>": {
                "auth": "SAMPLEAUTH=",
                "email": "noreply@sample.com"
        }
}
```

If you're using your own private registry, specify `INSECURE_REGISTRY` in the Docker configuration file.

How to do it...

We will use the official Docker image of nginx as an example. The image is prebuilt in Docker Hub (`https://hub.docker.com/_/nginx/`).

Many official and public images are available on Docker Hub so that you do not need to build it from scratch. Just pull it and set up your custom setting on top of it.

Running an HTTP server (nginx)

1. On the Kubernetes master, we could use `kubectl run` to create a certain number of containers. The Kubernetes master will then schedule the pods for the nodes to run:

```
$ kubectl run <replication controller name> --image=<image name>
--replicas=<number of replicas> [--port=<exposing port>]
```

2. The following example will create two replicas with the name `my-first-nginx` from the nginx image and expose port `80`. We could deploy one or more containers in what is referred to as a pod. In this case, we will deploy one container per pod. Just like a normal Docker behavior, if the nginx image doesn't exist in local, it will pull it from Docker Hub by default:

```
# Pull the nginx image and run with 2 replicas, and expose the
container port 80

$ kubectl run my-first-nginx --image=nginx --replicas=2 --port=80

CONTROLLER        CONTAINER(S)      IMAGE(S)    SELECTOR
REPLICAS

my-first-nginx    my-first-nginx    nginx       run=my-first-nginx
2
```

The name of replication controller <my-first-nginx> cannot be duplicate

The resource (pods, services, replication controllers, and so on) in one Kubernetes namespace cannot be duplicate. If you run the preceding command twice, the following error will pop up:

```
Error from server: replicationControllers "my-first-
nginx" already exists
```

3. Let's get and see the current status of all the pods using `kubectl get pods`. Normally, the status of the pods will hold on `Pending` for a while, since it takes some time for the nodes to pull the image from Docker Hub:

```
# get all pods

$ kubectl get pods

NAME                          READY     STATUS     RESTARTS    AGE
```

```
my-first-nginx-nzygc      1/1        Running    0           1m
my-first-nginx-yd84h      1/1        Running    0           1m
```

If the pod status is not running for a long time

You could always use `kubectl get pods` to check the current status of the pods and `kubectl describe pods $pod_name` to check the detailed information of a pod. If you make a typo of the image name, you might get the `Image not found` error message, and if you are pulling the images from a private repository or registry without proper credentials setting, you might get the `Authentication error` message. If you get the `Pending` status for a long time and check out the node capacity, make sure you don't run too many replicas that exceed the node capacity described in the *Preparing your environment* section. If there are other unexpected error messages, you could either stop the pods or the entire replication controller to force master to schedule the tasks again.

4. After waiting a few seconds, there are two pods running with the `Running` status:

```
# get replication controllers
$ kubectl get rc
CONTROLLER          CONTAINER(S)        IMAGE
(S)                                                         SELECTOR
REPLICAS
my-first-nginx     my-first-nginx      nginx
run=my-first-nginx
2
```

Exposing the port for external access

We might also want to create an external IP address for the `nginx` replication controller. On cloud providers, which support an external load balancer (such as Google Compute Engine) using the `LoadBalancer` type, will provision a load balancer for external access. On the other hand, you can still expose the port by creating a Kubernetes service as follows, even though you're not running on the platforms that support an external load balancer. We'll describe how to access this externally later:

```
# expose port 80 for replication controller named my-first-nginx
$ kubectl expose rc my-first-nginx --port=80 --type=LoadBalancer
NAME              LABELS               SELECTOR             IP(S)
PORT(S)
my-first-nginx    run=my-first-nginx   run=my-first-nginx               80/
TCP
```

We can see the service status we just created:

```
# get all services
$ kubectl get service
NAME                          LABELS
SELECTOR
IP(S)               PORT(S)
my-first-nginx                run=my-first-nginx
run=my-first-nginx
192.168.61.150        80/TCP
```

Congratulations! You just ran your first container with a Kubernetes pod and exposed port `80` with the Kubernetes service.

Stopping the application

We could stop the application using commands such as the `stop` replication controller and service. Before this, we suggest you read through the following introduction first to understand more about how it works:

```
# stop replication controller named my-first-nginx
$ kubectl stop rc my-first-nginx
replicationcontrollers/my-first-nginx


# stop service named my-first-nginx
$ kubectl stop service my-first-nginx
services/my-first-nginx
```

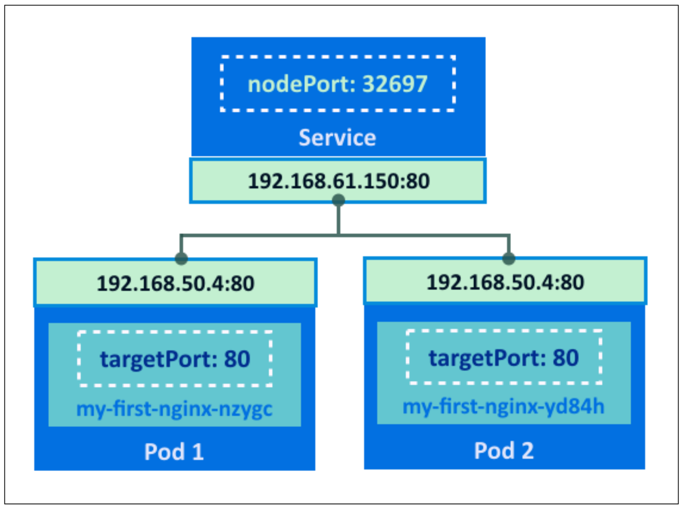
How it works...

Let's take a look at the insight of the service using `describe` in the `kubectl` command. We will create one Kubernetes service with the type `LoadBalancer`, which will dispatch the traffic into two `Endpoints` `192.168.50.4` and `192.168.50.5` with port `80`:

```
$ kubectl describe service my-first-nginx
Name:       my-first-nginx
Namespace:     default
Labels:       run=my-first-nginx
Selector:     run=my-first-nginx
Type:       LoadBalancer
IP:       192.168.61.150
Port:       <unnamed>   80/TCP
```

```
NodePort:     <unnamed>  32697/TCP
Endpoints:     192.168.50.4:80,192.168.50.5:80
Session Affinity:   None
No events.
```

`Port` here is an abstract service port, which will allow any other resources to access the service within the cluster. The `nodePort` will be indicating the external port for allowing external access. The `targetPort` is the port the container allows traffic into; by default, it will be the same with `Port`. The illustration is as follows. External access will access service with `nodePort`. Service acts as a load balancer to dispatch the traffic to the pod using `Port 80`. The pod will then pass through the traffic into the corresponding container using `targetPort 80`:



In any nodes or master (if your master has flannel installed), you should be able to access the nginx service using ClusterIP `192.168.61.150` with port `80`:

```
# curl from service IP
$ curl 192.168.61.150:80
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed
and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

It will be the same result if we curl to the target port of the pod directly:

```
# curl from endpoint
$ curl 192.168.50.4:80
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
```

```
<p>If you see this page, the nginx web server is successfully installed
and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

If you'd like to try out external access, use your browser to access the external IP address. Please note that the external IP address depends on which environment you're running in.

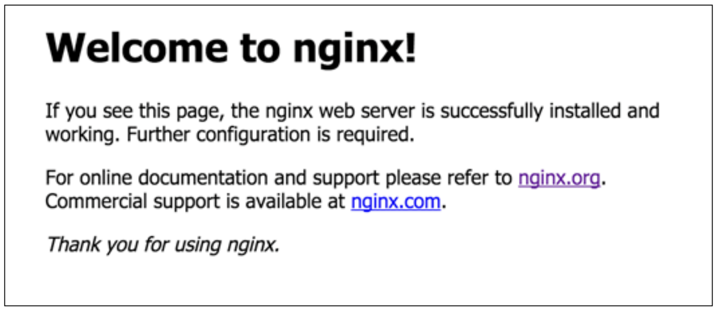
In Google Compute Engine, you could access it via a ClusterIP with proper firewall rules setting:

```
$ curl http://<clusterIP>
```

In a custom environment, such as on a premise datacenter, you could go through the IP address of nodes to access to:

```
$ curl http://<nodeIP>:<nodePort>
```

You should be able to see the following page using a web browser:



Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

See also

We have run our very first container in this section. Now:

- To explore more of the concepts in Kubernetes, refer to *Chapter 2*, *Walking through Kubernetes Concepts*

2
Walking through Kubernetes Concepts

In this chapter, we will cover:

- An overview of Kubernetes control
- Working with pods
- Working with a replication controller
- Working with services
- Working with volumes
- Working with secrets
- Working with names
- Working with namespaces
- Working with labels and selectors

Introduction

In this chapter, we will start by creating different kinds of resources on the Kubernetes system. To realize your application in a microservices structure, reading the recipes in this chapter will be a good start to understanding the concepts of the Kubernetes resources and consolidating them. After you deploy applications in Kubernetes, you can work on its scalable and efficient container management, and also fulfill the DevOps delivering procedure of microservices.

An overview of Kubernetes control

Working with Kubernetes is quite easy, using either **Command Line Interface** (**CLI**) or API (RESTful). This section will describe Kubernetes control by CLI. The CLI we used in this chapter is version 1.1.3.

Getting ready

After you install Kubernetes master, you can run a `kubectl` command as follows. It shows the `kubectl` and Kubernetes master versions (both 1.1.3).

```
% kubectl version
Client Version: version.Info{Major:"1", Minor:"1", GitVersion:"v1.1.3",
GitCommit:"6a81b50c7e97bbe0ade075de55ab4fa34f049dc2",
GitTreeState:"clean"}
Server Version: version.Info{Major:"1", Minor:"1", GitVersion:"v1.1.3",
GitCommit:"6a81b50c7e97bbe0ade075de55ab4fa34f049dc2",
GitTreeState:"clean"}
```

How to do it...

`kubectl` connects the Kubernetes API server using RESTful API. By default it attempts to access the localhost, otherwise you need to specify the API server address using the `--server` parameter. Therefore, it is recommended to use `kubectl` on the API server machine for practice.

If you use `kubectl` over the network, you need to consider authentication and authorization for the API server. See *Chapter 7, Advanced Cluster Administration*.

How it works...

`kubectl` is the only command for Kubernetes clusters, and it controls the Kubernetes cluster manager. Find more information at `http://kubernetes.io/docs/user-guide/kubectl-overview/`. Any container, or Kubernetes cluster operation, can be performed by a `kubectl` command.

In addition, `kubectl` allows the inputting of information by either the command line's optional arguments, or by file (use `-f` option), but it is highly recommended to use file, because you can maintain the Kubernetes cluster as code:

```
kubectl [command] [TYPE] [NAME] [flags]
```

The attributes of the preceding command are explained as follows:

- `command`: Specifies the operation that you want to perform on one or more resources.
- `TYPE`: Specifies the resource type. Resource types are case-sensitive and you can specify the singular, plural, or abbreviated forms.
- `NAME`: Specifies the name of the resource. Names are case-sensitive. If the name is omitted, details for all resources are displayed.
- `flags`: Specifies optional flags.

For example, if you want to launch nginx, you can use the `kubectl run` command as the following:

```
# /usr/local/bin/kubectl run my-first-nginx --image=nginx
replicationcontroller "my-first-nginx"
```

However, you can write either a YAML file or a JSON file to perform similar operations. For example, the YAML format is as follows:

```
# cat nginx.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-first-nginx
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: my-first-nginx
        image: nginx
```

Then specify the `create -f` option to execute the `kubectl` command as follows:

```
# kubectl create -f nginx.yaml
replicationcontroller "my-first-nginx" created
```

If you want to see the status of the replication controller, type the `kubectl get` command as follows:

```
# kubectl get replicationcontrollers
CONTROLLER       CONTAINER(S)     IMAGE(S)    SELECTOR     REPLICAS    AGE
my-first-nginx   my-first-nginx   nginx       app=nginx    1           12s
```

If you also want the support abbreviation, type the following:

```
# kubectl get rc
CONTROLLER       CONTAINER(S)     IMAGE(S)    SELECTOR     REPLICAS    AGE
my-first-nginx   my-first-nginx   nginx       app=nginx    1           1m
```

If you want to delete these resources, type the `kubectl delete` command as follows:

```
# kubectl delete rc my-first-nginx
replicationcontroller "my-first-nginx" deleted
```

The `kubectl` command supports many kinds of sub-commands, use `-h` option to see the details, for example:

```
//display whole sub command options
# kubectl -h

//display sub command "get" options
# kubectl get -h

//display sub command "run" options
# kubectl run -h
```

See also

This recipe describes how to use the `kubectl` command to control the Kubernetes cluster. The following recipes describe how to set up Kubernetes components:

- The *Building datastore*, *Creating an overlay network*, *Configuring master*, and *Configuring nodes* recipes in *Chapter 1*, *Building Your Own Kubernetes*

Working with pods

The pod is a group of one or more containers and the smallest deployable unit in Kubernetes. Pods are always co-located and co-scheduled, and run in a shared context. Each pod is isolated by the following Linux namespaces:

- Process ID (PID) namespace
- Network namespace
- Interprocess Communication (IPC) namespace
- Unix Time Sharing (UTS) namespace

In a pre-container world, they would have been executed on the same physical or virtual machine.

It is useful to construct your own application stack pod (for example, web server and database) that are mixed by different Docker images.

Getting ready

You must have a Kubernetes cluster and make sure that the Kubernetes node has accessibility to the Docker Hub (`https://hub.docker.com`) in order to download Docker images. You can simulate downloading a Docker image by using the `docker pull` command as follows:

```
//run as root on node machine

# docker pull centos
latest: Pulling from centos

47d44cb6f252: Pull complete
168a69b62202: Pull complete
812e9d9d677f: Pull complete
4234bfdd88f8: Pull complete
ce20c473cd8a: Pull complete
Digest: sha256:c96eeb93f2590858b9e1396e808d817fa0ba4076c68b59395445cb95
7b524408
Status: Downloaded newer image for centos:latest
```

How to do it...

1. Log in to the Kubernetes master machine and prepare the following YAML file. It defines the launch nginx container and the CentOS container.
2. The nginx container opens the HTTP port (TCP/80). On the other hand, the CentOS container attempts to access the `localhost:80` every three seconds using the `curl` command:

```
# cat my-first-pod.yaml

apiVersion: v1
kind: Pod
metadata:
  name: my-first-pod
spec:
  containers:
  - name: my-nginx
    image: nginx
  - name: my-centos
    image: centos
    command: ["/bin/sh", "-c", "while : ;do curl http://
localhost:80/; sleep 3; done"]
```

3. Then, execute the `kubectl create` command to launch `my-first-pod` as follows:

```
# kubectl create -f my-first-pod.yaml
pod "my-first-pod" created
```

   It takes between a few seconds and minutes, depending on the network bandwidth to the Docker Hub and Kubernetes nodes spec.

4. You can check `kubectl get pods` to see the status as follows:

```
//still downloading Docker images (0/2)
# kubectl get pods
NAME            READY     STATUS     RESTARTS    AGE
my-first-pod    0/2       Running    0           6s

//it also supports shorthand format as "po"
# kubectl get po
```

```
NAME            READY     STATUS    RESTARTS   AGE
my-first-pod    0/2       Running   0          7s


//my-first-pod is running (2/2)


# kubectl get pods
NAME            READY     STATUS    RESTARTS   AGE
my-first-pod    2/2       Running   0          8s
```

Now both the nginx container (`my-nginx`) and the CentOS container (`my-centos`) are ready.

5. Let's check whether the CentOS container can access nginx or not. You can check the stdout (standard output) by using the `kubectl logs` command and specifying the CentOS container (`my-centos`) as follows:

```
//it shows last 30 lines output (--tail=30)


# kubectl logs my-first-pod -c my-centos --tail=30
</body>
</html>
  % Total    % Received % Xferd  Average Speed   Time    Time
Time  Current
                                 Dload  Upload   Total   Spent
Left  Speed
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
```

```
<p>If you see this page, the nginx web server is successfully
installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
100   612  100   612    0     0   4059      0 --:--:-- --:--:--
--:--:--  4080
```

As you can see, the pod links two different containers, nginx and CentOS into the same Linux namespace.

How it works...

When launching a pod, the Kubernetes scheduler dispatches to the kubelet process to handle all the operations to launch both nginx and CentOS containers.

If you have two or more nodes, you can check the `-o wide` option to find a node which runs a pod:

```
//it indicates Node ip-10-96-219-25 runs my-first-pod

# kubectl get pods -o wide
NAME            READY     STATUS     RESTARTS   AGE       NODE
my-first-pod    2/2       Running    0          2m        ip-10-96-219-25
```

Log in to that node, then you can check the `docker ps` command to see the running containers as follows:

```
# docker ps
CONTAINER ID        IMAGE                                       COMMAND
CREATED             STATUS              PORTS                    NAMES
b7eb8d0925b2        centos                                      "/
bin/sh -c 'while :   2 minutes ago       Up 2 minutes
k8s_my-centos.704bf394_my-first-pod_default_a3b78651-a061-11e5-a7fb-
06676ae2a427_f8b61e2b
```

```
55d987322f53        nginx                                  "nginx
-g 'daemon of    2 minutes ago       Up 2 minutes
k8s_my-nginx.608bdf36_my-first-pod_default_a3b78651-a061-11e5-a7fb-
06676ae2a427_10cc491a
a90c8d2d40ee        gcr.io/google_containers/pause:0.8.0    "/pause"
2 minutes ago       Up 2 minutes                             k8s_
POD.6d00e006_my-first-pod_default_a3b78651-a061-11e5-a7fb-06676ae2a427_
dfaf502a
```

You may notice that three containers – CentOS, nginx and pause – are running instead of two. Because each pod we need to keep belongs to a particular Linux namespace, if both the CentOS and nginx containers die, the namespace will also destroyed. Therefore, the pause container just remains in the pod to maintain Linux namespaces.

Let's launch a second pod, rename it as `my-second-pod` and run the `kubectl create` command as follows:

```
//just replace the name from my-first-pod to my-second-pod


# cat my-first-pod.yaml | sed -e 's/my-first-pod/my-second-pod/' > my-
second.pod.yaml


# cat my-second.pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-second-pod
spec:
  containers:
  - name: my-nginx
    image: nginx
  - name: my-centos
    image: centos
    command: ["/bin/sh", "-c", "while : ;do curl http://localhost:80/;
sleep 3; done"]


# kubectl create -f my-second.pod.yaml
pod "my-second-pod" created


# kubectl get pods
NAME            READY     STATUS     RESTARTS    AGE
my-first-pod    2/2       Running    0           49m
my-second-pod   2/2       Running    0           5m
```

If you have two or more nodes, `my-second-pod` was probably launched by another node, because the Kubernetes scheduler chose the most suitable node.

[Note that, if you would like to deploy more of the same pod, consider using a replication controller instead.]

After your testing, you can run the `kubectl delete` command to delete your pod from the Kubernetes cluster:

```
//running both my-first-pod and my-second-pod
# kubectl get pods
NAME             READY     STATUS     RESTARTS    AGE
my-first-pod     2/2       Running    0           49m
my-second-pod    2/2       Running    0           5m


//delete my-second-pod
# kubectl delete pod my-second-pod
pod "my-second-pod" deleted
# kubectl get pods
NAME            READY     STATUS     RESTARTS    AGE
my-first-pod    2/2       Running    0           54m


//delete my-first-pod
# kubectl delete pod my-first-pod
pod "my-first-pod" deleted
# kubectl get pods
NAME      READY     STATUS     RESTARTS    AGE
```

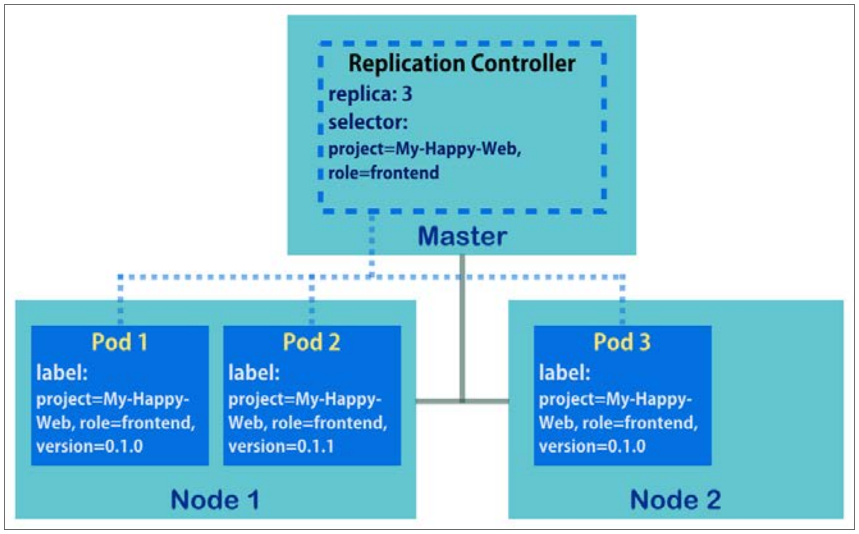
See also

This recipe described how to control pods. It is the basic component and operation of Kubernetes. The following recipes will describe advanced operation of pods using a replication controller, services and so on:

- *Working with a replication controller*
- *Working with services*
- *Working with labels and selectors*

Working with a replication controller

A replication controller is a term for API objects in Kubernetes that refers to pod replicas. The idea is to be able to control a set of pods' behaviors. The replication controller ensures that the pods, in a user-specified number, are running all the time. If some pods in the replication controller crash and terminate, the system will recreate pods with the original configurations on healthy nodes automatically, and keep a certain amount of processes continuously running. According to this feature, no matter whether you need replicas of pods or not, you can always shield the pods with the replication controller for autorecovery. In this recipe, you're going to learn how to manage your pods by using the replication controller:


Replication Controller
replica: 3
selector:
project=My-Happy-Web,
role=frontend
Master
Pod 1
label:
project=My-Happy-Web, role=frontend, version=0.1.0
Pod 2
label:
project=My-Happy-Web, role=frontend, version=0.1.1
Pod 3
label:
project=My-Happy-Web, role=frontend, version=0.1.0
Node 1
Node 2

The replication controller usually handles a tier of applications. As you see in the preceding image, we launch a replication controller with three pod replicas. Some mechanism details are listed as follows:

- The daemon in the master is called the controller manager and helps to maintain the resource running in its desired state. For example, the desired state of the replication controller in the previous image is three pod replicas.
- The daemon scheduler in the master takes charge of assigning tasks to healthy nodes.
- The selector of the replication controller is used for deciding which pods it covers. If the key-value pairs in the pod's label include all items in the selector of the replication controller, this pod belongs to this replication controller. As you will see, the previous image shows three pods are under the charge of the replication controller. Since the selector is covered and labelled **project** and **role**, the pod with the different minor version number, (**Pod 2)**, could still be one of the members.

Getting ready

We demonstrated the management of the replication controller in the Kubernetes master, when we installed the Kubernetes client package. Please login to the master first and make sure your environment is able to create replication controllers.

The evaluation of replication controller creation from the master

You can verify whether your Kubernetes master is a practical one through checking the following items:

Check whether the daemons are running or not. There should be three working daemon processes on the master node: `apiserver`, `scheduler` and `controller-manager`.

Check the command `kubectl` exists and is workable. Try the command `kubectl get componentstatuses` or `kubectl get cs`, so you can verify not only the components' status but also the feasibility of `kubectl`.

Check the nodes are ready to work. You can check them by the command `kubectl get nodes` for their status.

In case some of the preceding items are invalid, please refer to *Chapter 1*, *Building Your Own Kubernetes* for proper guidelines of the installation.

How to do it...

A replication controller can be created directly through CLI, or through a template file. We will express the former solution in this section. For template one, please refer to the recipe *Working with configuration files* in *Chapter 3*, *Playing with Containers*.

Creating a replication controller

To create replication controllers, we use the subcommand `run` after `kubectl`. The basic command pattern looks like the following:

```
//  kubectl run <REPLICATION CONTROLLER NAME> --images=<IMAGE NAME>
[OPTIONAL_FLAGS]
# kubectl run my-first-rc --image=nginx
CONTROLLER     CONTAINER(S)   IMAGE(S)   SELECTOR       REPLICAS
my-first-rc    my-first-rc    nginx      run=my-first-rc   1
```

This simple command is creating a replication controller by image nginx from the Docker Hub (`https://hub.docker.com`). The name, `my-first-rc`, must be unique in all replication controllers. Without specified number of replicas, the system will only build one pod as its default value. There are more optional flags you may integrate together to create a qualified replication controller:

--Flag=[Default Value]	**Description**	**Example**
`--replicas=1`	The number of pod replicas	`--replicas=3`
`--port=-1`	The exposed port of the container	`--port=80`
`--hostport=-1`	The host port mapping for the container port. Make sure that if using the flag, you do not run multiple pod replicas on a node in order to avoid port conflicts.	`--hostport=8080`
`--labels=""`	Key-value pairs as the labels for pods. Separate each pair with a comma.	`--labels="ProductName=HappyCloud,ProductionState=staging,ProjectOwner=Amy"`
`--command[=false]`	After the container boots up, run a different command through this flag. Two dashed lines append the flag as segmentation.	`--command -- /myapp/run.py -o logfile`
`--env=[]`	Set environment variables in containers. This flag could be used multiple times in a single command.	`--env="USERNAME=amy"` `--env="PASSWD=pa$$w0rd"`
`--overrides=""`	Use this flag to override some generated objects of the system in JSON format. The JSON value uses a single quote to escape the double quote. The field `apiVersion` is necessary. For detailed items and requirements, please refer to the *Working with configuration files* recipe in *Chapter 3*, *Playing with Containers*.	`--overrides='{"apiVersion": "v1"}'`

--Flag=[Default Value]	Description	Example
`--limits=""`	The upper limit of resource usage in the container. You can specify how much CPU and memory could be used. The unit of CPU is the number of cores in the format `NUMBERm`. `m` indicates `milli` (10-3). The unit of memory is byte. Please also check `--requests`.	`--limits="cpu=1000m, memory=512Mi"`
`--requests=""`	The minimum requirement of resource for container. The rule of value is the same as `--limits`.	`--requests="cpu=250m, memory=256Mi"`
`--dry-run[=false]`	Display object configuration without sending it out for being created.	`--dry-run`
`--attach[=false]`	For the replication controller, your terminal will attach to one of the replicas, and view the runtime log coming from the program. The default is to attach the first container in the pod, in the same way as Dockers attach. Some logs from the system show the container's pending status.	`--attach`
`-i , --stdin[=false]`	Enable the interactive mode of the container. The replica must be 1.	`-i`
`--tty=[false]`	Allocate a `tty` (new controlling terminal) to each container. You must enable the interactive mode by attaching the flag `-i` or `--stdin`.	`--tty`

The subcommand `run` will create a replication controller by default because of the flag `--restart`, which is preset as `Always`, meaning that the generated objects will always be triggered and run to meet the desired numbers of the replication controller.

For example, you can launch a replication controller, and then add new features or modify configurations:

```
// Run a replication controller with some flags for Nginx nodes. Try to
verify the settings with "--dry-run" first!
# kubectl run nginx-rc-test --image=nginx --labels="Owner=Amy,ProductionS
tate=test" --replicas=2 --port=80 --dry-run
CONTROLLER      CONTAINER(S)    IMAGE(S)    SELECTOR
REPLICAS
```

```
nginx-rc-test   nginx-rc-test   nginx      Owner=Amy,ProductionState=te
st    2


// Send out the request
# kubectl run nginx-rc --image=nginx --labels="Owner=Amy,ProductionState=
test" --replicas=2 --port=80


// Try to override container name while generating, which name is the
same as the name of replication controller in default
# kubectl run nginx-rc-override --image=nginx --overrides='{"apiVer
sion":"v1","spec":{"template":{"spec": {"containers":[{"name": "k8s-
nginx","image":"nginx"}]}}}}'
CONTROLLER        CONTAINER(S)   IMAGE(S)   SELECTOR             REPLICAS
nginx-rc-override   k8s-nginx      nginx      run=nginx-rc-override   1
//Interact with container after create a pod in replication controller
# kubectl run nginx-bash --image=nginx --tty -i --command -- /bin/bash
Waiting for pod to be scheduled
Waiting for pod default/nginx-bash-916y6 to be running, status is
Running, pod ready: false
Waiting for pod default/nginx-bash-916y6 to be running, status is
Running, pod ready: false
ls
bin   dev  home  lib64    mnt  proc  run  srv  tmp  var
boot  etc  lib  media  opt  root  sbin  sys  usr
root@nginx-bash-916y6:/#
```

Getting information of a replication controller

After we create a replication controller, the subcommand `get` and `describe` can help us to capture the information and pod status. In the CLI of Kubernetes, we usually use the abbreviation `rc` for resource type, instead of the full name replication controller:

First, we can check any replication controller in the system:

```
// Use subcommand "get" to list replication controllers
# kubectl get rc
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR         REPLICAS   AGE
check-rc-1   check-rc-1     nginx      run=check-rc-1   5          7m
check-rc-2   check-rc-2     nginx      app=nginx        2          6m
```

As it displays, the special column items are `SELECTOR` and `REPLICAS`. The selector must be the pods' labels, which indicate that the pods are controlled by this replication controller. We may specify the selector by the flag `--labels` while creating the replication controller with `kubectl run`. The default selector assigned to the replication controller, created by CLI, is in the form of `run=<REPLICATION CONTROLLER NAME>`:

```
// We can also get status of pod through selector/labels
# kubectl get pod -l app=nginx
NAME                READY     STATUS    RESTARTS   AGE
check-rc-2-95851    1/1       Running   0          6m
check-rc-2-mjezz    1/1       Running   0          6m
```

Furthermore, the subcommand `describe` helps users to get detailed items and logs of the resources:

```
# kubectl describe rc check-rc-2
Name:     check-rc-2
Namespace:   default
Image(s):   nginx
Selector:   app=nginx
Labels:     app=nginx
Replicas:   2 current / 2 desired
Pods Status:   2 Running / 0 Waiting / 0 Succeeded / 0 Failed
No volumes.
Events:
  FirstSeen  LastSeen  Count  From         SubobjectPath  Reason
Message
  ─────────  ────────  ─────  ────         ─────────────  ──────
───────

  6m       6m     1  {replication-controller }      SuccessfulCreate
Created pod: check-rc-2-95851
  6m     6m    1  {replication-controller }      SuccessfulCreate  Created
pod: check-rc-2-mjezz
```

Changing the configuration of a replication controller

The subcommands known as `edit`, `patch` and `replace` can help to update live replication controllers. All these three change the settings by way of a configuration file. Here we just take `edit` for example.

The subcommand `edit` lets users modify resource configuration through the editor. Try to update your replication controller through the command `kubectl edit rc/<REPLICATION CONTROLLER NAME>` (to change to another resource type, you can change `rc`, for example, `po`, `svc`, `ns`), you will access this via the default editor with a YAML configuration file, except for resource type and resource name. Take a look at the *Working with configuration files* recipe in *Chapter 3*, *Playing with Containers* for reference, and try to change the other values:

```
// Try to update your by subcommand edit
# kubectl get rc
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR         REPLICAS   AGE
test-edit    test-edit      nginx      run=test-edit    1          5m
# kubectl edit rc/test-edit
replicationcontroller "test-edit" edited
# kubectl get rc
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR
REPLICAS   AGE
test-edit    nginx-rc       nginx      app=nginx,run=after-edit   3
7m
```

Removing a replication controller

In order to remove replication controllers from the system, you can rely on the subcommand `delete`. The similar subcommand `stop` is deprecated and covered by `delete`, so we just introduce `delete` here. While we use `delete` to remove the resource, it removes the target objects forcefully and ignores any requests for the target objects at the same time:

```
// A replication controller that we want to remove has 5 replicas
# kubectl get rc
CONTROLLER    CONTAINER(S)   IMAGE(S)   SELECTOR          REPLICAS   AGE
test-delete   test-delete    nginx      run=test-delete   5          19s
# kubectl get pod
NAME                 READY     STATUS    RESTARTS   AGE
test-delete-g4xyy    1/1       Running   0          34s
test-delete-px9z6    1/1       Running   0          34s
test-delete-vctnk    1/1       Running   0          34s
test-delete-vsikc    1/1       Running   0          34s
test-delete-ye07h    1/1       Running   0          34s
// timing the response of "delete" and check the state of pod directly
# time kubectl delete rc test-delete && kubectl get pod
replicationcontroller "test-delete" deleted
real   0m2.028s
user   0m0.014s
sys   0m0.007s
```

```
NAME                READY     STATUS        RESTARTS    AGE
test-delete-g4xyy   1/1       Terminating   0           1m
test-delete-px9z6   0/1       Terminating   0           1m
test-delete-vctnk   1/1       Terminating   0           1m
test-delete-vsikc   0/1       Terminating   0           1m
test-delete-ye07h   1/1       Terminating   0           1m
```

We find that the response time is quite short and the effect is also instantaneous.

Removing pods from the replication controller

It is impossible to remove or scale down the replication controller by deleting pods on it, because while a pod is removed, the replication controller is out of its desired status, and the controller manager will ask it to create another one. This concept is shown in the following commands:

```
// Check replication controller and pod first
# kubectl get rc,pod
CONTROLLER         CONTAINER(S)       IMAGE(S)    SELECTOR
REPLICAS    AGE
test-delete-pod    test-delete-pod    nginx       run=test-
delete-pod    3           12s
NAME                     READY     STATUS     RESTARTS
AGE
test-delete-pod-8hooh    1/1       Running    0
14s
test-delete-pod-jwthw    1/1       Running    0
14s
test-delete-pod-oxngk    1/1       Running    0
14s
// Remove certain pod and check pod status to see what
happen
# kubectl delete pod test-delete-pod-8hooh && kubectl get
pod
NAME                     READY     STATUS         RESTARTS
AGE
test-delete-pod-8hooh    0/1       Terminating    0
1m
test-delete-pod-8nryo    0/1       Running        0
3s
test-delete-pod-jwthw    1/1       Running        0
1m
test-delete-pod-oxngk    1/1       Running        0
1m
```

How it works...

The replication controller defines a set of pods by a pod template and labels. As you know from previous sections, the replication controller only manages the pods by their labels. It is possible that the pod template and the configuration of the pod, are different. And it also means that standalone pods can be added into a controller's group by label modification. According to the following commands and results, let's evaluate this concept on selectors and labels:

```
// Two pod existed in system already, they have the same label app=nginx
# kubectl get pod -L app -L owner
NAME       READY     STATUS    RESTARTS   AGE       APP       OWNER
web-app1   1/1       Running   0          6m        nginx     Amy
web-app2   1/1       Running   0          6m        nginx     Bob
```

Then, we create a three-pod replication controller with the selector `app=nginx`:

```
# kubectl run rc-wo-create-all --replicas=3 --image=nginx
--labels="app=nginx"
replicationcontroller "rc-wo-create-all" created
```

We can find that the replication controller meets the desired state of three pods but only needs to boot one pod. The pod `web-app1` and `web-app2` are now controlled by running `rc-wo-create-all`:

```
# kubectl get pod -L app -L owner
NAME                     READY     STATUS    RESTARTS   AGE       APP
OWNER
rc-wo-create-all-jojve   1/1       Running   0          5s        nginx
<none>
web-app1                 1/1       Running   0          7m        nginx
Amy
web-app2                 1/1       Running   0          7m        nginx
Bob


# kubectl describe rc rc-wo-create-all
Name:     rc-wo-create-all
Namespace:   default
Image(s):  nginx
Selector:  app=nginx
Labels:    app=nginx
Replicas:  3 current / 3 desired
```

```
Pods Status:  3 Running / 0 Waiting / 0 Succeeded / 0 Failed
No volumes.
Events:
  FirstSeen  LastSeen  Count  From         SubobjectPath  Reason
Message
  ─────────  ────────  ─────  ────         ─────────────  ──────
───────
  1m     1m     1  {replication-controller }      SuccessfulCreate  Created
pod: rc-wo-create-all-jojve
```

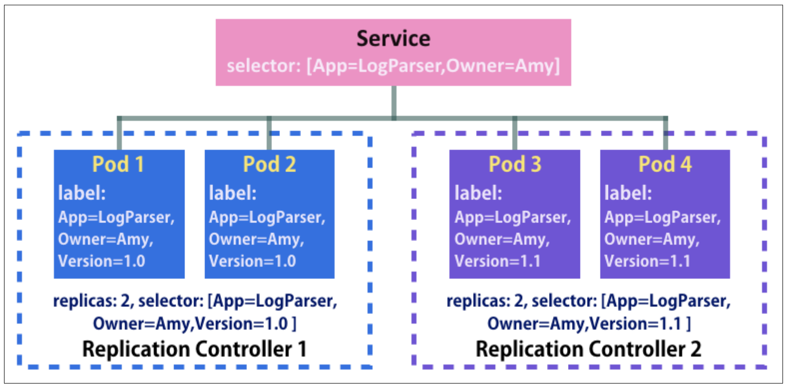
See also

In this chapter, there are some recipes for getting more ideas about the replication controller:

- *Working with pods*
- *Working with services*
- *Working with labels and selectors*
- *The Working with configuration files recipe in Chapter 3, Playing with Containers*

Working with services

The network service is an application that receives requests and provides a solution. Clients access the service by a network connection. They don't have to know the architecture of the service or how it runs. The only thing that clients have to verify is whether the end point of the service is contactable, and then follow its usage policy to solve problems. The Kubernetes service has similar ideas. It is not necessary to understand every pod before reaching their functionalities. For components outside the Kubernetes system, they just access the Kubernetes service with an exposed network port to communicate with running pods. It is not necessary to be aware of the containers' IPs and ports. Therefore, we can fulfill a zero downtime update for our container programs without struggling:



The preceding image shows the basic structure of the service and realizes the following concepts:

- As with the replication controller, the service directs the pods that have labels containing the service's selector. In other words, the pods selected by the service are based on their labels.
- The load of requests sent to the services will distribute to four pods.
- The replication controller ensures that the number of running pods meets its desired state. It monitors the pods for the service, making sure someone will take over duties from the service.

In this recipe, you will learn how to create services along with your pods.

Getting ready

Prior to applying services, it is important to verify whether all your nodes in the system are running `kube-proxy`. Daemon `kube-proxy` works as a network proxy in node. It helps to reflect service settings like IPs or ports on each node. To check whether `kube-proxy` is enabled or not, you can inspect the status of the daemon or search running processes on the node with a specific name:

```
// check the status of service kube-proxy
# service kube-proxy status
```

or

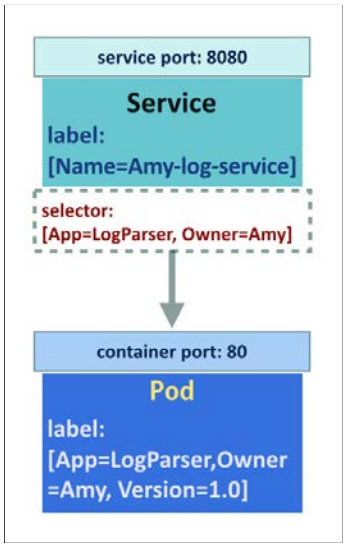
```
// Check processes on each node, and focus on kube-proxy
// grep "kube-proxy" or "hyperkube proxy"
# ps aux | grep "kube-proxy"
```

For demonstration in later sections, you can also install a private network environment on the master node. The daemons related to the network settings are `flanneld` and `kube-proxy`. It is easier for you to do the operation and verification on a single machine. Otherwise, please verify Kubernetes services on a node, which by default, has an internal network ready.

How to do it...

We can define and create a new Kubernetes service through the CLI or a configuration file. Here we are going to explain how to deploy the services by command. The subcommands `expose` and `describe` are utilized in the following commands for various scenarios. The version of Kubernetes we used in this recipe is 1.1.3. For file-format creation, please go to *Working with configuration files* recipe in *Chapter 3*, *Playing with Containers* for a detailed discussion.

When creating services, there are two configurations with which we have to take care: one is the label, the other is the port. As the following image indicates, the service and pod have their own key-value pair labels and ports. Be assured to use correct tags for these settings:



To create a service like this one, push the following command:

```
# kubectl expose pod <POD_NAME> --labels="Name=Amy-log-service" --selecto
r="App=LogParser,Owner=Amy" --port=8080 --target-port=80
```

The `--labels` tag `expose` in the subcommand is for labeling the services with key-value pairs. It is used to mark the services. For defining the selector of the service, use tag `--selector`. Without setting the selector for service, the selector would be the same as the labels of resource. In the preceding image, the selector would have an addition label: **Version=1.0**.

To expose the service port, we send out a port number with the tag `--port` in the subcommand `expose`. The service will take the container port number as its exposed port if no specific number is assigned. On the other hand, the tag `--target-port` points out the container port for service. While the target port is different from the exposed port of the container, users will get an empty response. At the same time, if we only assign the service port, the target port will copy it. Taking the previous image as an example, the traffic will be directed to container port `8080` supposing we don't use the tag `--target-port`, which brings out a refused connection error.

Creating services for different resources

You can attach a service to a pod, a replication controller and an endpoint outside the Kubernetes system, or even another service. We will show you these, one by one, in the next pages. The service creation is in the format: `kubectl expose RESOURCE_TYPE RESOURCE_NAME [TAGS]` or `kubectl expose -f CONFIGURATION_FILE [TAGS]`. Simply put, the resource types pod, replication controller and service are supported by the subcommand `expose`. So is the configuration file which follows the type limitation.

Creating a service for a pod

The pods shielded by the service need to contain labels, because the service takes this as a necessary condition based on the selector:

```
// Create a pod, and add labels to it for the selector of service.
# kubectl run nginx-pod --image=nginx --port=80 --restart="Never"
--labels="app=nginx"
pod "nginx-pod" created
# kubectl expose pod nginx-pod --port=8000 --target-port=80
--name="service-pod"
service "service-pod" exposed
```

The abbreviation of Kubernetes resources

While managing resources through CLI, you can type their abbreviations instead of the full names to save time and avoid typing errors.

Resource type	Abbreviated alias
Componentstatuses	`cs`
Events	`ev`
Endpoints	`ep`
Horizontalpodautoscaler	`hpa`
Limitranges	`limits`
Nodes	`no`
Namespaces	`ns`
Pods	`po`
Persistentvolumes	`pv`
Persistentvolumesclaims	`pvc`
Resourcequotas	`qotas`
Replicationcontrollers	`rc`
Services	`svc`
Ingress	`ing`

```
// "svc" is the abbreviation of service
# kubectl get svc service-pod
NAME          CLUSTER_IP        EXTERNAL_IP   PORT(S)    SELECTOR    AGE
service-pod   192.168.195.195   <none>        8000/TCP   app=nginx   11s
```

As you see in these commands, we open a service with port `8000` exposed. The reason why we specify the container port is so that the service doesn't take `8000` as the container port, by default. To verify whether the service is workable or not, go ahead with the following command in an internal network environment (which has been installed with the Kubernetes cluster CIDR).

```
// accessing by services CLUSTER_IP and PORT
# curl 192.168.195.195:8000
```

Creating a service for the replication controller and adding an external IP

A replication controller is the ideal resource type for a service. For pods supervised by the replication controller, the Kubernetes system has a controller manager to look over the lifecycle of them. It is also helpful for updating the version or state of program by binding existing services to another replication controller:

```
// Create a replication controller with container port 80 exposed
# kubectl run nginx-rc --image=nginx --port=80 --replicas=2
replicationcontroller "nginx-rc" created
# kubectl expose rc nginx-rc --name="service-rc" --external-ip="<USER_
SPECIFIED_IP>"
service "service-rc" exposed
```

In this case, we can provide the service with another IP address, which doesn't need to be inside the cluster network. The tag `--external-ip` of the subcommand `expose` can realize this static IP requirement. Be aware that the user-specified IP address could be contacted, for example, with the master node public IP:

```
// EXTERNAL_IP has Value shown on
# kubectl get svc service-rc
NAME         CLUSTER_IP      EXTERNAL_IP          PORT(S)     SELECTOR
AGE
service-rc   192.168.126.5   <USER_SPECIFIED_IP>  80/TCP     run=nginx-rc
4s
```

Now, you can verify the service by `192.168.126.5:80` or `<USER_SPECIFIED_IP>:80`:

```
// Take a look of service in details
# kubectl describe svc service-rc
Name:       service-rc
Namespace:     default
Labels:       run=nginx-rc
Selector:     run=nginx-rc
Type:       ClusterIP
IP:       192.168.126.5
Port:       <unnamed>  80/TCP
Endpoints:     192.168.45.3:80,192.168.47.2:80
Session Affinity:  None
No events.
```

You will find that the label and selector of a service is the default of the replication controller. In addition, there are multiple endpoints, which are replicas of the replication controller, available for dealing with requests from the service.

Creating a no-selector service for an endpoint

First, you should have an endpoint with an IP address. For example, we can generate an individual container in an instance, where it is located outside our Kubernetes system but is still contactable:

```
// Create an nginx server on another instance with IP address <FOREIGN_
IP>
# docker run -d -p 80:80 nginx
2a17909eca39a543ca46213839fc5f47c4b5c78083f0b067b2df334013f62002
# docker ps
CONTAINER ID        IMAGE                                   COMMAND
CREATED             STATUS              PORTS
NAMES
2a17909eca39        nginx                                   "nginx -g
'daemon off"   21 seconds ago      Up 20 seconds       0.0.0.0:80->80/
tcp, 443/tcp   goofy_brown
```

Then, in the master, we can create a Kubernetes endpoint by using the configuration file. The endpoint is named `service-foreign-ep`. We could configure multiple IP addresses and ports in the template:

```
# cat nginx-ep.json
{
    "kind": "Endpoints",
    "apiVersion": "v1",
    "metadata": {
        "name": "service-foreign-ep"
    },
    "subsets": [
        {
            "addresses": [
                { "ip": "<FOREIGN_IP>" }
            ],
            "ports": [
                { "port": 80 }
            ]
        }
    ]
}
# kubectl create -f nginx-ep.json
endpoints "service-foreign-ep" created
# kubectl get ep service-foreign-ep
NAME                    ENDPOINTS                         AGE
service-foreign-ep      <FOREIGN_IP>:80                   16s
```

As mentioned in the previous section, we can start a service for a resource-configured template with the subcommand `expose`. However, the CLI is unable to support exposing an endpoint in the file format:

```
// Give it a try!
# kubectl expose -f nginx-ep.json
error: invalid resource provided: Endpoints, only a replication
controller, service or pod is accepted
```

Therefore, we create the service through a configuration file:

```
# cat service-ep.json
{
    "kind": "Service",
    "apiVersion": "v1",
    "metadata": {
        "name": "service-foreign-ep"
    },
    "spec": {
        "ports": [
            {
                "protocol": "TCP",
                "port": 80,
                   "targetPort" : 80
            }
        ]
    }
}
```

The most important thing of all is that there is no selector defined in the template. This is quite reasonable since the endpoints are not in the Kubernetes system. The relationship between endpoints and service is built up by resource name. As you can see, the name of the service must be identical to the name of the endpoint:

```
# kubectl create -f service-ep.json
service "service-foreign-ep" created
// Check the details in service
# kubectl describe svc service-foreign-ep
Name:      service-ep
Namespace:    default
Labels:       <none>
Selector:     <none>
Type:      ClusterIP
IP:      192.168.234.21
Port:      <unnamed>  80/TCP
Endpoints:    <FOREIGN_IP>:80
Session Affinity:  None
No events.
```

Finally, the no-selector service is created for the external endpoint. Verify the result with `<FOREIGN_IP>:80`.

Creating a service with session affinity based on another service

Through the subcommand `expose`, we can also copy the settings of one service to another:

```
// Check the service we created for replication controller in previous
section
# kubectl describe svc service-rc
Name:       service-rc
Namespace:     default
Labels:       run=nginx-rc
Selector:     run=nginx-rc
Type:       ClusterIP
IP:       192.168.126.5
Port:       <unnamed>  80/TCP
Endpoints:     192.168.45.3:80,192.168.47.2:80
Session Affinity:  None
No events.
//Create a new service with different name and service port
# kubectl expose svc service-rc --port=8080 --target-port=80
--name=service-2nd --session-affinity="ClientIP"
service "service-2nd" exposed
```

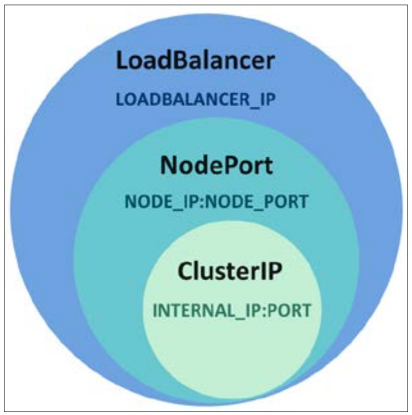
The new service named `service-2nd` is reset with service port `8080` and session affinity is enabled:

```
# kubectl describe svc service-2nd
Name:       service-2nd
Namespace:     default
Labels:       run=nginx-rc
Selector:     run=nginx-rc
Type:       ClusterIP
IP:       192.168.129.65
Port:       <unnamed>  8080/TCP
Endpoints:     192.168.45.3:80,192.168.47.2:80
Session Affinity:  ClientIP
No events.
```

Currently, the `ClientIP` is the only valued setting for the tag `--session-affinity`. While session affinity to the `ClientIP` is enabled, instead of round robin, the request of which endpoint the service should be sent to would be decided by the `ClientIP`. For example, if the requests from the client in the CIDR range `192.168.45.0/24` are sent to service `service-2nd`, they will be transferred to the endpoint `192.168.45.3:80`.

Creating a service in a different type

There are three types of service: **ClusterIP**, **NodePort** and **LoadBalancer**:



By default, every service is created as a ClusterIP type. The service in the ClusterIP type would be assigned an internal IP address randomly. For the NodePort type, it covers the ClusterIP's feature, and also allows the user to expose services on each node with the same port. The LoadBalancer is on the top of the other two types. The LoadBalancer service would be exposed internally and on the node. Besides this, if your cloud provider supports external load balancing servers, you can bind the load balancer IP to the service and this will become another exposing point.

Creating a service in NodePort type

Next, we are going to show you how to create a `NodePort` service. The tag `--type` in the subcommand `expose` helps to define the service type:

```
// Create a service with type NodePort, attaching to the replication
controller we created before
# kubectl expose rc nginx-rc --name=service-nodeport --type="NodePort"
service "service-nodeport" exposed
# kubectl describe svc service-nodeport
Name:       service-nodeport
Namespace:    default
Labels:      run=nginx-rc
```

```
Selector:    run=nginx-rc
Type:       NodePort
IP:       192.168.57.90
Port:       <unnamed>  80/TCP
NodePort:    <unnamed>  31841/TCP
Endpoints:    192.168.45.3:80,192.168.47.2:80
Session Affinity:  None
No events.
```

In the preceding case, the network port `31841` exposed on a node is randomly assigned by the system; the default port range is `30000` to `32767`. Notice that the port is exposed on every node in the system, so it is fine to access the service through `<NODE_IP>:31841`, for example, through the domain name of a node, like `kube-node1:31841`.

Deleting a service

You can simply work with the subcommand `delete` in cases where you want to stop a service:

```
# kubectl delete svc <SERVICE_NAME>
service "<SERVICE_NAME>" deleted
```

How it works...

The main actors in the Kubernetes system that perform the service environment are flanneld and kube-proxy. Daemon flanneld builds up a cluster network by allocating a subnet lease out of a preconfigured address space, and storing the network configuration in etcd, while kube-proxy directs the endpoints of services and pods.

See also

To get the best use of services, the following recipes are suggested to be read as well:

- *Working with a replication controller*
- *Working with labels and selectors*
- The *Working with configuration files* recipe in *Chapter 3*, *Playing with Containers*
- The *Moving monolithic to microservices* recipe in *Chapter 5*, *Building a Continuous Delivery Pipeline*

Working with volumes

Files in a container are ephemeral. When the container is terminated, the files are gone. Docker has introduced data volumes and data volume containers to help us manage the data by mounting from the host disk directory or from other containers. However, when it comes to a container cluster, it is hard to manage volumes across hosts and their lifetime by using Docker.

Kubernetes introduces volume, which lives with a pod across container restarts. It supports the following different types of network disks:

- emptyDir
- hostPath
- nfs
- iscsi
- flocker
- glusterfs
- rbd
- gitRepo
- awsElasticBlockStore
- gcePersistentDisk
- secret
- downwardAPI

In this section, we'll walk through the details of emptyDir, hostPath, nfs and glusterfs. Secret, which is used to store credentials, will be introduced in the next section. Most of them have similar Kubernetes syntax with a different backend.

Getting ready

The storage providers are required when you start to use volume in Kubernetes except for emptyDir, which will be erased when the pod is removed. For other storage providers, folders, servers or clusters have to be built before using them in the pod definition.

Different volume types have different storage providers:

Volume Type	Storage Provider
`emptyDir`	Local host
`hostPath`	Local host
`nfs`	NFS server
`iscsi`	iSCSI target provider
`flocker`	Flocker cluster
`glusterfs`	GlusterFS cluster
`rbd`	Ceph cluster
`gitRepo`	Git repository
`awsElasticBlockStore`	AWS EBS
`gcePersistentDisk`	GCE persistent disk
`secret`	Kubernetes configuration file
`downwardAPI`	Kubernetes pod information

How to do it...

Volumes are defined in the volumes section of the pod definition with a unique name. Each type of volume has a different configuration to be set. Once you define the volumes, you can mount them in the `volumeMounts` section in container `spec`. `volumeMounts.name` and `volumeMounts.mountPath` are required, which indicate the name of the volumes you defined and the mount path inside the container.

We'll use the Kubernetes configuration file with the YAML format to create a pod with volumes in the following examples.

emptyDir

`emptyDir` is the simplest volume type, which will create an empty volume for containers in the same pod to share. When the pod is removed, the files in `emptyDir` will be erased as well. `emptyDir` is created when a pod is created. In the following configuration file, we'll create a pod running Ubuntu with commands to sleep for 3600 seconds. As you can see, one volume is defined in the volumes section with name `data`, and the volumes will be mounted under `/data-mount` path in the Ubuntu container:

```
// configuration file of emptyDir volume
# cat emptyDir.yaml
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu
labels:
  name: ubuntu
spec:
  containers:
    -
      image: ubuntu
      command:
        - sleep
        - "3600"
      imagePullPolicy: IfNotPresent
      name: ubuntu
      volumeMounts:
        -
          mountPath: /data-mount
          name: data
  volumes:
    -
      name: data
      emptyDir: {}

// create pod by configuration file emptyDir.yaml
# kubectl create -f emptyDir.yaml
```

Check which node the pod is running on

By using the `kubectl describe pod <Pod name> | grep Node` command you could check which node the pod is running on.

After the pod is running, you could use `docker inspect <container ID>` on the target node and you could see the detailed mount points inside your container:

```
    "Mounts": [
        {
            "Source": "/var/lib/kubelet/pods/<id>/volumes/kubernetes.
io~empty-dir/data",
            "Destination": "/data-mount",
            "Mode": "",
            "RW": true
        },
      ...
    ]
```

Here, you can see Kubernetes simply create an empty folder with the path `/var/lib/kubelet/pods/<id>/volumes/kubernetes.io~empty-dir/<volumeMount name>` for the pod to use. If you create a pod with more than one container, all of them will mount the same destination `/data-mount` with the same source.

`emptyDir` could be mounted as `tmpfs` if we set the `emptyDir.medium` setting to `Memory` in the previous configuration file `emptyDir.yaml`:

```
  volumes:
    -
      name: data
      emptyDir:
        medium: Memory
```

We could also check the `Volumes` information by `kubectl describe pods ubuntu` to see whether it's set successfully:

```
# kubectl describe pods ubuntu
Name:          ubuntu
Namespace:        default
Image(s):       ubuntu
Node:          ip-10-96-219-192/
Status:          Running
...
Volumes:
  data:
    Type:  EmptyDir (a temporary directory that shares a pod's lifetime)
    Medium:  Memory
```

hostPath

`hostPath` acts as data volume in Docker. The local folder on a node listed in `hostPath` will be mounted into the pod. Since the pod can run on any nodes, read/write functions happening in the volume could explicitly exist in the node on which the pod is running. In Kubernetes, however, the pod should not be node-aware. Please note that configuration and files might be different on different nodes when using `hostPath`. Therefore, the same pod, created by same command or configuration file, might act differently on different nodes.

By using `hostPath`, you're able to read and write the files between containers and local host disks of nodes. What we need for volume definition is for `hostPath.path` to specify the target mounted folder on the node:

```
// configuration file of hostPath volume
# cat hostPath.yaml
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu
spec:
  containers:
    -
      image: ubuntu
      command:
        - sleep
        - "3600"
      imagePullPolicy: IfNotPresent
      name: ubuntu
      volumeMounts:
        -
          mountPath: /data-mount
          name: data
  volumes:
    -
      name: data
      hostPath:
        path: /target/path/on/host
```

Using `docker inspect` to check the volume details, you will see the volume on the host is mounted in `/data-mount` destination:

```
    "Mounts": [
        {
            "Source": "/target/path/on/host",
            "Destination": "/data-mount",
            "Mode": "",
            "RW": true
        },
      ...
    ]
```

Touching a file to validate that the volume is mounted successfully

Using `kubectl exec <pod name> <command>` you could run the command inside a pod. In this case, if we run `kubectl exec ubuntu touch /data-mount/sample`, we should be able to see one empty file named `sample` under `/target/path/on/host`.

nfs

You can mount the **Network File System** (**NFS**) to your pod as a `nfs` volume. Multiple pods can mount and share the files in the same `nfs` volume. The data stored in the `nfs` volume will be persistent across the pod's lifetime. You have to create your own NFS server before using `nfs` volume, and make sure that the `nfs-utils` package is installed on the Kubernetes nodes.

Checking that the nfs server works before you go

You should check out that the `/etc/exports` file has proper sharing parameters and directory, and is using the `mount -t nfs <nfs server>:<share name> <local mounted point>` command to check whether it could be mounted locally.

The configuration file of a volume type with `nfs` is similar to others, but the `nfs.server` and `nfs.path` are required in the volume definition to specify NFS server information, and the path mounting from. `nfs.readOnly` is an optional field for specifying whether the volume is read-only or not (default is false):

```
// configuration file of nfs volume
# cat nfs.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nfs
spec:
  containers:
    -
      name: nfs
      image: ubuntu
      volumeMounts:
          - name: nfs
            mountPath: "/data-mount"
  volumes:
  - name: nfs
    nfs:
      server: <your nfs server>
      path: "/"
```

After you run `kubectl create -f nfs.yaml`, you can describe your pod by using `kubectl describe <pod name>` to check the mounting status. If it's mounted successfully, it should show conditions. It's ready if it shows `true` and the target `nfs` you have mounted:

```
Conditions:
  Type     Status
  Ready    True
Volumes:
  nfs:
    Type:  NFS (an NFS mount that lasts the lifetime of a pod)
    Server:  <your nfs server>
    Path:  /
    ReadOnly:  false
```

If we inspect the container by using the Docker command, you will see the volume information in the `Mounts` section:

```
    "Mounts": [
 {
            "Source": "/var/lib/kubelet/pods/<id>/volumes/kubernetes.
io~nfs/nfs",
            "Destination": "/data-mount",
            "Mode": "",
            "RW": true
        },
      ...
     ]
```

Actually, Kubernetes just mounts your `<nfs server>:<share name>` into `/var/lib/kubelet/pods/<id>/volumes/kubernetes.io~nfs/nfs` and then mounts it into a container as its destination in the `/data-mount`. You could also use `kubectl exec` to touch the file, as the previous tip mentions, to test whether it's perfectly mounted.

glusterfs

GlusterFS (`https://www.gluster.org`) is a scalable network-attached storage file system. The `glusterfs` volume type allows you mount the GlusterFS volume into your pod. Just like NFS volume, the data in the GlusterFS volume is persistent across the pod's lifetime. If the pod is terminated, the data is still accessible in the GlusterFS volume. You should build a GlusterFS system before using a GlusterFS volume.

Checking GlusterFS works before you go

By using `gluster volume info` on GlusterFS servers, you can see currently available volumes. By using `mount -t glusterfs <glusterfs server>:/<volume name> <local mounted point>` locally, you can check whether the GlusterFS system can be successfully mounted.

Since the volume replica in GlusterFS must be greater than 1, let's assume we have two replicas in servers `gfs1` and `gfs2` and the volume name is `gvol`.

First, we need to create an endpoint acting as a bridge for `gfs1` and `gfs2`:

```
# cat gfs-endpoint.yaml
kind: Endpoints
apiVersion: v1
metadata:
  name: glusterfs-cluster
subsets:
  -
    addresses:
      -
        ip: <gfs1 server ip>
    ports:
      -
        port: 1
  -
    addresses:
      -
```

```
        ip: <gfs2 server ip>
    ports:
      -
        port: 1

// create endpoints
# kubectl create -f gfs-endpoint.yaml
```

Then we could use `kubectl get endpoints` to check the endpoint is created properly:

```
# kubectl get endpoints
NAME                ENDPOINTS                          AGE
glusterfs-cluster   <gfs1>:1,<gfs2>:1                  12m
```

After that, we should be able to create the pod with the GlusterFS volume by `glusterfs.yaml`. The parameters of the `glusterfs` volume definition are `glusterfs.endpoints`, which specify the endpoint name we just created, and the `glusterfs.path` which is the volume name `gvol`. `glusterfs.readOnly` and is used to set whether the volume is mounted in read-only mode:

```
# cat glusterfs.yaml
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu
spec:
  containers:
    -
      image: ubuntu
      command:
        - sleep
        - "3600"
      imagePullPolicy: IfNotPresent
      name: ubuntu
      volumeMounts:
        -
          mountPath: /data-mount
          name: data
  volumes:
    -
      name: data
      glusterfs:
        endpoints: glusterfs-cluster
        path: gvol
```

Let's check the volume setting by `kubectl describe`:

```
Volumes:
  data:
    Type:    Glusterfs (a Glusterfs mount on the host that shares a pod's
lifetime)
    EndpointsName:  glusterfs-cluster
    Path:    gvol
    ReadOnly:    false
```

Using `docker inspect` you should be able to see the mounted source is `/var/lib/kubelet/pods/<id>/volumes/kubernetes.io~glusterfs/data` to the destination `/data-mount`.

iscsi

The `iscsi` volume is used to mount the existing iSCSI to your pod. Unlike `nfs` volume, the `iscsi` volume is only allowed to be mounted in a single container in read-write mode. The data will be persisted across the pod's lifecycle:

Field Name	Field Definition
`targetPortal`	IP Address of iSCSI target portal
`Iqn`	IQN of the target portal
`Lun`	Target LUN for mounting
`fsType`	File system type on LUN
`readOnly`	Specify read-only or not, default is false

flocker

Flocker is an open-source container data volume manager. The `flocker` volume will be moved to the target node when the container moves. Before using Flocker with Kubernetes, the Flocker cluster (Flocker control service, Flocker dataset agent, Flocker container agent) is required. Flocker's official website (`https://docs.clusterhq.com/en/1.8.0/install/index.html`) has detailed installation instructions.

After you get your Flocker cluster ready, create a dataset and specify the dataset name in the Flocker volume definition in the Kubernetes configuration file:

Field Name	**Field Definition**
`datasetName`	Target dataset name in Flocker

rbd

Ceph RADOS Block Device (`http://docs.ceph.com/docs/master/rbd/rbd/`) could be mounted into your pod by using rbd volume. You need to install Ceph before using the rbd volume. The definition of rbd volume support is secret in order to keep authentication secrets:

Field Name	**Field Definition**	**Default Value**
`monitors`	Cepth monitors	
`pool`	The name of RADOS pool	`rbd`
`image`	The image name rbd created	
`user`	RADOS user name	`admin`
`keyring`	The path of keyring, will be overwritten if secret name is provided	`/etc/ceph/keyring`
`secretName`	Secret name	
`fsType`	File system type	
`readOnly`	Specify read-only or not	`False`

gitRepo

The `gitRepo` volume will mount as an empty dictionary and Git clone a repository with certain revision in a pod for you to use:

Field Name	**Field Definition**
`repository`	Your Git repository with SSH or HTTPS
`Revision`	The revision of repository
`readOnly`	Specify read-only or not

awsElasticBlockStore

awsElasticBlockStore volume mounts an AWS EBS volume into a pod. In order to use it, you have to have your pod running on AWS EC2 with the same availability zone with EBS. For now, EBS only supports attaching to an EC2 in nature, so it means you cannot attach a single EBS volume to multiple EC2 instances:

Field Name	Field Definition
volumeID	EBS volume info - aws://<availability-zone>/<volume-id>
fsType	File system type
readOnly	Specify read-only or not

gcePersistentDisk

Similar to awsElasticBlockStore, the pod using the gcePersistentDisk volume must be running on GCE with the same project and zone. The gcePersistentDisk supports only a single writer when readOnly = false:

Field Name	Field Definition
pdName	GCE persistent disk name
fsType	File system type
readOnly	Specify read-only or not

downwardAPI

The downwardAPI volume is a Kubernetes volume plugin with the ability to save some pod information in a plain text file into a container. The current supporting metadata of the downwardAPI volume is:

- metadata.annotations
- metadata.namespace
- metadata.name
- metadata.labels

The definition of the downwardAPI is a list of items. An item contains a path and fieldRef. Kubernetes will then dump the specified metadata listed in the fieldRef to a file named path under mountPath and mount the <volume name> into the destination you specified:

```
          {
              "Source": "/var/lib/kubelet/pods/<id>/volumes/kubernetes.
io~downward-api/<volume name>",
              "Destination": "/tmp",
              "Mode": "",
              "RW": true
          }
```

For the IP of the pod, using the environment variable to propagate in the pod spec would be much easier:

```
spec:
  containers:
    - name: envsample-pod-info
      env:
        - name: MY_POD_IP
          valueFrom:
            fieldRef:
              fieldPath: status.podIP
```

For more examples, look at the sample folder in Kubernetes GitHub (`https://github.com/kubernetes/kubernetes/tree/master/docs/user-guide/downward-api`) which contains more examples for both environment variables and `downwardAPI` volume.

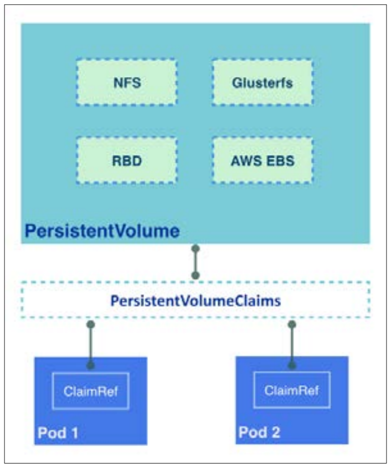
There's more...

In previous cases, the user needed to know the details of the storage provider. Kubernetes provides **PersistentVolume** (**PV**) to abstract the details of the storage provider and storage consumer. Kubernetes currently supports the PV types as follows:

- GCEPersistentDisk
- AWSElasticBlockStore
- NFS
- iSCSI
- RBD (Ceph Block Device)
- GlusterFS
- HostPath (not workable in multi-node cluster)

PersistentVolume

The illustration of persistent volume is shown in the following graph. At first, administrator provisions the specification of a `PersistentVolume`. Second, they provision consumer requests for storage by `PersistentVolumeClaim`. Finally, the pod mounts the volume by the reference of the `PersistentVolumeClaim`:



The administrator needs to provision and allocate the persistent volume first.

Here is an example using NFS:

```
// example of PV with NFS
# cat pv.yaml
  apiVersion: v1
  kind: PersistentVolume
  metadata:
    name: pvnfs01
  spec:
    capacity:
      storage: 3Gi
    accessModes:
      - ReadWriteOnce
    nfs:
      path: /
      server: <your nfs server>
    persistentVolumeReclaimPolicy: Recycle
```

```
// create the pv
# kubectl create -f pv.yaml
persistentvolume "pvnfs01" created
```

We can see there are three parameters here: `capacity`, `accessModes` and `persistentVolumeReclaimPolicy`. `capacity` is the size of this PV. `accessModes` is based on the capability of the storage provider, and can be set to a specific mode during provision. For example, NFS supports multiple readers and writers simultaneously, thus we could specify the `accessModes` as `ReadWriteOnce`, `ReadOnlyMany` or `ReadWriteMany`. The `accessModes` of one volume could be set to one mode at a time. `persistentVolumeReclaimPolicy` is used to define the behavior when PV is released. Currently, the supported policy is `Retain` and `Recycle` for `nfs` and `hostPath`. You have to clean the volume by yourself in `Retain` mode; on the other hand, Kubernetes will scrub the volume in `Recycle` mode.

PV is a resource like node. We could use `kubectl get pv` to see current provisioned PVs:

```
// list current PVs
# kubectl get pv
NAME       LABELS    CAPACITY   ACCESSMODES   STATUS    CLAIM
REASON     AGE
pvnfs01    <none>    3Gi        RWO           Bound     default/pvclaim01
37m
```

Next, we will need to bind `PersistentVolume` with `PersistentVolumeClaim` in order to mount it as a volume into the pod:

```
// example of PersistentVolumeClaim
# cat claim.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvclaim01
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi

// create the claim
# kubectl create -f claim.yaml
```

```
persistentvolumeclaim "pvclaim01" created

// list the PersistentVolumeClaim (pvc)
# kubectl get pvc
NAME        LABELS    STATUS    VOLUME    CAPACITY   ACCESSMODES   AGE
pvclaim01   <none>    Bound     pvnfs01   3Gi        RWO           59m
```

The constraints of `accessModes` and `storage` could be set in the `PersistentVolumeClaim`. If the claim is bound successfully, its status will turn to `Bound`; conversely, if the status is `Unbound`, it means that currently no PV matches the requests.

Then we are able to mount the PV as a volume by using `PersistentVolumeClaim`:

```
// example of mounting into Pod
# cat nginx.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    project: pilot
    environment: staging
    tier: frontend
spec:
  containers:
    -
      image: nginx
      imagePullPolicy: IfNotPresent
      name: nginx
      volumeMounts:
      - name: pv
        mountPath: "/usr/share/nginx/html"
      ports:
      - containerPort: 80
  volumes:
    - name: pv
      persistentVolumeClaim:
        claimName: "pvclaim01"
```

```
// create the pod
# kubectl create -f nginx.yaml
pod "nginx" created
```

The syntax is similar to the other volume type. Just add the `claimName` of the `persistentVolumeClaim` in the volume definition. We are all set! Let's check the details to see whether we have mounted it successfully:

```
// check the details of a pod
# kubectl describe pod nginx
...
Volumes:
  pv:
    Type:  PersistentVolumeClaim (a reference to a PersistentVolumeClaim
in the same namespace)
    ClaimName:  pvclaim01
    ReadOnly:  false
...
```

We can see that we have a volume mounted in the pod nginx with type `pv pvclaim01`. Use `docker inspect` to see how it is mounted:

```
     "Mounts": [
        {
            "Source": "/var/lib/kubelet/pods/<id>/volumes/kubernetes.
io~nfs/pvnfs01",
            "Destination": "/usr/share/nginx/html",
            "Mode": "",
            "RW": true
        },
      ...
    ]
```

Kubernetes mounts `/var/lib/kubelet/pods/<id>/volumes/kubernetes.io~nfs/<persistentvolume name>` into the destination in the pod.

See also

Volumes are put in container specs in pods or replication controllers. Check out the following recipes to jog your memory:

- *Working with pods*
- *Working with a replication controller*

Working with secrets

Kubernetes secrets manage information in key-value formats with the value encoded. With secrets, users don't have to set values in the configuration file or type them in CLI. When secrets are used properly, they can reduce the risk of credential leak and make our resource configurations more organized.

Currently, there are three types of secret:

- Opaque: `https://en.wikipedia.org/wiki/Opaque_data_type`
- Service account token
- Docker authentication

Opaque is the default type. We will put service account tokens and the authentication of Docker in the remark part.

Getting ready

Before using our credentials with secrets, some precautions must be taken First, secrets have a 1 MB size limitation. It works fine for defining several key-value pairs in a single secret. But, be aware that the total size should not exceed 1 MB. Next, secret acts like a volume for containers, so secrets should be created prior to dependent pods.

How to do it...

We can only generate secrets by using configuration files. In this recipe, we will deliver a simple template file and focus on the functionality. For various template designs, please take a look at the *Working with configuration files* recipe in *Chapter 3*, *Playing with Containers*.

Creating a secret

The configuration file of secrets contains secret type and data:

```
// A simple file for configuring secret
# cat secret-test.json
{
  "kind": "Secret",
  "apiVersion": "v1",
  "metadata": {
    "name": "secret-test"
  },
  "type": "Opaque",
  "data": {
    "username": "YW15Cg==",
    "password": " UGEkJHcwcmQhCg=="
  }
}
```

The secret type `Opaque` is the default one, which simply indicates that the data is not shown. The other types, service account token and Docker authentication, are applied when using the values `kubernetes.io/service-account-token` and `kubernetes.io/dockercfg` at the type item stage, respectively.

The data `username` and `password` are customized keys. Their corresponding values are base64-encoded string. You can get your encoded value through these pipe commands:

```
# echo "amy" | base64
YW15Cg==
```

The resource annotation and management of secrets is similar to other resource types. Feel free to create a secret and check its status by using common subcommands:

```
# kubectl create -f secret-test.json
secret "secret-test" created
# kubectl get secret
NAME           TYPE      DATA      AGE
secret-test    Opaque    2         39s
# kubectl describe secret secret-test
Name:     secret-test
Namespace:   default
Labels:     <none>
Annotations:   <none>

Type:   Opaque

Data
====
password:   10 bytes
username:   4 bytes
```

As you can see, although secret hides the information, we can get the amount of data, the data name and also the size of the value.

Picking up secret in the container

In order to let the pod get the secret information, secret data is mounted as a file in the container. The key-value pair data will be shown in plain-text file format, which takes a key name as the file name and the decoded value as the file content. Therefore, we create the pod by configuration file, where the container's mounting volume is pointed to secret:

```
# cat pod-secret.json
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "pod-with-secret"
  },
  "spec": {
    "volumes": [
      {
        "name": "secret-volume",
        "secret": {
          "secretName": "secret-test"
        }
      }
    ],
    "containers": [
      {
  "name": "secret-test-pod",
        "image": "nginx",
        "volumeMounts": [
          {
            "name": "secret-volume",
            "readOnly": true,
            "mountPath": "/tmp/secret-volume"
          }
        ]
      }
    ]
  }
}
```

For the previous template, we defined a volume called `secret-volume` which includes physical files with the content of the secret `secret-test`; the containers' mounting point is also defined along with the location, where to put secret files, and bound with `secret-volume`. In this case, the container could access secrets in its local file system by using `/tmp/secrets/<SECRET_KEY>`.

To verify the content is decrypted for the usage of the container program, let's take a look at the specific container on the node:

```
// login to node and enable bash process with new tty
# docker exec -it <CONTAINER_ID> bash
root@pod-with-secret:/# ls /tmp/secrets/
password  username
root@pod-with-secret:/# cat /tmp/secrets/password
Pa$$w0rd!
root@pod-with-secret:/# cat /tmp/secrets/username
amy
```

Deleting a secret

Secret, like other resources, can be stopped by the subcommand `delete`. Both methods, deleting according to configuration file or deleting by resource name are workable:

```
# kubectl delete -f secret-test.json
secret "secret-test" deleted
```

How it works...

In order to reduce the risk of leaking the secrets' content, the Kubernetes system never saves the data of secrets on disk. Instead, secrets are stored in the memory. For a more accurate statement, the Kubernetes API server pushes secret to the node on which the demanded container is running. The node stores the data in `tmpfs`, which will be flashed if the container is destroyed.

Go and check the node, which has container with secrets running on it:

```
// check the disk
df -h --type=tmpfs
Filesystem      Size  Used Avail Use% Mounted on
tmpfs           920M     0  920M   0% /dev/shm
tmpfs           920M   17M  903M   2% /run
tmpfs           920M     0  920M   0% /sys/fs/cgroup
tmpfs           184M     0  184M   0% /run/user/2007
tmpfs           920M  8.0K  920M   1% /var/lib/kubelet/pods/2edd4eb4-
b39e-11e5-9663-0200e755981f/volumes/kubernetes.io~secret/secret-volume
```

Furthermore, I suggest that you avoid creating a large-size secret or many small-size secrets. Since secrets are kept in the memory of nodes, reducing the total size of secrets could help to save resources and maintain good performance.

There's more...

In the previous sections, secret is configured in the default service account. The service account can make processes in containers in contact with the API server. You could have different authentication by creating different service accounts.

Let's see how many service accounts we currently have:

```
$ kubectl get serviceaccounts
NAME           SECRETS    AGE
default        0          18d
```

Kubernetes will create a default service account. Let's see how to create our own one:

```
# example of service account creation
$ cat serviceaccount.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: test-account

# create service account named test-account
$ kubectl create -f serviceaccount.yaml
serviceaccount "test-account" created
```

After creation, let's list the accounts by `kubectl`:

```
$ kubectl get serviceaccounts
NAME            SECRETS    AGE
default         0          18d
test-account    0          37s
```

We can see there is a new service account named `test-account` in the list now.

Each service account could have its own API token, image pull secrets and mountable secrets.

Similarly, we could delete the service account by using `kubectl`:

```
$ kubectl delete serviceaccount test-account
serviceaccount "test-account" deleted
```

On the other hand, Docker authentication can also be saved as a secret data for pulling images. We will discuss the usage in *Working with the private Docker registry* recipe in *Chapter 5*, *Building a Continuous Delivery Pipeline*.

See also

- The *Working with configuration files* recipe in *Chapter 3*, *Playing with Containers*
- The *Moving monolithic to microservices*, *Working with the private Docker registry* recipes in *Chapter 5*, *Building a Continuous Delivery Pipeline*
- The *Advanced settings in kubeconfig* recipe in *Chapter 7*, *Advanced Cluster Administration*

Working with names

When you create any Kubernetes objects such as a pod, replication controller and service, you can assign a name to it. The names in Kubernetes are spatially unique, which means you cannot assign the same name in the pods.

Getting ready

Kubernetes allows us to assign a name with the following restrictions:

- Up to 253 characters
- Lowercase of alphabet and numeric characters
- May contain special characters in the middle but only dash (-) and dot (.)

How to do it...

The following example is the pod definition that assigns the pod name as `my-pod`, to the container name as `my-container`, you can successfully create it as follows:

```
# cat my-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: nginx
```

```
# kubectl create -f my-pod.yaml
pod "my-pod" created

# kubectl get pods
NAME      READY     STATUS    RESTARTS   AGE
my-pod    0/1       Running   0          4s
```

You can use the `kubectl describe` command to see the container name `my-container` as follows:

```
# kubectl describe pod my-pod
Name:         my-pod
Namespace:       default
Image(s):       nginx
Node:         ip-10-96-219-25/10.96.219.25
Start Time:       Wed, 16 Dec 2015 00:46:33 +0000
Labels:         <none>
Status:         Running
Reason:
Message:
IP:         192.168.34.35
Replication Controllers:  <none>
Containers:
  my-container:
    Container ID:  docker://5501d115703e334ae44c1541b990a7e22ce4f310226ea
fea206594e4c85c90d9
    Image:    nginx
    Image ID:    docker://6ffc02088cb870652eca9ccd4c4fb582f75b29af2879792
ed09bb46fd1c898ef
    State:    Running
      Started:    Wed, 16 Dec 2015 00:46:34 +0000
    Ready:     True
    Restart Count:   0
    Environment Variables:
```

On the other hand, the following example contains two containers, but assigns the same name as `my-container`, therefore the `kubectl` command returns an error and can't create the pod.

```
//delete previous pods
# kubectl delete pods --all
```

```
pod "my-pod" deleted
# cat duplicate.yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: nginx
  - name: my-container
    image: centos
    command: ["/bin/sh", "-c", "while : ;do curl http://localhost:80/;
sleep 3; done"]
# kubectl create -f duplicate.yaml
The Pod "my-pod" is invalid.
spec.containers[1].name: duplicate value 'my-container'
```

You can add the `-validate` flag

For example: `kubectl create -f duplicate.yaml -validate`

Use a schema to validate the input before sending it

In another example, the YAML contains a replication controller and service, both of which are using the same name `my-nginx`, but it is successfully created because the replication controller and service are different:

```
# cat nginx.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx
spec:
  replicas: 2
  selector:
      sel : my-selector
  template:
    metadata:
        labels:
          sel : my-selector
    spec:
      containers:
      - name: my-container
        image: nginx
---
apiVersion: v1
kind: Service
metadata:
  name: my-nginx

spec:
  ports:
    - protocol: TCP
      port: 80
      nodePort: 30080
  type: NodePort
  selector:
     sel: my-selector

# kubectl create -f nginx.yaml
replicationcontroller "my-nginx" created
service "my-nginx" created

# kubectl get rc
CONTROLLER    CONTAINER(S)   IMAGE(S)    SELECTOR          REPLICAS    AGE
my-nginx      my-container   nginx       sel=my-selector   2           8s

# kubectl get service
NAME          CLUSTER_IP       EXTERNAL_IP   PORT(S)    SELECTOR
AGE
kubernetes    192.168.0.1      <none>        443/TCP    <none>
6d
my-nginx      192.168.38.134   nodes         80/TCP     sel=my-selector
14s
```

How it works...

Name is just a unique identifier, all naming conventions are good, however it is recommended to look up and identify the container image. For example:

- `memcached-pod1`
- `haproxy.us-west`
- `my-project1.mysql`

On the other hand, the following examples do not work because of Kubernetes restrictions:

- Memcache-pod1 (contains uppercase)
- haproxy.us_west (contains underscore)
- my-project1.mysql. (dot in the last)

Note that Kubernetes supports a label that allows to assign a key=value style identifier. It also allows duplication. Therefore, if you want to assign something like the information below, use a label instead:

- environment (for example: staging, production)
- version (for example: v1.2)
- application role (for example: frontend, worker)

In addition, Kubernetes also supports namespaces which have isolated namespaces. This means that you can use the same name in different namespaces (for example: nginx). Therefore, if you want to assign just an application name, use namespaces instead.

See also

This section described how to assign and find the name of objects. This is just a basic methodology, but Kubernetes has more powerful naming tools such as namespace and selectors to manage clusters:

- *Working with pods*
- *Working with a replication controller*
- *Working with services*
- *Working with namespaces*
- *Working with labels and selectors*

Working with namespaces

The name of a resource is a unique identifier within a namespace in the Kubernetes cluster. Using a Kubernetes namespace could isolate namespaces for different environments in the same cluster. It gives you the flexibility of creating an isolated environment and partitioning resources to different projects and teams.

Pods, services, replication controllers are contained in a certain namespace. Some resources, such as nodes and PVs, do not belong to any namespace.

Getting ready

By default, Kubernetes has created a namespace named `default`. All the objects created without specifying namespaces will be put into default namespaces. You could use `kubectl` to list namespaces:

```
// check all namespaces
# kubectl get namespaces
NAME       LABELS     STATUS     AGE
default    <none>     Active     8d
```

Kubernetes will also create another initial namespace called `kube-system` for locating Kubernetes system objects, such as a Kubernetes UI pod.

The name of a namespace must be a DNS label and follow the following rules:

- At most 63 characters
- Matching regex [a-z0-9]([-a-z0-9]*[a-z0-9])

How to do it...

1. After selecting our desired name, let's create a namespace named `new-namespace` by using the configuration file:

   ```
   # cat newNamespace.yaml
   apiVersion: v1
   kind: Namespace
   metadata:
     name: new-namespace

   // create the resource by kubectl
   # kubectl create -f newNamespace.yaml
   ```

2. After the namespace is created successfully, list the namespace again:

```
// list namespaces
# kubectl get namespaces
NAME             LABELS    STATUS    AGE
default          <none>    Active    8d
new-namespace    <none>    Active    12m
```

   You can see now that we have two namespaces.

3. Let's run the nginx replication controller described in *Chapter 1*, *Building Your Own Kubernetes* in a new namespace:

```
// run a nginx RC in namespace=new-namespace
# kubectl run nginx --image=nginx --namespace=new-namespace
```

4. Then let's list the pods:

```
# kubectl get pods
NAME                                        READY     STATUS
RESTARTS   AGE
```

5. There are no pods running! Let's run again with the `--namespace` parameter:

```
// to list pods in all namespaces
# kubectl get pods --all-namespaces
NAMESPACE      NAME          READY     STATUS    RESTARTS   AGE
new-namespace  nginx-ns0ig   1/1       Running   0          17m

// to get pods from new-namespace
# kubectl get pods --namespace=new-namespace
NAME          READY     STATUS    RESTARTS   AGE
nginx-ns0ig   1/1       Running   0          18m
```

   We can see our pods now.

6. By default, if you don't specify any namespace in the command line, Kubernetes will create the resources in the default namespace. If you want to create resources by configuration file, just simply specify it when doing `kubectl create`:

```
# kubectl create -f myResource.yaml --namespace=new-namespace
```

Changing the default namespace

It is possible to switch the default namespace in Kubernetes:

1. Find your current context:

   ```
   # kubectl config view | grep current-context
   current-context: ""
   ```

   It reveals that we don't have any context setting now.

2. No matter whether there is current context or not, using `set-context` could create a new one or overwrite the existing one:

   ```
   # kubectl config set-context <current context or new context name>
   --namespace=new-namespace
   ```

3. After setting the context with a new namespace, we can check the current configuration:

   ```
   # kubectl config view
   apiVersion: v1
   clusters: []
   contexts:
   - context:
       cluster: ""
       namespace: new-namespace
       user: ""
     name: new-context
   current-context: ""
   kind: Config
   preferences: {}
   users: []
   ```

   We can see the namespace is set properly in the contexts section.

4. Switch the context to the one we just created:

   ```
   # kubectl config use-context new-context
   ```

5. Then check the current context again:

   ```
   # kubectl config view | grep current-context
   current-context: new-context
   ```

   We can see that `current-context` is `new-context` now.

6. Let's list the current pods again. There's no need to specify the `Namespace` parameter, as we can list the pods in `new-namespace`:

   ```
   # kubectl get pods
   NAME           READY     STATUS    RESTARTS   AGE
   nginx-ns0ig    1/1       Running   0          54m
   ```

7. Namespace is listed in the pod description as well:

   ```
   # kubectl describe pod nginx-ns0ig
   Name:         nginx-ns0ig
   Namespace:       new-namespace
   Image(s):      nginx
   Node:         ip-10-96-219-156/10.96.219.156
   Start Time:      Sun, 20 Dec 2015 15:03:40 +0000
   Labels:        run=nginx
   Status:        Running
   ```

Deleting a namespace

1. Using `kubectl delete` could delete the resources including the namespace. Deleting a namespace will erase all the resources under that namespace:

   ```
   # kubectl delete namespaces new-namespace
   namespace "new-namespace" deleted
   ```

2. After the namespace is deleted, our nginx pod is gone:

   ```
   # kubectl get pods
   NAME      READY     STATUS    RESTARTS   AGE
   ```

3. However, the default namespace in the context is still set as `new-namespace`:

   ```
   # kubectl config view | grep current-context
   current-context: new-context
   ```

   Will it be a problem?

4. Let's run an `nginx` replication controller again.

   ```
   # kubectl run nginx --image=nginx
   Error from server: namespaces "new-namespace" not found
   ```

   It will try to create an `nginx` replication controller and replica pod in the current namespace we just deleted. Kubernetes will throw out an error if the namespace is not found.

5. Let's switch back to the default namespace.

   ```
   # kubectl config set-context new-context --namespace=""
   context "new-context" set.
   ```

6. Let's run an `nginx` again.

   ```
   # kubectl run nginx --image=nginx
   replicationcontroller "nginx" created
   Does it real run in default namespace? Let's describe the pod.
   # kubectl describe pods nginx-ymqeh
   Name:          nginx-ymqeh
   Namespace:        default
   Image(s):       nginx
   Node:          ip-10-96-219-156/10.96.219.156
   Start Time:       Sun, 20 Dec 2015 16:13:33 +0000
   Labels:         run=nginx
   Status:         Running
   ...
   ```

   We can see the pod is currently running in `Namespace: default`. Everything looks fine.

There's more…

Sometimes you'll need to limit the resource quota for each team by distinguishing the namespace. After you create a new namespace, the details look like this:

```
$ kubectl describe namespaces new-namespace
Name:   new-namespace
Labels:   <none>
Status:   Active

No resource quota.


No resource limits.
```

Resource quota and limits are not set by default. Kubernetes supports constraint for a container or pod. `LimitRanger` in the Kubernetes API server has to be enabled before setting the constraint. You could either use a command line or configuration file to enable it:

```
// using command line-
# kube-apiserver --admission-control=LimitRanger
```

```
// using configuration file
# cat /etc/kubernetes/apiserver
...
# default admission control policies
KUBE_ADMISSION_CONTROL="--admission_control=NamespaceLifecycle,NamespaceE
xists,LimitRanger,SecurityContextDeny,ResourceQuota"
...
```

The following is a good example for creating a limit in a namespace.

We will then limit the resources in a pod with the values `2` as `max` and `200m` as `min` for `cpu`, and `1Gi` as `max` and `6Mi` as `min` for `memory`. For the container, the `cpu` is limited between `100m` - `2` and the memory is between `3Mi` - `1Gi`. If the `max` is set, then you have to specify the limit in the pod/container spec during the resource creation; if the `min` is set then the request has to be specified during the pod/container creation. The `default` and `defaultRequest` section in `LimitRange` is used to specify the default limit and request in the container spec:

```
# cat limits.yaml
apiVersion: v1
kind: LimitRange
metadata:
  name: limits
  namespace: new-namespace
spec:
  limits:
  - max:
      cpu: "2"
      memory: 1Gi
    min:
      cpu: 200m
      memory: 6Mi
    type: Pod
  - default:
      cpu: 300m
      memory: 200Mi
    defaultRequest:
      cpu: 200m
      memory: 100Mi
    max:
      cpu: "2"
      memory: 1Gi
    min:
      cpu: 100m
      memory: 3Mi
    type: Container

// create LimitRange
# kubectl create -f limits.yaml
limitrange "limits" created
```

After the `LimitRange` is created, we can list these down just like with any other resource:

```
// list LimitRange
# kubectl get LimitRange --namespace=new-namespace
NAME      AGE
limits    22m
```

When you describe the new namespace you will now be able to see the constraint:

```
# kubectl describe namespace new-namespace
Name:  new-namespace
Labels:   <none>
Status:   Active

No resource quota.

Resource Limits
 Type       Resource  Min    Max    Request  Limit   Limit/Request
 ----       --------  ---     ---    -------  -----   -------------
 Pod       memory    6Mi      1Gi  -          -      -
 Pod       cpu       200m   2     -          -      -
 Container  cpu        100m   2     200m      300m   -
 Container  memory    3Mi      1Gi  100Mi      200Mi   -
```

All the pods and containers created in this namespace have to follow the resource limits listed here. If the definitions violate the rule, a validation error will be thrown accordingly.

Deleting LimitRange

We could delete the `LimitRange` resource via:

```
# kubectl delete LimitRange <limit name> --namespace=<namespace>
```

Here, the limit name is `limits` and the namespace is `new-namespace`. After that when you describe the namespace, the constraint is gone:

```
# kubectl describe namespace <namespace>
Name:   new-namespace
Labels:   <none>
Status:   Active


No resource quota.


No resource limits.
```

See also

Many resources are running under a namespace, check out the following recipes:

- *Working with pods*
- *Working with names*
- The *Setting resource in nodes* recipe in *Chapter 7*, *Advanced Cluster Administration*

Working with labels and selectors

Labels are a set of key/value pairs, which are attached to object metadata. We could use labels to select, organize and group objects, such as pods, replication controllers and services. Labels are not necessarily unique. Objects could carry the same set of labels.

Label selectors are used to query objects via labels. Current supported selector types are:

- Equality-based label selector
- Set-based label selector
- Empty label selector
- Null label selector

An equality-based label selector is a set of equality requirements, which could filter labels by equal or non-equal operation. A set-based label selector is used to filter labels by a set of values, and currently supports `in` and `notin` operators. When a label value matches the values in the `in` operator, it will be returned by the selector; conversely, if a label value does not match the values in the `notin` operator, it will be returned. Empty label selectors select all objects and null labels select no objects. Selectors are combinable. Kubernetes will return the objects that match all the requirements in selectors.

Getting ready

Before you get to set labels into the objects, you should consider the valid naming convention of key and value.

A valid key should follow these rules:

- A name with an optional prefix, separated by a slash.
- A prefix must be a DNS subdomain, separated by dots, no longer than 253 characters.
- A name must be less than 63 characters with the combination of [a-z0-9A-Z] and dashes, underscores and dots. Note that symbols are illegal if put at the beginning and the end.

A valid value should follow the following rules:

- A name must be less than 63 characters with the combination of [a-z0-9A-Z] and dashes, underscores and dots. Note that symbols are illegal if put at the beginning and the end.

You should also consider the purpose, too. For example, we have a service in the pilot project under different development environments which contain multiple tiers. Then we could make our labels:

- `project: pilot`
- `environment: development`, `environment: staging`, `environment: production`
- `tier: frontend`, `tier: backend`

How to do it...

Let's try to create an nginx pod with the previous labels in both a staging and production environment:

1. We will create the same staging for pod and production as that for the replication controller (RC):

```
# cat staging-nginx.yaml
apiVersion: v1
```

```
kind: Pod
metadata:
  name: nginx
  labels:
    project: pilot
    environment: staging
    tier: frontend
spec:
  containers:
    -
      image: nginx
      imagePullPolicy: IfNotPresent
      name: nginx
      ports:
      - containerPort: 80

// create the pod via configuration file
# kubectl create -f staging-nginx.yaml
pod "nginx" created
```

2. Let's see the details of the pod:

```
# kubectl describe pod nginx
Name:         nginx
Namespace:      default
Image(s):      nginx
Node:         ip-10-96-219-231/
Start Time:      Sun, 27 Dec 2015 18:12:31 +0000
Labels:        environment=staging,project=pilot,tier=frotend
Status:        Running
...
```

We could then see the labels in the pod description as `environment=staging,project=pilot,tier=frontend`.

Good. We have a staging pod now.

3. Now, get on with creating the RC for a production environment by using the command line:

   ```
   $ kubectl run nginx-prod --image=nginx --replicas=2 --port=80 --la
   bels="environment=production,project=pilot,tier=frontend"
   ```

   This will then create an RC named `nginx-prod` with two replicas, an opened port `80`, and with the labels `environment=production`,`project=pilot`,`tier=frontend`.

4. We can see that we currently have a total three pods here. One pod is created for staging, the other two are for production:

   ```
   # kubectl get pods
   NAME                 READY      STATUS     RESTARTS    AGE
   nginx                1/1        Running    0           8s
   nginx-prod-50345     1/1        Running    0           19s
   nginx-prod-pilb4     1/1        Running    0           19s
   ```

5. Let's get some filters for the selecting pods. For example, if I wanted to select production pods in the pilot project:

   ```
   # kubectl get pods -l "project=pilot,environment=production"
   NAME                 READY      STATUS     RESTARTS    AGE
   nginx-prod-50345     1/1        Running    0           9m
   nginx-prod-pilb4     1/1        Running    0           9m
   ```

   By adding `-l` followed by key/value pairs as filter requirements, we could see the desired pods.

Linking service with a replication controller by using label selectors

Service in Kubernetes is used to expose the port and for load-balancing:

1. In some cases, you'll need to add a service in front of the replication controller in order to expose the port to the outside world or balance the load. We will use the configuration file to create services for the staging pod and command line for production pods in the following example:

   ```
   // example of exposing staging pod
   # cat staging-nginx-service.yaml
   apiVersion: v1
   kind: Service
   metadata:
     name: nginx
   ```

```
  labels:
    project: pilot
    environment: staging
    tier: frontend
spec:
  ports:
    -
      protocol: TCP
      port: 80
      targetPort: 80
  selector:
    project: pilot
    environment: staging
    tier: frontend
  type: LoadBalancer
// create the service by configuration file
# kubectl create -f staging-nginx-service.yaml
service "nginx" created
```

2. Using `kubectl describe` to describe the details of the service:

```
// describe service
# kubectl describe service nginx
Name:         nginx
Namespace:      default
Labels:        environment=staging,project=pilot,tier=frontend
Selector:      environment=staging,project=pilot,tier=frontend
Type:         LoadBalancer
IP:          192.168.167.68
Port:         <unnamed>   80/TCP
Endpoints:      192.168.80.28:80
Session Affinity:   None
No events.
```

Using `curl` for the ClusterIP could return the welcome page of nginx.

3. Next, let's add the service for RC with label selectors:

   ```
   // add service for nginx-prod RC
   # kubectl expose rc nginx-prod --port=80 --type=LoadBalancer --sel
   ector="project=pilot,environment=production,tier=frontend"
   ```

4. Using `kubectl describe` to describe the details of service:

   ```
   # kubectl describe service nginx-prod
   Name:      nginx-prod
   Namespace:    default
   Labels:      environment=production,project=pilot,tier=frontend
   Selector:    environment=production,project=pilot,tier=frontend
   Type:      LoadBalancer
   IP:      192.168.200.173
   Port:      <unnamed>  80/TCP
   NodePort:    <unnamed>  32336/TCP
   Endpoints:    192.168.80.31:80,192.168.80.32:80
   Session Affinity:  None
   No events.
   ```

   When we use `curl 192.168.200.173`, we can see the welcome page of nginx just like the staging one.

It will return a `Connection reset by peer` error if you specify the empty pod set by the selector.

There's more…

In some cases, we might want to tag the resources with some values just for reference in the programs or tools. The non-identifying tags could use annotations instead, which are able to use structured or unstructured data. Unlike labels, annotations are not for querying and selecting. The following example will show you how to add an annotation into a pod and how to leverage them inside the container by downward API:

```
# cat annotation-sample.yaml
apiVersion: v1
kind: Pod
metadata:
  name: annotation-sample
```

```
  labels:
    project: pilot
    environment: staging
  annotations:
    git: 6328af0064b3db8b913bc613876a97187afe8e19
    build: "20"
spec:
  containers:
    -
      image: busybox
      imagePullPolicy: IfNotPresent
      name: busybox
      command: ["sleep", "3600"]
```

You could then use downward API, which we discussed in volumes, to access annotations in containers:

```
# cat annotation-sample-downward.yaml
apiVersion: v1
kind: Pod
metadata:
  name: annotation-sample
  labels:
    project: pilot
    environment: staging
  annotations:
    git: 6328af0064b3db8b913bc613876a97187afe8e19
    build: "20"
spec:
  containers:
    -
      image: busybox
      imagePullPolicy: IfNotPresent
      name: busybox
      command: ["sh", "-c", "while true; do if [[ -e /etc/annotations ]];
then cat /etc/annotations; fi; sleep 5; done"]
```

```
      volumeMounts:
      - name: podinfo
        mountPath: /etc
volumes:
    - name: podinfo
      downwardAPI:
        items:
          - path: "annotations"
            fieldRef:
              fieldPath: metadata.annotations
```

In this way, `metadata.annotations` will be exposed in the container as a file format under `/etc/annotations`. We could also check the pod logs are printing out the file content into stdout:

```
// check the logs we print in command section
# kubectl logs -f annotation-sample
build="20"
git="6328af0064b3db8b913bc613876a97187afe8e19"
kubernetes.io/config.seen="2015-12-28T12:23:33.154911056Z"
kubernetes.io/config.source="api"
```

See also

You can practice labels and selectors through the following recipes:

- *Working with pods*
- *Working with a replication controller*
- *Working with services*
- *Working with volumes*
- The *Working with configuration files* recipe in *Chapter 3*, *Playing with Containers*

3
Playing with Containers

In this chapter, we will cover the following topics:

- Scaling your containers
- Updating live containers
- Forwarding container ports
- Ensuring flexible usage of your containers
- Working with configuration files

Introduction

Talking about container management, you need to know some differences to compare it with application package management, such as rpm/dpkg, because you can run multiple containers on the same machine. You also need to care of the network port conflicts. This chapter covers how to update, scale, and launch the container application using Kubernetes.

Scaling your containers

Kubernetes has a scheduler to assign the container to the right node. In addition, you can easily scale out and scale down the number of containers. The Kubernetes scaling function will conduct the replication controller to adjust the number of containers.

Getting ready

Prepare the following YAML file, which is a simple replication controller to launch two nginx containers. Also, service will expose the TCP port `30080`:

```
# cat nginx-rc-svc.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx
spec:
  replicas: 2
  selector:
      sel : my-selector
  template:
    metadata:
        labels:
          sel : my-selector
    spec:
      containers:
      - name: my-container
        image: nginx
---
apiVersion: v1
kind: Service
metadata:
name: my-nginx

spec:
  ports:
    - protocol: TCP
      port: 80
      nodePort: 30080
  type: NodePort
  selector:
     sel: my-selector
```

[`NodePort` will bind all the Kubernetes nodes; therefore, make sure `NodePort` is not used by other processes.]

Use the `kubectl` command to create resources as follows:

```
# kubectl create -f nginx-service.yaml
replicationcontroller "my-nginx" created
service "my-nginx" created
```

Wait for a moment to completely launch two nginx containers as follows:

```
# kubectl get pods
NAME              READY     STATUS     RESTARTS    AGE
my-nginx-iarzy    1/1       Running    0           7m
my-nginx-ulkvh    1/1       Running    0           7m

# kubectl get services
NAME          CLUSTER_IP        EXTERNAL_IP    PORT(S)    SELECTOR
AGE
kubernetes    192.168.0.1       <none>         443/TCP    <none>
44d
my-nginx      192.168.95.244    nodes          80/TCP     sel=my-selector
7m
```

How to do it…

Kubernetes has a command that changes the number of replicas for service:

1. Type the `kubectl scale` command as follows to specify the desired replicas:

   ```
   # kubectl scale --replicas=4 rc my-nginx
   replicationcontroller "my-nginx" scaled
   ```

   This example indicates that the replication controller, which is named `my-nginx`, changes the replicas to `4`.

2. Type `kubectl get pods` to confirm the result as follows:

   ```
   # kubectl get pods
   NAME              READY     STATUS     RESTARTS    AGE
   my-nginx-iarzy    1/1       Running    0           20m
   my-nginx-r5lnq    1/1       Running    0           1m
   my-nginx-uhe8r    1/1       Running    0           1m
   my-nginx-ulkvh    1/1       Running    0           20m
   ```

How it works...

The `kubectl scale` feature can change the number of replicas; not only increase, but also decrease. For example, you can change back to two replicas as follows:

```
# kubectl scale --replicas=2 rc my-nginx
replicationcontroller "my-nginx" scaled


# kubectl get pods
NAME              READY     STATUS         RESTARTS    AGE
my-nginx-iarzy    0/1       Terminating    0           40m
my-nginx-r5lnq    1/1       Running        0           21m
my-nginx-uhe8r    1/1       Running        0           21m
my-nginx-ulkvh    0/1       Terminating    0           40m
# kubectl get pods
NAME              READY     STATUS     RESTARTS    AGE
my-nginx-r5lnq    1/1       Running    0           25m
my-nginx-uhe8r    1/1       Running    0           25m
```

There is an option `--current-replicas` that specifies the expected current replicas. If it doesn't match, Kubernetes doesn't perform the scale function as follows:

```
//abort scaling, because current replica is 2, not 3
# kubectl scale --current-replicas=3 --replicas=4
rc my-nginx
Expected replicas to be 3, was 2


# kubectl get pods
NAME              READY     STATUS     RESTARTS    AGE
my-nginx-r5lnq    1/1       Running    0           27m
my-nginx-uhe8r    1/1       Running    0           27m
```

It will help prevent human error. By default, `--current-replicas` equals -1, which means bypass to check the current number of replicas:

```
//no matter current number of replicas, performs to change to 4
# kubectl scale --current-replicas=-1 --replicas=4
 rc my-nginx
replicationcontroller "my-nginx" scaled

# kubectl get pods
NAME              READY     STATUS    RESTARTS   AGE
my-nginx-dimxj    1/1       Running   0          5s
my-nginx-eem3a    1/1       Running   0          5s
my-nginx-r5lnq    1/1       Running   0          35m
my-nginx-uhe8r    1/1       Running   0          35m
```

See also

This recipe described how to change the number of pods using the scaling option by the replication controller. It is useful to scale up and scale down your application quickly.
To know more about how to update your container, refer to the following recipes:

- *Updating live containers*
- *Ensuring flexible usage of your containers*

Updating live containers

For the benefit of containers, we can easily publish new programs by executing the latest image, and reduce the headache of environment setup. But, what about publishing the program on running containers? Using native Docker commands, we have to stop the running containers prior to booting up new ones with the latest images and the same configurations. There is a simple and efficient zero-downtime method to update your program in the Kubernetes system. It is called rolling-update. We will show this solution to you in this recipe.

Getting ready

Rolling-update works on the units of the replication controller. The effect is to create new pods one by one to replace the old one. The new pods in the target replication controller are attached to the original labels. Therefore, if any service exposes this replication controller, it will take over the newly created pods directly.

For a later demonstration, we are going to update a new nginx image. In addition to this, we are going to make sure that nodes get your customized image, pushing it to Docker Hub, the public Docker registry, or private registry.

For example, you can create the image by writing your own `Dockerfile`:

```
$ cat Dockerfile
FROM nginx
RUN echo "Happy Programming!" > /usr/share/nginx/html/index.html
```

In this Docker image, we changed the content of the default `index.html` page. Then, you can build your image and push it with the following commands:

```
// push to Docker Hub
$ docker build -t <DOCKERHUB_ACCOUNT>/common-nginx . && docker push
<DOCKERHUB_ACCOUNT>/common-nginx
// Or, you can also push to your private docker registry
$ docker build -t <RESITRY_NAME>/common-nginx . && docker push <RESITRY_
NAME>/common-nginx
```

To add nodes' access authentications of the private Docker registry, please take the *Working with the private Docker registry* recipe in *Chapter 5*, *Building a Continuous Delivery Pipeline*, as a reference.

How to do it...

You'll now learn how to publish a Docker image. The following steps will help you successfully publish a Docker image:

1. At the beginning, create a pair of replication controller and service for rolling-update testing. As shown in the following statement, a replication controller with five replicas will be created. The nginx program exposed port `80` to the container, while the Kubernetes service transferred the port to `8080` in the internal network:

   ```
   // Create a replication controller named nginx-rc
   # kubectl run nginx-rc --image=nginx --replicas=5 --port=80 --labe
   ls="User=Amy,App=Web,State=Testing"
   replicationcontroller "nginx-rc" created
   // Create a service supporting nginx-rc
   # kubectl expose rc nginx-rc --port=8080 --target-port=80
   --name="nginx-service"
   service "nginx-service" exposed
   # kubectl get service nginx-service
   ```

```
NAME              CLUSTER_IP        EXTERNAL_IP    PORT(S)     SELECTOR
AGE

nginx-service     192.168.163.46    <none>         8080/TCP    App=Web,
State=Testing,User=Amy    35s
```

You can evaluate whether the components work fine or not by examining `<POD_IP>:80` and `<CLUSTER_IP>:8080`.

2. Now, we are good to move on to the container update step! The Kubernetes subcommand `rolling-update` helps to keep the live replication controller up to date. In the following command, users have to specify the name of the replication controller and the new image. Here, we will use the image that is being uploaded to Docker Hub:

```
# kubectl rolling-update nginx-rc --image=<DOCKERHUB_ACCOUNT>/
common-nginx

Created nginx-rc-b6610813702bab5ad49d4aadd2e5b375

Scaling up nginx-rc-b6610813702bab5ad49d4aadd2e5b375 from 0 to 5,
scaling down nginx-rc from 5 to 0 (keep 5 pods available, don't
exceed 6 pods)

Scaling nginx-rc-b6610813702bab5ad49d4aadd2e5b375 up to 1
```

3. You may see that the process is hanging. Because `rolling-update` will start a single new pod at a time and wait for a period of time; the default is one minute to stop an old pod and create a second new pod. From this idea, while updating, there always is one more pod on the serving, one more pod than the desired state of the replication controller. In this case, there would be six pods. While updating the replication controller, please access another terminal for a brand-new process.

4. Check the state of the replication controller for more concepts:

```
# kubectl get rc

CONTROLLER                                        CONTAINER(S)      IMA
GE(S)                                                         SELECTOR
REPLICAS    AGE

nginx-rc                                          nginx-rc          ngi
nx                                                            App=Web,S
tate=Testing,User=Amy,deployment=313da350dea9227b89b4f0340699a388
5           1m

nginx-rc-b6610813702bab5ad49d4aadd2e5b375   nginx-
rc          <DOCKERHUB_ACCOUNT>/common-nginx
App=Web,State=Testing,User=Amy,deployment=b6610813702bab5ad49d4aad
d2e5b375    1           16s
```

5. As you will find, the system creates an almost identical replication controller with a postfix name. A new label key `deployment` is added to both the replication controllers for discriminating. On the other hand, new `nginx-rc` is attached to the other original labels. Service will also take care of the new pods at the same time:

```
// Check service nginx-service while updating
# kubectl describe service nginx-service
Name:      nginx-service
Namespace:    default
Labels:      App=Web,State=Testing,User=Amy
Selector:    App=Web,State=Testing,User=Amy
Type:      ClusterIP
IP:      192.168.163.46
Port:      <unnamed>  8080/TCP
Endpoints:    192.168.15.5:80,192.168.15.6:80,192.168.15.7:80 + 3
more...
Session Affinity:  None
No events.
```

There are six endpoints of pods covered by `nginx-service`, which is supported by the definition of rolling-update.

6. Go back to the console running the update process. After it completes the update, you can find procedures as follows:

```
Created nginx-rc-b6610813702bab5ad49d4aadd2e5b375
Scaling up nginx-rc-b6610813702bab5ad49d4aadd2e5b375 from 0 to 5,
scaling down nginx-rc from 5 to 0 (keep 5 pods available, don't
exceed 6 pods)
Scaling nginx-rc-b6610813702bab5ad49d4aadd2e5b375 up to 1
Scaling nginx-rc down to 4
Scaling nginx-rc-b6610813702bab5ad49d4aadd2e5b375 up to 2
Scaling nginx-rc down to 3
Scaling nginx-rc-b6610813702bab5ad49d4aadd2e5b375 up to 3
Scaling nginx-rc down to 2
Scaling nginx-rc-b6610813702bab5ad49d4aadd2e5b375 up to 4
Scaling nginx-rc down to 1
Scaling nginx-rc-b6610813702bab5ad49d4aadd2e5b375 up to 5
Scaling nginx-rc down to 0
Update succeeded. Deleting old controller: nginx-rc
Renaming nginx-rc-b6610813702bab5ad49d4aadd2e5b375 to nginx-rc
replicationcontroller "nginx-rc" rolling updated
```

Old `nginx-rc` is gradually taken out of service by scaling down.

7. At the final steps of the update, the new replication controller is scaled up to five pods to meet the desired state and replace the old one eventually:

```
// Take a look a current replication controller
// The new label "deployment" is remained after update
# kubectl get rc nginx-rc
CONTROLLER   CONTAINER(S)   IMAGE(S)                SELECTOR
REPLICAS   AGE
nginx-rc     nginx-rc       <DOCKERHUB_ACCOUNT>/common-nginx   App
=Web,State=Testing,User=Amy,deployment=b6610813702bab5ad49d4aadd2e
5b375   5          40s
```

8. Checking service with ClusterIP and port, we can now have all our pods in the replication controller updated:

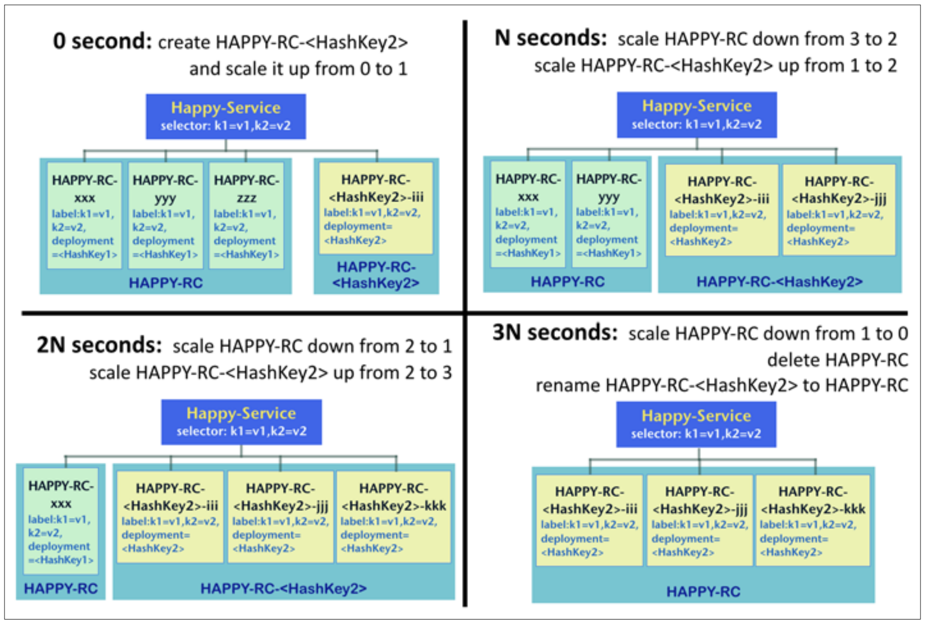
```
# curl 192.168.163.46:8080
Happy Programming!
```

9. According to the previous demonstration, it costs about five minutes to publish a new Docker image. It is because the updating time is set to one minute by default for the procedure of scaling up and down. It is possible for you to have a faster or slower pace of update by counting on the tag `--update-period`. The valid time units are `ns`, `us`, `ms`, `s`, `m`, and `h`. For example, `--update-period=1m0s`:

```
// Try on this one!
# kubectl rolling-update <REPLICATION_CONTROLLER_NAME>
--image=<IMAGE_NAME> --update-period=10s
```

How it works...

In this section, we will discuss rolling-update in detail. How about renewing a replication controller with *N* seconds as the period of updating? See the following image:


0 second: create HAPPY-RC-<HashKey2> and scale it up from 0 to 1
Happy-Service selector: k1=v1,k2=v2
HAPPY-RC-xxx label:k1=v1, k2=v2, deployment =<HashKey1>
HAPPY-RC-yyy label:k1=v1, k2=v2, deployment =<HashKey1>
HAPPY-RC-zzz label:k1=v1, k2=v2, deployment =<HashKey1>
HAPPY-RC
HAPPY-RC-<HashKey2>-iii label:k1=v1,k2=v2, deployment=<HashKey2>
HAPPY-RC-<HashKey2>
N seconds: scale HAPPY-RC down from 3 to 2 scale HAPPY-RC-<HashKey2> up from 1 to 2
HAPPY-RC-<HashKey2>-jjj
2N seconds: scale HAPPY-RC down from 2 to 1 scale HAPPY-RC-<HashKey2> up from 2 to 3
HAPPY-RC-<HashKey2>-kkk
3N seconds: scale HAPPY-RC down from 1 to 0 delete HAPPY-RC rename HAPPY-RC-<HashKey2> to HAPPY-RC

The previous image indicates each step of the updating procedure. We may get some important ideas from rolling-update:

- Each pod in both the replication controllers has a new label, but an unequal value to point out the difference. Besides, the other labels are the same, so service can still cover both the replication controllers by selectors while updating.
- We would spend *# pod in replication controller * update period* time for migrating a new configuration.
- For zero-downtime updating, the total number of pods covered by the service should meet the desired state. For example, in the preceding image, there should be always three pods running at a time for the service.
- Rolling-update procedure doesn't assure users when the newly created pod, in **HAPPY-RC-<HashKey2>**, is in running state. That's why we need an update period. After a period of time, *N* seconds in the preceding case, a new pod should be ready to take the place of an old pod. Then, it is good to terminate one old pod.
- The period of updating time should be the worst case of the time required by a new pod from pulling an image to running.

There's more...

While doing rolling-update, we may specify the image for a new replication controller. But sometimes, we cannot update the new image successfully. It is because of container's image pull policy.

To update with a specific image, it will be great if users provide a tag so that what version of the image should be pulled is clear and accurate. However, most of the time, the latest one to which users look for and the latest tagged image could be regarded as the same one in local, since they are called the latest as well. Like the command `<DOCKERHUB_ACCOUNT>/common-nginx:latest image` will be used in this update:

```
# kubectl rolling-update nginx-rc --image=<DOCKERHUB_ACCOUNT>/common-
nginx --update-period=10s
```

Still, nodes will ignore to pull the latest version of `common-nginx` if they find an image labeled as the same request. For this reason, we have to make sure that the specified image is always pulled from the registry.

In order to change the configuration, the subcommand `edit` can help in this way:

```
# kubectl edit rc <REPLICATION_CONTROLLER_NAME>
```

Then, you can edit the configuration of the replication controller in the YAML format. The policy of image pulling could be found in the following class structure:

```
apiVersion: v1
kind: replicationcontroller
spec:
  template:
    spec:
      containers:
      - name: <CONTAINER_NAME>
        image: <IMAGE_TAG>
                 imagePullPolicy: IfNotPresent
:
```

The value `IfNotPresent` tells the node to only pull the image not presented on the local disk. By changing the policy to `Always`, users will be able to avoid updating failure. It is workable to set up the key-value item in the configuration file. So, the specified image is guaranteed to be the one in the image registry.

See also

Pod is the basic computing unit in the Kubernetes system. You can learn how to use pods even more effectively through the following recipes:

- *Scaling your containers*
- The *Moving monolithic to microservices*, *Integrating with Jenkins*, *Working with the private Docker registry*, and *Setting up the Continuous Delivery pipeline* recipes in *Chapter 5*, *Building a Continuous Delivery Pipeline*

Forwarding container ports

In the previous chapters, you learned how to work with the Kubernetes services to forward the container port internally and externally. Now, it's time to take it a step further to see how it works.

There are four networking models in Kubernetes, and we'll explore the details in the following sections:

- Container-to-container communications
- Pod-to-pod communications
- Pod-to-service communications
- External-to-internal communications

Getting ready

In this section, we will run two nginx web apps in order to demonstrate how these four models work. The default port of nginx in Docker Hub (`https://hub.docker.com/_/nginx`) is `80`. We will then create another nginx Docker image by modifying the nginx configuration file and Dockerfile from `80` to `8800`. The following steps will show you how to build it from scratch, but you are free to skip it and use our prebuilt image (`https://hub.docker.com/r/msfuko/nginx_8800`) as well.

Let's create one simple nginx configuration file first. Note that we need to listen to the `8800` port:

```
// create one nginx config file
# cat nginx.conf
server {
    listen       8800;
    server_name  localhost;
    #charset koi8-r;
    #access_log  /var/log/nginx/log/host.access.log  main;

    location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm;
    }

    #error_page  404              /404.html;

    # redirect server error pages to the static page /50x.html
    #
    error_page   500 502 503 504  /50x.html;
    location = /50x.html {
        root   /usr/share/nginx/html;
    }
}
```

Next, we need to change the default nginx `Dockerfile` from expose `80` to `8800`:

```
// modifying Dockerfile as expose 8800 and add config file inside
# cat Dockerfile
FROM debian:jessie

MAINTAINER NGINX Docker Maintainers "docker-maint@nginx.com"

RUN apt-key adv --keyserver hkp://pgp.mit.edu:80 --recv-keys
573BFD6B3D8FBC641079A6ABABF5BD827BD9BF62
RUN echo "deb http://nginx.org/packages/mainline/debian/ jessie nginx" >>
/etc/apt/sources.list

ENV NGINX_VERSION 1.9.9-1~jessie

RUN apt-get update && \
    apt-get install -y ca-certificates nginx=${NGINX_VERSION} && \
        rm -rf /var/lib/apt/lists/*
COPY nginx.conf /etc/nginx/conf.d/default.conf
```

```
# forward request and error logs to docker log collector
RUN ln -sf /dev/stdout /var/log/nginx/access.log
RUN ln -sf /dev/stderr /var/log/nginx/error.log

VOLUME ["/var/cache/nginx"]

EXPOSE 8800

CMD ["nginx", "-g", "daemon off;"]
```

Then, we'll need to build it using the Docker command:

```
// build docker image
# docker build -t $(YOUR_DOCKERHUB_ACCOUNT)/nginx_8800 .
```

Finally, we can push to our Docker Hub repository:

```
// be sure to login via `docker login` first
# docker push $(YOUR_DOCKERHUB_ACCOUNT)/nginx_8800
```

After this, you should be able to run the container by the pure Docker command: `docker run -d -p 8800:8800 msfuko/nginx_8800`. Using `curl $IP:8800`, you should be able to see the welcome page of nginx.

How to find my $IP?

If you are running on Linux, then `ifconfig` could help to figure out the value of $IP. If you are running on another platform via Docker machine, `docker-machine ip` could help you with that.

How to do it…

Pod contains one or more containers, which run on the same host. Each pod has their own IP address; all the containers inside a pod see each other as on the same host. Containers inside a pod will be created, deployed, and deleted almost at the same time.

Container-to-container communications

We'll create two nginx containers in one pod that will listen to port `80` and `8800`, individually. In the following configuration file, you should change the second image path as the one you just built and pushed:

```
// create 2 containers inside one pod
# cat nginxpod.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: nginxpod
spec:
  containers:
    -
      name: nginx80
      image: nginx
      ports:
        -
          containerPort: 80
          hostPort: 80
    -
      name: nginx8800
      image: msfuko/nginx_8800
      ports:
        -
          containerPort: 8800
          hostPort: 8800

// create the pod
# kubectl create -f nginxpod.yaml
pod "nginxpod" created
```

After the image is pulled and run, we can see that the status becomes running using the `kubectl` command:

```
// list nginxpod pod
# kubectl get pods nginxpod
NAME        READY      STATUS     RESTARTS    AGE
nginxpod    2/2        Running    0           12m
```

We could find the count in the `READY` column become `2/2`, since there are two containers inside this pod. Using the `kubectl describe` command, we can see the details of the pod:

```
// show the details of nginxpod
# kubectl describe pod nginxpod
Name:           nginxpod
```

```
Namespace:      default
Image(s):      nginx,msfuko/nginx_8800
Node:        kube-node1/10.96.219.33
Start Time:      Sun, 24 Jan 2016 10:10:01 +0000
Labels:        <none>
Status:        Running
Reason:
Message:
IP:        192.168.55.5
Replication Controllers:  <none>
Containers:
  nginx80:
    Container ID:  docker://3b467d8772f09c57d0ad85caa66b8379799f3a60da055
d7d8d362aee48dfa832
    Image:    nginx
    ...
  nginx8800:
    Container ID:  docker://80a77983f6e15568db47bd58319fad6d22a330c1c4c92
63bca9004b80ecb6c5f
    Image:    msfuko/nginx_8800
   ...
```

We could see the pod is run on `kube-node1` and the IP of the pod is `192.168.55.5`. Let's log in to `kube-node1` to inspect these two containers:

```
// list containers
# docker ps
CONTAINER ID        IMAGE
COMMAND                  CREATED             STATUS              POR
TS                                NAMES
80a77983f6e1        msfuko/nginx_8800
"nginx -g 'daemon off"   32 minutes ago      Up 32 minutes
k8s_nginx8800.645004b9_nginxpod_default_a08ed7cb-c282-11e5-9f21-
025a2f393327_9f85a41b
3b467d8772f0        nginx
"nginx -g 'daemon off"   32 minutes ago      Up 32 minutes
k8s_nginx80.5098ff7f_nginxpod_default_a08ed7cb-c282-11e5-9f21-
025a2f393327_9922e484
71073c074a76        gcr.io/google_containers/pause:0.8.0
"/pause"                 32 minutes ago      Up 32 minutes
0.0.0.0:80->80/tcp, 0.0.0.0:8800->8800/tcp   k8s_POD.5c2e23f2_nginxpod_
default_a08ed7cb-c282-11e5-9f21-025a2f393327_77e79a63
```

We know that the ID of the two containers we created are `3b467d8772f0` and `80a77983f6e1`.

We will use `jq` as the JSON parser to reduce the redundant information. For installing `jq`, simply download the binary file from `https://stedolan.github.io/jq/download`:

```
// inspect the nginx container and use jq to parse it
# docker inspect 3b467d8772f0 | jq '.[]| {NetworkMode: .HostConfig.
NetworkMode, NetworkSettings: .NetworkSettings}'
{
  "NetworkMode": "container:71073c074a761a33323bb6601081d44a79ba7de3dd593
45fc33a36b00bca613f",
  "NetworkSettings": {
    "Bridge": "",
    "SandboxID": "",
    "HairpinMode": false,
    "LinkLocalIPv6Address": "",
    "LinkLocalIPv6PrefixLen": 0,
    "Ports": null,
    "SandboxKey": "",
    "SecondaryIPAddresses": null,
    "SecondaryIPv6Addresses": null,
    "EndpointID": "",
    "Gateway": "",
    "GlobalIPv6Address": "",
    "GlobalIPv6PrefixLen": 0,
    "IPAddress": "",
    "IPPrefixLen": 0,
    "IPv6Gateway": "",
    "MacAddress": "",
    "Networks": null
  }
}
```

We can see that the network mode is set as mapped container mode. The network bridge container is `container:71073c074a761a33323bb6601081d44a79ba7de3dd59345fc33a36b00bca613f`.

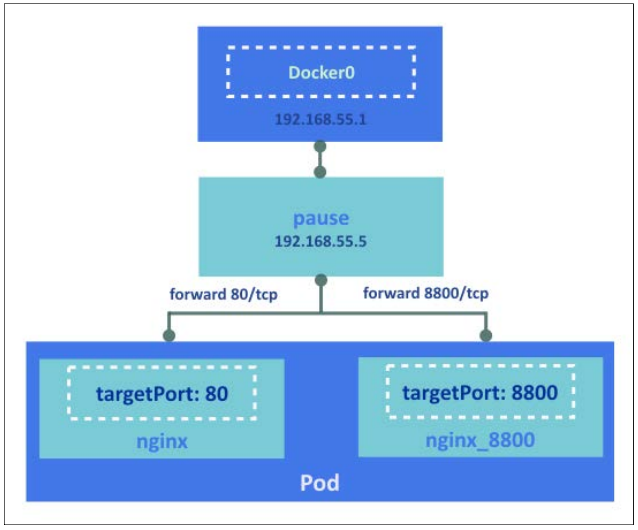
Let's see another setting about container `nginx_8800`:

```
// inspect nginx_8800
# docker inspect 80a77983f6e1 | jq '.[]| {NetworkMode: .HostConfig.
NetworkMode, NetworkSettings: .NetworkSettings}'
{
  "NetworkMode": "container:71073c074a761a33323bb6601081d44a79ba7de3dd593
45fc33a36b00bca613f",
  "NetworkSettings": {
    "Bridge": "",
    "SandboxID": "",
    "HairpinMode": false,
    "LinkLocalIPv6Address": "",
    "LinkLocalIPv6PrefixLen": 0,
    "Ports": null,
    "SandboxKey": "",
    "SecondaryIPAddresses": null,
    "SecondaryIPv6Addresses": null,
    "EndpointID": "",
    "Gateway": "",
    "GlobalIPv6Address": "",
    "GlobalIPv6PrefixLen": 0,
    "IPAddress": "",
    "IPPrefixLen": 0,
    "IPv6Gateway": "",
    "MacAddress": "",
    "Networks": null
  }
}
```

The network mode is also set to container:
`71073c074a761a33323bb6601081d44a79ba7de3dd59345fc33a36b00bca613`. Which container is this? We will then find out that this network container is created by Kubernetes when your pod starts. The container is named `gcr.io/google_containers/pause`:

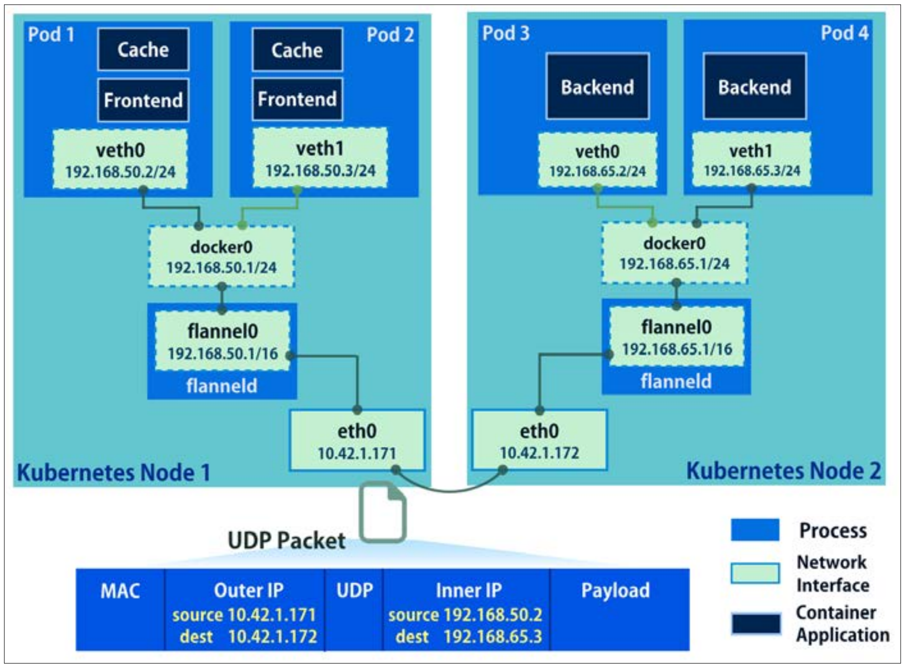
```
// inspect network container `pause`
# docker inspect 71073c074a76 | jq '.[]| {NetworkMode: .HostConfig.
NetworkMode, NetworkSettings: .NetworkSettings}'
{
  "NetworkMode": "default",
  "NetworkSettings": {
    "Bridge": "",
    "SandboxID":
"59734bbe4e58b0edfc92db81ecda79c4f475f6c8433e17951e9c9047c69484e8",
    "HairpinMode": false,
    "LinkLocalIPv6Address": "",
    "LinkLocalIPv6PrefixLen": 0,
    "Ports": {
      "80/tcp": [
        {
          "HostIp": "0.0.0.0",
          "HostPort": "80"
        }
      ],
      "8800/tcp": [
        {
          "HostIp": "0.0.0.0",
          "HostPort": "8800"
        }
      ]
    },
    "SandboxKey": "/var/run/docker/netns/59734bbe4e58",
    "SecondaryIPAddresses": null,
    "SecondaryIPv6Addresses": null,
    "EndpointID":
"d488fa8d5ee7d53d939eadda106e97ff01783f0e9dc9e4625d9e69500e1fa451",
    "Gateway": "192.168.55.1",
    "GlobalIPv6Address": "",
    "GlobalIPv6PrefixLen": 0,
    "IPAddress": "192.168.55.5",
    "IPPrefixLen": 24,
    "IPv6Gateway": "",
    "MacAddress": "02:42:c0:a8:37:05",
    "Networks": {
      "bridge": {
        "EndpointID":
"d488fa8d5ee7d53d939eadda106e97ff01783f0e9dc9e4625d9e69500e1fa451",
        "Gateway": "192.168.55.1",
        "IPAddress": "192.168.55.5",
        "IPPrefixLen": 24,
        "IPv6Gateway": "",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "MacAddress": "02:42:c0:a8:37:05"
      }
    }
  }
}
```

We will find out that the network mode is set to `default` and its IP address is set to the IP of the pod `192.168.55.5`; the gateway is set to `docker0` of the node. The routing illustration will be as shown in the following image. The network container pause will be created when a pod is created, which will be used to handle the route of the pod network. Then, two containers will share the network interface with pause; that's why they see each other as localhost:



Pod-to-pod communications

Since each pod has its own IP address, it makes the communication between pods easy. In the previous chapter, we use flannel as the overlay network, which will define different network segments for each node. Each packet is encapsulated to a UDP packet so that each pod IP is routable. The packet from **Pod1** will go though the **veth** (virtual network interface) device, which connects to **docker0** and routes to **flannel0**. The traffic is encapsulated by flanneld and sent to the host (**10.42.1.172**) of the target pod:



Pod-to-service communications

Pods could be stopped accidentally, so the IP of the pod could be changed. When we expose the port for a pod or a replication controller, we create a Kubernetes service acting as a proxy or a load balancer. Kubernetes will create a virtual IP, which will receive the request from clients and proxy the traffic to the pods in a service. Let's review how to do this. At first, we will create a replication controller named `my-first-nginx`:

```
// create a rc named my-first-nginx
# kubectl run my-first-nginx --image=nginx --replicas=2 --port=80
replicationcontroller "my-first-nginx" created
```

Then, let's list the pods to ensure two pods are created by this `rc`:

```
// two pods will be launched by this rc
# kubectl get pod
NAME                          READY     STATUS    RESTARTS    AGE
my-first-nginx-hsdmz          1/1       Running   0           17s
my-first-nginx-xjtxq          1/1       Running   0           17s
```

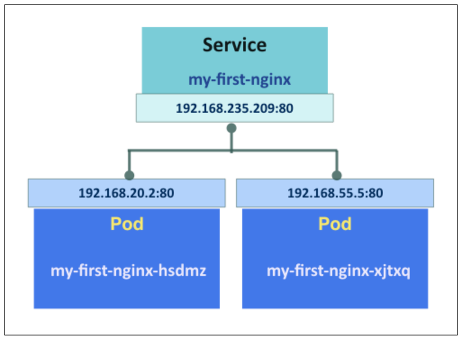
Next, let's expose one port `80` for the pod, which is the default port of the `nginx` app:

```
// expose port 80 for my-first-nginx
# kubectl expose rc my-first-nginx --port=80
service "my-first-nginx" exposed
```

Use `describe` to see the details of the service. The service type we create is a ClusterIP:

```
// check the details of the service
# kubectl describe service my-first-nginx
Name:       my-first-nginx
Namespace:     default
Labels:       run=my-first-nginx
Selector:     run=my-first-nginx
Type:       ClusterIP
IP:       192.168.235.209
Port:       <unnamed>   80/TCP
Endpoints:     192.168.20.2:80,192.168.55.5:80
Session Affinity:   None
No events.
```

The virtual IP of the service is `192.168.235.209`, which exposes the port `80`. The service will then dispatch the traffic into two endpoints `192.168.20.2:80` and `192.168.55.5:80`. The illustration is as follows:



`kube-proxy` is a daemon that works as a network proxy on every node. It helps to reflect the settings of services, such as IPs or ports, on each node. It will create the corresponding iptables rules:

```
// list iptables rule by filtering my-first-nginx
# iptables -t nat -L | grep my-first-nginx
REDIRECT   tcp  --  anywhere            192.168.235.209      /* default/
my-first-nginx: */ tcp dpt:http redir ports 43321
DNAT       tcp  --  anywhere            192.168.235.209      /* default/
my-first-nginx: */ tcp dpt:http to:10.96.219.33:43321
```

These two rules are under the `KUBE-PORTALS-CONTAINER` and `KUBE-PORTALS-HOST` chains, which represent any traffic destined for the virtual IP with port `80` that will be redirected to the localhost on port `43321` no matter whether the traffic is from containers or hosts. The kube-proxy programs the iptables rule to make the pod and service communication available. You should be able to access `localhost:43321` on the target node or `$nodeIP:43321` inside the cluster.

Using the environment variables of the Kubernetes service in your program

Sometimes, you'll need to access the Kubernetes service in your program inside the Kubernetes cluster. You could use environment variables or DNS to access it. Environment variables are the easiest way and are supported naturally. When a service is created, `kubelet` will add a set of environment variables about this service:

- `$SVCNAME_SERVICE_HOST`
- `$SVCNAME_SERVICE_PORT`

Here, `$SVNNAME` is uppercase and the dashes are converted into underscores. The service that a pod wants to access must be created before the pod, otherwise the environment variables will not be populated. For example, the environment variables that `my-first-nginx` populate are:

- `MY_FIRST_NGINX_PORT_80_TCP_PROTO=tcp`
- `MY_FIRST_NGINX_SERVICE_HOST=192.168.235.209`
- `MY_FIRST_NGINX_SERVICE_PORT=80`
- `MY_FIRST_NGINX_PORT_80_TCP_ADDR=192.168.235.209`
- `MY_FIRST_NGINX_PORT=tcp://192.168.235.209:80`
- `MY_FIRST_NGINX_PORT_80_TCP_PORT=80`
- `MY_FIRST_NGINX_PORT_80_TCP=tcp://192.168.235.209:80`

External-to-internal communications

The external-to-internal communications could be set up using the external load balancer, such as GCE's ForwardingRules or AWS's ELB, or by accessing node IP directly. Here, we will introduce how accessing node IP could work. First, we'll run a replication controller with two replicas named `my-second-nginx`:

```
// create a rc with two replicas
# kubectl run my-second-nginx --image=nginx --replicas=2 --port=80
```

Next, we'll expose the service with port `80` with the type `LoadBalancer`. As we discussed in the service section, `LoadBalancer` will also expose a node port:

```
// expose port 80 for my-second-nginx rc with type LoadBalancer
# kubectl expose rc my-second-nginx --port=80 --type=LoadBalancer
```

We could now check the details of `my-second-nginx` service. It has a virtual IP `192.168.156.93` with port `80`. It also has a node port `31150`:

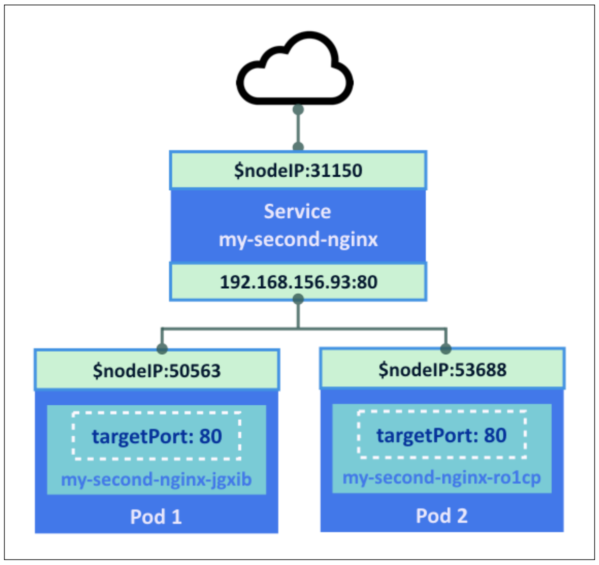
```
// list the details of service
# kubectl describe service my-second-nginx
Name:      my-second-nginx
Namespace:    default
Labels:      run=my-second-nginx
Selector:    run=my-second-nginx
Type:      LoadBalancer
IP:      192.168.156.93
Port:      <unnamed>  80/TCP
NodePort:    <unnamed>  31150/TCP
Endpoints:    192.168.20.3:80,192.168.55.6:80
Session Affinity:  None
No events.
```

Let's list the iptables rules to see the differences between the different types of service:

```
// list iptables rules and filtering my-seconde-nginx on node1
# iptables -t nat -L | grep my-second-nginx
REDIRECT   tcp  --  anywhere             anywhere             /* default/
my-second-nginx: */ tcp dpt:31150 redir ports 50563
DNAT       tcp  --  anywhere             anywhere             /* default/
my-second-nginx: */ tcp dpt:31150 to:10.96.219.141:50563
REDIRECT   tcp  --  anywhere             192.168.156.93       /* default/
my-second-nginx: */ tcp dpt:http redir ports 50563
```

```
DNAT       tcp  --  anywhere             192.168.156.93       /* default/
my-second-nginx: */ tcp dpt:http to:10.96.219.141:50563


// list iptables rules and filtering my-second-nginx on node2
# iptables -t nat -L | grep my-second-nginx
REDIRECT   tcp  --  anywhere             anywhere             /* default/
my-second-nginx: */ tcp dpt:31150 redir ports 53688
DNAT       tcp  --  anywhere             anywhere             /* default/
my-second-nginx: */ tcp dpt:31150 to:10.96.219.33:53688
REDIRECT   tcp  --  anywhere             192.168.156.93       /* default/
my-second-nginx: */ tcp dpt:http redir ports 53688
DNAT       tcp  --  anywhere             192.168.156.93       /* default/
my-second-nginx: */ tcp dpt:http to:10.96.219.33:53688
```

We have four rules related to `my-second-nginx` now. They are under the `KUBE-NODEPORT-CONTAINER`, `KUBE-NODEPORT-HOST`, `KUBE-PORTALS-CONTAINER`, and `KUBE-PORTALS-HOST` chains. Since we expose the node port in this example, if the traffic is from the outside world to node port `31150`, the traffic will be redirected to the target pod locally or across nodes. Following is an illustration of routing:



The traffic from node port (`x.x.x.x:31150`) or from ClusterIP (`192.168.156.93:80`) will be redirected to target pods by providing a load balancing mechanism across nodes. The ports `50563` and `53688` are dynamically assigned by Kubernetes.

See also

Kubernetes forwards port based on the overlay network. In this chapter, we also run pods and services with nginx. Reviewing the previous sections will help you to understand more about how to manipulate it. Also, look at the following recipes:

- The *Creating an overlay network*, *Running your first container in Kubernetes* recipes in *Chapter 1*, *Building Your Own Kubernetes*
- The *Working with pods*, *Working with services* in *Chapter 2*, *Walking through Kubernetes Concepts*

Ensuring flexible usage of your containers

Pod, in Kubernetes, means a set of containers, which is also the smallest computing unit. You may have known about the basic usage of pod in the previous recipes. Pods are usually managed by replication controllers and exposed by services; they work as applications with this scenario.

In this recipe, we will discuss two new features: `job` and `daemon set`. These two features can make the usage of pods more effective.

Getting ready

What are a job-like pod and a daemon-like pod? Pods in a Kubernetes job will be terminated directly after they complete their work. On the other hand, a daemon-like pod will be created in every node, while users want it to be a long running program served as a system daemon.

Both the job and daemon set belong to the extension of the Kubernetes API. Furthermore, the daemon set type is disabled in the default API settings. Then, you have to enable the usage later for testing. Without starting the setting of the daemon set, you will get an error about the unknown type:

```
# kubectl create -f daemonset-test.yaml
error validating "daemonset-test.yaml": error validating data: couldn't
find type: v1beta1.DaemonSet; if you choose to ignore these errors, turn
validation off with --validate=false
Or error of this one
Error from server: error when creating "daemonset-free.yaml": the server
could not find the requested resource
```

To enable the daemon set in the Kubernetes system, you should update a tag in the daemon of the Kubernetes apiserver: `--runtime-config=extensions/v1beta1/daemonsets=true`. Modify your service scripts or configuration options:

```
// For service init.d scripts, attached the tag after command hyperkube
apiserver or kube-apiserver
# cat /etc/init.d/kubernetes-master
(heading lines ignored)
:
# Start daemon.
echo $"Starting apiserver: "
daemon $apiserver_prog \
--service-cluster-ip-range=${CLUSTER_IP_RANGE} \
--insecure-port=8080 \
--secure-port=6443 \
--basic-auth-file="/root/ba_file" \
--address=0.0.0.0 \
--etcd_servers=${ETCD_SERVERS} \
--cluster_name=${CLUSTER_NAME} \
--runtime-config=extensions/v1beta1/daemonsets=true \
> ${logfile}-apiserver.log 2>&1 &
:
(ignored)
// For systemd service management, edit configuration files by add the
tag as optional one
# cat /etc/kubernetes/apiserver
(heading lines ignored)
:
# Add your own!
KUBE_API_ARGS="--cluster_name=Happy-k8s-cluster --runtime-
config=extensions/v1beta1/daemonsets=true"
```

After you set up the tag, remove the directory `/tmp/kubectl.schema`, which caches API schemas. Then, it is good to restart the Kubernetes apiserver:

```
// Remove the schema file
# rm -f /tmp/kubectl.schema
// The method to restart apiserver for init.d script
# service kubernetes-master restart
```

```
// Or, restart the daemon for systemd service management
# systemd restart kube-apiserver
// After restart daemon apiserver, you can find daemonsets is enable in
your API server
# curl http://localhost:8080/apis/extensions/v1beta1
{
  "kind": "APIResourceList",
  "groupVersion": "extensions/v1beta1",
  "resources": [
    {
      "name": "daemonsets",
      "namespaced": true
    },
    {
      "name": "daemonsets/status",
      "namespaced": true
    },
:
```

Next, for the following sections, we are going to demonstrate how to create a job and daemon set using configuration files. Take a look at the recipe *Working with configuration files* in this chapter to know more about other concepts.

How to do it…

There is no command-line interface for us to create a job or a daemon set. Therefore, we will build these two resource types by writing all the configurations in a template file.

Pod as a job

A job-like pod is suitable for testing your containers, which can be used for unit test or integration test; or, it can be used for static program. Like in the following example, we will write a job template to check the packages installed in image `ubuntu`:

```
# cat job-dpkg.yaml
apiVersion: extensions/v1beta1
kind: Job
metadata:
  name: package-check
spec:
  selector:
    matchLabels:
      image: ubuntu
      test: dpkg
  template:
    metadata:
      labels:
        image: ubuntu
        test: dpkg
        owner: Amy
    spec:
      containers:
      - name: package-check
        image: ubuntu
        command: ["dpkg-query", "-l"]
      restartPolicy: Never
```

A job resource needs a selector to define which pods should be covered as this job. If no selector is specified in template, it will just be the same as the labels of the job. The restart policy for pods created in a job should be set to `Never` or `OnFailure`, since a job goes to termination once it is completed successfully.

Now, you are ready to create a job using your template:

```
# kubectl create -f job-dpkg.yaml
job "package-check" created
```

After pushing the requested file, it is possible to verify the status of both the pod and job:

```
# kubectl get job
JOB             CONTAINER(S)    IMAGE(S)    SELECTOR
SUCCESSFUL
package-check   package-check   ubuntu      image in (ubuntu),test in
(dpkg)    1
// Check the job as well
# kubectl get pod
NAME                          READY     STATUS    RESTARTS    AGE
package-check-jrry1           0/1       Pending   0           6s
```

You will find that a pod is booting up for handling this task. This pod is going to be stopped very soon at the end of the process. The subcommand `logs` helps to get the result:

```
# kubectl logs package-check-gtyrc
Desired=Unknown/Install/Remove/Purge/Hold
| Status=Not/Inst/Conf-files/Unpacked/halF-conf/Half-inst/trig-aWait/
Trig-pend
|/ Err?=(none)/Reinst-required (Status,Err: uppercase=bad)
||/ Name                           Version
Architecture Description
+++-===================================-
===============================================-============-
=====================================================================-
===========
ii  adduser                        3.113+nmu3ubuntu3                all
add and remove users and groups
ii  apt                            1.0.1ubuntu2.10
amd64        commandline package manager
ii  apt-utils                      1.0.1ubuntu2.10
amd64        package management related utility programs
ii  base-files                     7.2ubuntu5.3
amd64        Debian base system miscellaneous files
:
(ignored)
```

Please go ahead and check the job `package-check` using the subcommand `describe`; the confirmation for pod completion and other messages are shown as system information:

```
# kubectl describe job package-check
```

Later, to remove the job you just created, stop it with the name:

```
# kubectl stop job package-check
job "package-check" deleted
```

Creating a job with multiple pods running

User can also decide the number of tasks that should be finished in a single job. It is helpful to solve some random and sampling problems. Let's try it on the same template in the previous example. We have to add the `spec.completions` item to indicate the pod number:

```
# cat job-dpkg.yaml
apiVersion: extensions/v1beta1
kind: Job
```

```
metadata:
  name: package-check
spec:
  completions: 3
  template:
    metadata:
      name: package-check-amy
      labels:
        image: ubuntu
        test: dpkg
        owner: Amy
    spec:
      containers:
      - name: package-check
        image: ubuntu
        command: ["dpkg-query", "-l"]
      restartPolicy: Never
```

Then, check how the job looks like using the subcommand `describe`:

```
# kubectl describe job package-check
Name:      package-check
Namespace:    default
Image(s):   ubuntu
Selector:   image in (ubuntu),owner in (Amy),test in (dpkg)
Parallelism:   3
Completions:   3
Labels:     image=ubuntu,owner=Amy,test=dpkg
Pods Statuses:   3 Running / 0 Succeeded / 0 Failed
No volumes.
Events:
  FirstSeen  LastSeen  Count  From  SubobjectPath  Reason      Message
  ---------  --------  -----  ----  -------------  ------      -------

  6s     6s     1   {job }      SuccessfulCreate  Created pod: package-
check-dk184
  6s     6s     1   {job }      SuccessfulCreate  Created pod: package-
check-3uwym
  6s     6s     1   {job }      SuccessfulCreate  Created pod: package-
check-eg4nl
```

As you can see, three pods are created to solve this job. Also, since the selector part is removed, the selector is copied from the labels.

Pod as a daemon set

If a Kubernetes daemon set is created, the defined pod will be deployed in every single node. It is guaranteed that the running containers occupy equal resources in each node. In this scenario, the container usually works as the daemon process.

For example, the following template has an `ubuntu` image container that keeps checking its memory usage half minute a time. We are going to build it as a daemon set:

```
# cat daemonset-free.yaml
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: ram-check
spec:
  selector:
    app: checkRam
    version: v1
  template:
    metadata:
      labels:
        app: checkRam
        owner: Amy
        version: v1
    spec:
      containers:
      - name: ubuntu-free
        image: ubuntu
        command: ["/bin/bash","-c","while true; do free; sleep 30; done"]
      restartPolicy: Always
```

As the job, the selector could be ignored, but it takes the values of the labels. We will always configure the restart policy of the daemon set as `Always`, which makes sure that every node has a pod running.

The abbreviation of the daemon set is `ds`; use this shorter one in the command-line interface for convenience:

```
# kubectl create -f daemonset-free.yaml
daemonset "ram-check" created
# kubectl get ds
NAME        CONTAINER(S)   IMAGE(S)   SELECTOR                     NODE-
SELECTOR
ram-check   ubuntu-free    ubuntu     app=checkRam,version=v1     <none>
// Go further look at the daemon set by "describe"
# kubectl describe ds ram-check
Name:     ram-check
Image(s):   ubuntu
Selector:   app=checkRam,version=v1
Node-Selector:   <none>
Labels:     app=checkRam,owner=Amy,version=v1
Desired Number of Nodes Scheduled: 3
Current Number of Nodes Scheduled: 3
Number of Nodes Misscheduled: 0
Pods Status:  3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Events:
  FirstSeen  LastSeen  Count  From    SubobjectPath  Reason      Message
  _________  ________  _____  ____    _____________  ______      _______
  3m     3m     1  {daemon-set }      SuccessfulCreate  Created pod: ram-
check-bti08
  3m     3m     1  {daemon-set }      SuccessfulCreate  Created pod: ram-
check-u9e5f
  3m     3m     1  {daemon-set }      SuccessfulCreate  Created pod: ram-
check-mxry2
```

Here, we have three pods running in separated nodes. They can still be recognized in the channel of the pod:

```
# kubectl get pod --selector=app=checkRam
NAME              READY     STATUS    RESTARTS    AGE
ram-check-bti08   1/1       Running   0           4m
ram-check-mxry2   1/1       Running   0           4m
ram-check-u9e5f   1/1       Running   0           4m
// Get the evidence!
```

```
# kubectl describe pods --selector=app=checkRam -o wide
NAME              READY     STATUS    RESTARTS   AGE    NODE
ram-check-bti08   1/1       Running   0          4m     kube-node1
ram-check-mxry2   1/1       Running   0          4m     kube-node3
ram-check-u9e5f   1/1       Running   0          4m     kube-node2
```

It is good for you to evaluate the result using the subcommand `logs`:

```
# kubectl logs ram-check-bti08
             total       used       free     shared    buffers     cached
Mem:       2051644    1256876     794768        148     136880     450620
-/+ buffers/cache:     669376    1382268
Swap:            0          0          0
             total       used       free     shared    buffers     cached
Mem:       2051644    1255888     795756        156     136912     450832
-/+ buffers/cache:     668144    1383500
Swap:            0          0          0
:
(ignored)
```

Next, delete this daemon set by the reference of template file or by the name of the resource:

```
# kubectl stop -f daemonset-free.yaml
// or
# kubectl stop ds ram-check
```

Running the daemon set only on specific nodes

There is also a solution for you to deploy daemon-like pods simply on specified nodes. First, you have to make nodes in groups by tagging them. We will only tag node `kube-node3` with the special key-value label, which indicates the one for running the daemon set:

```
# kubectl label nodes kube-node3 runDS=ok
node "kube-node3" labeled
# kubectl get nodes
NAME          LABELS                                        STATUS
AGE
kube-node1   kubernetes.io/hostname=kube-node1            Ready     27d
kube-node2   kubernetes.io/hostname=kube-node2            Ready     27d
kube-node3   kubernetes.io/hostname=kube-node3,runDS=ok  Ready     4d
```

Then, we will select this one-member group in the template. The item `spec.template.spec.nodeSelector` can add any key-value pairs for node selection:

```
# cat daemonset-free.yaml
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: ram-check
spec:
  selector:
    app: checkRam
    version: v1
  template:
    metadata:
      labels:
        app: checkRam
        owner: Amy
        version: v1
    spec:
      nodeSelector:
        runDS: ok
      containers:
      - name: ubuntu-free
        image: ubuntu
        command: ["/bin/bash","-c","while true; do free; sleep 30; done"]
      restartPolicy: Always
```

While assigning the daemon set to a certain node group, we can run it in a single node of the three-node system:

```
# kubectl describe pods --selector=app=checkRam | grep "Node"
Node:        kube-node3/10.96.219.251
```

How it works...

Although job and daemon set are the special utilities of pods, the Kubernetes system has different managements between them and pods.

For job, its selector cannot point to the existing pod. It is for fear to take a pod controlled by the replication controller as a job. The replication controller has a desired number of pods running, which is against job's ideal situation: pods should be deleted once they finish the tasks. The pod in the replication controller won't get the state of end.

On the other hand, different from the general pod, a pod in a daemon set can be created without the Kubernetes scheduler. This concept is apparent because the daemon set only considers the labels of nodes, not their CPU or memory usages.

See also

In this recipe, we went deeply into the Kubernetes pod. Also, we used a bunch of Kubernetes configuration files. The recipe about configuration files will make you learn more about the following:

- The *Working with pods* recipe in *Chapter 2*, *Walking through Kubernetes Concepts*
- *Working with configuration files*

Working with configuration files

Kubernetes supports two different file formats YAML and JSON. Each format can describe the same function of Kubernetes.

Getting ready

Both YAML and JSON have official websites to describe the standard format.

YAML

The YAML format is very simple with less syntax rules; therefore, it is easy to read and write by a human. To know more about YAML, you can refer to the following website link:

`http://www.yaml.org/spec/1.2/spec.html`

The following example uses the YAML format to set up the `nginx` pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
```

JSON

The JSON format is also simple and easy to read by humans, but more program-friendly. Because it has data types (number, string, Boolean, and object), it is popular to exchange the data between systems. To know more about JSON, you can refer to the following website link:

`http://json.org/`

The following example of the pod is the same as the preceding YAML format, but using the JSON format:

```
{
    "apiVersion" : "v1",
    "kind" : "Pod",
    "metadata" : {
      "name" : "nginx",
      "labels": {
        "name": "nginx"
      }
    },
    "spec" : {
      "containers" : [
        {
            "name" : "nginx",
            "image" : "nginx",
            "ports" : [
               {
                 "containerPort": 80
               }
            ]
        }
      ]
    }
}
```

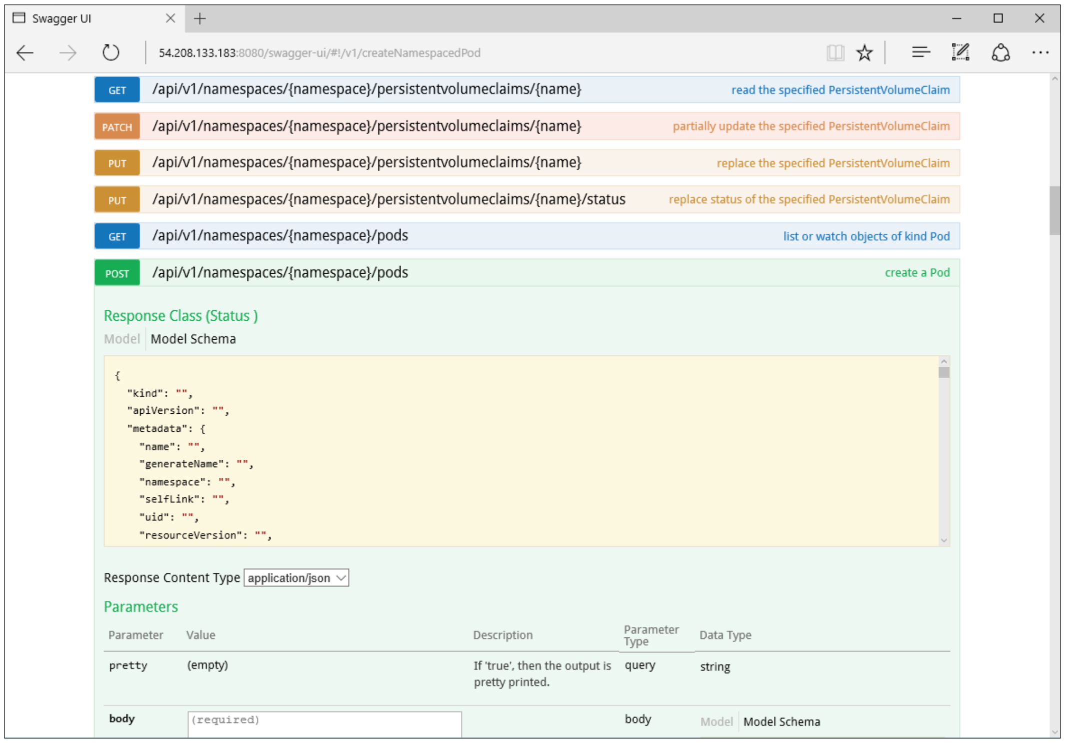
How to do it...

Kubernetes has a schema that is defined using the configuration format; schema can be generated in the `/tmp/kubectl.schema/` directory on executing the `kubectl create` command as follows:

```
# cat pod.json
{
    "apiVersion" : "v1",
    "kind" : "Pod",
    "metadata" : {
      "name" : "nginx",
      "labels": {
        "name": "nginx"
      }
    },
    "spec" : {
      "containers" : [
        {
            "name" : "nginx",
            "image" : "nginx",
            "ports" : [
               {
                 "containerPort": 80
               }
            ]
        }
      ]
    }
}

# kubectl create -f pod.json
pod "nginx" created

# ls -l /tmp/kubectl.schema/v1.1.3/api/v1/schema.json
-rw------- 2 root root 446224 Jan 24 04:50 /tmp/kubectl.schema/v1.1.3/
api/v1/schema.json
```

There is an alternative way, because Kubernetes is also using swagger (`http://swagger.io/`) to generate the REST API; therefore, you can access `swagger-ui` via `http://<kubernetes-master>:8080/swagger-ui/`.

Each configuration, for example, pods, replication controllers, and services are described in the **POST** section, as shown in the following screenshot:



GET /api/v1/namespaces/{namespace}/persistentvolumeclaims/{name} read the specified PersistentVolumeClaim

PATCH /api/v1/namespaces/{namespace}/persistentvolumeclaims/{name} partially update the specified PersistentVolumeClaim

PUT /api/v1/namespaces/{namespace}/persistentvolumeclaims/{name} replace the specified PersistentVolumeClaim

PUT /api/v1/namespaces/{namespace}/persistentvolumeclaims/{name}/status replace status of the specified PersistentVolumeClaim

GET /api/v1/namespaces/{namespace}/pods list or watch objects of kind Pod

POST /api/v1/namespaces/{namespace}/pods create a Pod

Response Class (Status)

Model | Model Schema

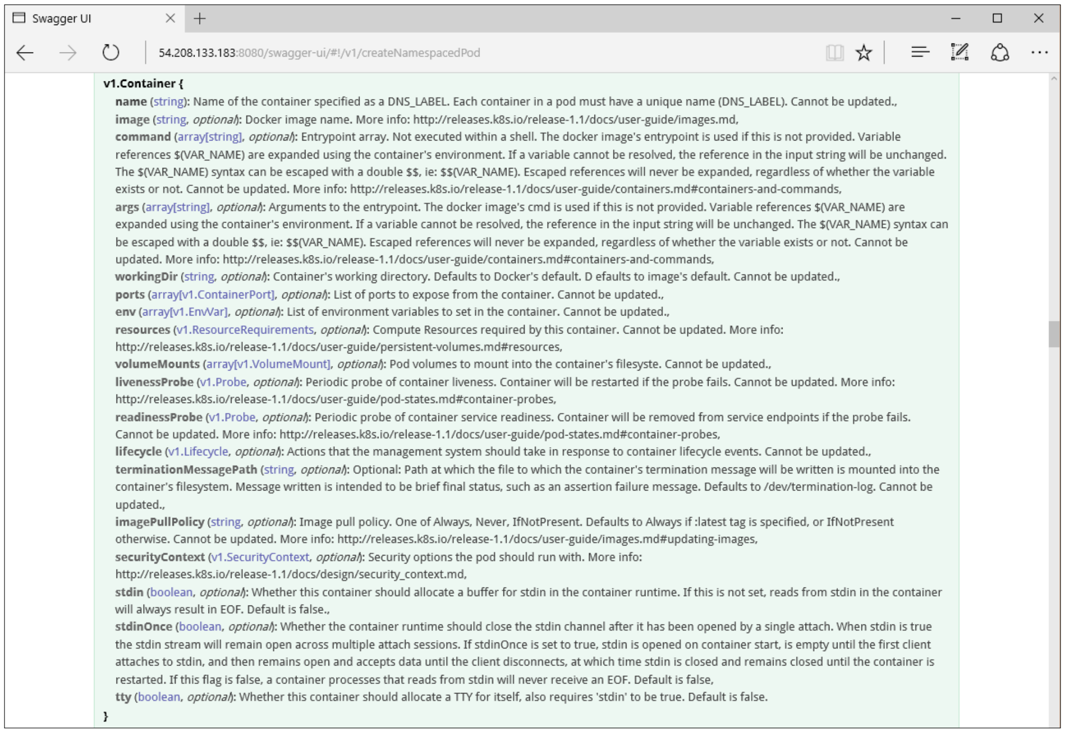
```
{
  "kind": "",
  "apiVersion": "",
  "metadata": {
    "name": "",
    "generateName": "",
    "namespace": "",
    "selfLink": "",
    "uid": "",
    "resourceVersion": "",
```

Response Content Type application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
pretty	(empty)	If 'true', then the output is pretty printed.	query	string
body	(required)		body	Model | Model Schema

The preceding screenshot of `swagger-ui`, shows the pod's definition. Types of items, such as `string`, `array`, and `integer` are shown when you click on **Model** as follows:



```
v1.Container {
  name (string): Name of the container specified as a DNS_LABEL. Each container in a pod must have a unique name (DNS_LABEL). Cannot be updated.,
  image (string, optional): Docker image name. More info: http://releases.k8s.io/release-1.1/docs/user-guide/images.md,
  command (array[string], optional): Entrypoint array. Not executed within a shell. The docker image's entrypoint is used if this is not provided. Variable
  references $(VAR_NAME) are expanded using the container's environment. If a variable cannot be resolved, the reference in the input string will be unchanged.
  The $(VAR_NAME) syntax can be escaped with a double $$, ie: $$(VAR_NAME). Escaped references will never be expanded, regardless of whether the variable
  exists or not. Cannot be updated. More info: http://releases.k8s.io/release-1.1/docs/user-guide/containers.md#containers-and-commands,
  args (array[string], optional): Arguments to the entrypoint. The docker image's cmd is used if this is not provided. Variable references $(VAR_NAME) are
  expanded using the container's environment. If a variable cannot be resolved, the reference in the input string will be unchanged. The $(VAR_NAME) syntax can
  be escaped with a double $$, ie: $$(VAR_NAME). Escaped references will never be expanded, regardless of whether the variable exists or not. Cannot be
  updated. More info: http://releases.k8s.io/release-1.1/docs/user-guide/containers.md#containers-and-commands,
  workingDir (string, optional): Container's working directory. Defaults to Docker's default. D efaults to image's default. Cannot be updated.,
  ports (array[v1.ContainerPort], optional): List of ports to expose from the container. Cannot be updated.,
  env (array[v1.EnvVar], optional): List of environment variables to set in the container. Cannot be updated.,
  resources (v1.ResourceRequirements, optional): Compute Resources required by this container. Cannot be updated. More info:
  http://releases.k8s.io/release-1.1/docs/user-guide/persistent-volumes.md#resources,
  volumeMounts (array[v1.VolumeMount], optional): Pod volumes to mount into the container's filesyste. Cannot be updated.,
  livenessProbe (v1.Probe, optional): Periodic probe of container liveness. Container will be restarted if the probe fails. Cannot be updated. More info:
  http://releases.k8s.io/release-1.1/docs/user-guide/pod-states.md#container-probes,
  readinessProbe (v1.Probe, optional): Periodic probe of container service readiness. Container will be removed from service endpoints if the probe fails.
  Cannot be updated. More info: http://releases.k8s.io/release-1.1/docs/user-guide/pod-states.md#container-probes,
  lifecycle (v1.Lifecycle, optional): Actions that the management system should take in response to container lifecycle events. Cannot be updated.,
  terminationMessagePath (string, optional): Optional: Path at which the file to which the container's termination message will be written is mounted into the
  container's filesystem. Message written is intended to be brief final status, such as an assertion failure message. Defaults to /dev/termination-log. Cannot be
  updated.,
  imagePullPolicy (string, optional): Image pull policy. One of Always, Never, IfNotPresent. Defaults to Always if :latest tag is specified, or IfNotPresent
  otherwise. Cannot be updated. More info: http://releases.k8s.io/release-1.1/docs/user-guide/images.md#updating-images,
  securityContext (v1.SecurityContext, optional): Security options the pod should run with. More info:
  http://releases.k8s.io/release-1.1/docs/design/security_context.md,
  stdin (boolean, optional): Whether this container should allocate a buffer for stdin in the container runtime. If this is not set, reads from stdin in the container
  will always result in EOF. Default is false.,
  stdinOnce (boolean, optional): Whether the container runtime should close the stdin channel after it has been opened by a single attach. When stdin is true
  the stdin stream will remain open across multiple attach sessions. If stdinOnce is set to true, stdin is opened on container start, is empty until the first client
  attaches to stdin, and then remains open and accepts data until the client disconnects, at which time stdin is closed and remains closed until the container is
  restarted. If this flag is false, a container processes that reads from stdin will never receive an EOF. Default is false,
  tty (boolean, optional): Whether this container should allocate a TTY for itself, also requires 'stdin' to be true. Default is false.
}
```

The preceding screenshot shows the pod container definition. There are many items that are defined; however, some of them are indicated as `optional`, which is not necessary and applied as a default value or not set if you don't specify it.

> Some of the items are indicated as `readonly`, such as UID. Kubernetes generates these items. If you specify this in the configuration file, it will be ignored.

How it works...

There are some mandatory items that need to be defined in each configuration file, as follows:

Pods

Item	Type	Example
apiVersion	String	v1
kind	String	Pod
metadata.name	String	my-nginx
spec	v1.PodSpec	
v1.PodSpec.containers	array[v1.Container]	
v1.Container.name	String	my-nginx
v1.Container.image	String	nginx

Therefore, the minimal pod configuration is as follows (in the YAML format):

```
apiVersion: v1
kind: Pod
metadata:
  name: my-nginx
spec:
  containers:
  - name: my-nginx
    image: nginx
```

Replication controllers

Item	Type	Example
apiVersion	String	v1
kind	String	ReplicationController
metadata.name	String	my-nginx-rc
spec	v1.ReplicationControllerSpec	
v1.ReplicationControllerSpec.template	v1.PodTemplateSpec	
v1.PodTemplateSpec.metadata.labels	Map of String	app: nginx
v1.PodTemplateSpec.spec	v1.PodSpec	
v1.PodSpec.containers	array[v1.Container]	As same as pod

The following example is the minimal configuration of the replication controller (in the YAML format):

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx-rc
spec:
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: my-nginx
        image: nginx
```

Services

Item	Type	Example
apiVersion	String	v1
kind	String	Service
metadata.name	String	my-nginx-service
spec	v1.ServiceSpec	
v1.ServiceSpec.selector	Map of String	sel: my-selector
v1.ServiceSpec.ports	array[v1.ServicePort]	
v1.ServicePort.protocol	String	TCP
v1.ServicePort.port	Integer	80

The following example is the minimal configuration of service (in the YAML format):

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx

spec:
  selector:
     sel: my-selector
  ports:
    - protocol: TCP
      port: 80
```

See also

This recipe described how to find and understand a configuration syntax. Kubernetes has some detailed options to define container and components. For more details, the following recipes will describe how to define pods, replication controllers, and services:

- The *Working with pods*, *Working with a replication controller*, and *Working with services* recipes in *Chapter 2*, *Walking through Kubernetes Concepts*

4

Building a High Availability Cluster

In this chapter, we will cover the following topics:

- Clustering etcd
- Building multiple masters

Introduction

Avoiding a single point of failure is a concept we need to always keep in mind. In this chapter, you will learn how to build components in Kubernetes with high availability. We will also go through the steps to build a three-node etcd cluster and masters with multinodes.

Clustering etcd

etcd stores network information and states in Kubernetes. Any data loss could be crucial. Clustering is strongly recommended in etcd. etcd comes with support for clustering; a cluster of *N* members can tolerate up to *(N-1)/2* failures. There are three mechanisms for creating an etcd cluster. They are as follows:

- Static
- etcd discovery
- DNS discovery

In this recipe, we will discuss how to bootstrap an etcd cluster by static and etcd discovery.

Getting ready

Before you start building an etcd cluster, you have to decide how many members you need. How big the etcd cluster should be really depends on the environment you want to create. In the production environment, at least three members are recommended. Then, the cluster can tolerate at least one permanent failure. In this recipe, we will use three members as an example of the development environment:

Name/Hostname	IP address
`ip-172-31-0-1`	`172.31.0.1`
`ip-172-31-0-2`	`172.31.0.2`
`ip-172-31-0-3`	`172.31.0.3`

How to do it...

A static mechanism is the easiest way to set up a cluster. However, the IP address of every member should be known beforehand. It means that if you bootstrap an etcd cluster in some cloud provider environment, the static mechanism might not be so practical. Therefore, etcd also provides a discovery mechanism to bootstrap itself from the existing cluster.

Static

With a static mechanism, you have to know the address information of each member:

Parameters	Meaning
`-name`	The name of this member
`-initial-advertise-peer-urls`	Used to peer with other members, should be the same as the one listing in `-initial-cluster`
`-listen-peer-urls`	The URL to accept peer traffic
`-listen-client-urls`	The URL to accept client traffic
`-advertise-client-urls`	etcd member used to advertise to other members
`-initial-cluster-token`	A unique token for distinguishing different clusters
`-initial-cluster`	Advertised peer URLs of all the members
`-initial-cluster-state`	Specifies the state of the initial cluster

Use the etcd command-line tool to bootstrap a cluster with additional parameters on each member:

```
// on the host ip-172-31-0-1, running etcd command to make it peer with
ip-172-31-0-2 and ip-172-31-0-3, advertise and listen other members via
port 2379, and accept peer traffic via port 2380
# etcd -name ip-172-31-0-1 \
       -initial-advertise-peer-urls http://172.31.0.1:2380 \
       -listen-peer-urls http://172.31.0.1:2380 \
       -listen-client-urls http://0.0.0.0:2379 \
       -advertise-client-urls http://172.31.0.1:2379 \
       -initial-cluster-token mytoken \
       -initial-cluster ip-172-31-0-1=http://172.31.0.1:2380,ip-172-31-0-
2=http://172.31.0.2:2380,ip-172-31-0-3=http://172.31.0.3:2380 \
       -initial-cluster-state new
...
2016-05-01 18:57:26.539787 I | etcdserver: starting member
e980eb6ff82d4d42 in cluster 8e620b738845cd7
2016-05-01 18:57:26.551610 I | etcdserver: starting server... [version:
2.2.5, cluster version: to_be_decided]
2016-05-01 18:57:26.553100 N | etcdserver: added member 705d980456f91652
[http://172.31.0.3:2380] to cluster 8e620b738845cd7
2016-05-01 18:57:26.553192 N | etcdserver: added member 74627c91d7ab4b54
[http://172.31.0.2:2380] to cluster 8e620b738845cd7
2016-05-01 18:57:26.553271 N | etcdserver: added local member
e980eb6ff82d4d42 [http://172.31.0.1:2380] to cluster 8e620b738845cd7
2016-05-01 18:57:26.553349 E | rafthttp: failed to dial 705d980456f91652
on stream MsgApp v2 (dial tcp 172.31.0.3:2380: getsockopt: connection
refused)
2016-05-01 18:57:26.553392 E | rafthttp: failed to dial 705d980456f91652
on stream Message (dial tcp 172.31.0.3:2380: getsockopt: connection
refused)
2016-05-01 18:57:26.553424 E | rafthttp: failed to dial 74627c91d7ab4b54
on stream Message (dial tcp 172.31.0.2:2380: getsockopt: connection
refused)
2016-05-01 18:57:26.553450 E | rafthttp: failed to dial 74627c91d7ab4b54
on stream MsgApp v2 (dial tcp 172.31.0.2:2380: getsockopt: connection
refused)
```

The `etcd` daemon on `ip-172-31-0-1` will then start checking whether all the members are online. The logs show connection refused since `ip-172-31-0-2` and `ip-172-31-0-3` are still offline. Let's go to the next member and run the `etcd` command:

```
// on the host ip-172-31-0-2, running etcd command to make it peer with
ip-172-31-0-1 and ip-172-31-0-3, advertise and listen other members via
port 2379, and accept peer traffic via port 2380

# etcd -name ip-172-31-0-2 \

       -initial-advertise-peer-urls http://172.31.0.2:2380 \

       -listen-peer-urls http://172.31.0.2:2380 \

       -listen-client-urls http://0.0.0.0:2379 \

       -advertise-client-urls http://172.31.0.2:2379 \

       -initial-cluster-token mytoken \

       -initial-cluster ip-172-31-0-1=http://172.31.0.1:2380,ip-172-31-0-
2=http://172.31.0.2:2380, ip-172-31-0-3=http://172.31.0.3:2380 -initial-
cluster-state new

...

2016-05-01 22:59:55.696357 I | etcdserver: starting member
74627c91d7ab4b54 in cluster 8e620b738845cd7

2016-05-01 22:59:55.696397 I | raft: 74627c91d7ab4b54 became follower at
term 0

2016-05-01 22:59:55.696407 I | raft: newRaft 74627c91d7ab4b54 [peers: [],
term: 0, commit: 0, applied: 0, lastindex: 0, lastterm: 0]

2016-05-01 22:59:55.696411 I | raft: 74627c91d7ab4b54 became follower at
term 1

2016-05-01 22:59:55.706552 I | etcdserver: starting server... [version:
2.2.5, cluster version: to_be_decided]

2016-05-01 22:59:55.707627 E | rafthttp: failed to dial 705d980456f91652
on stream MsgApp v2 (dial tcp 172.31.0.3:2380: getsockopt: connection
refused)

2016-05-01 22:59:55.707690 N | etcdserver: added member 705d980456f91652
[http://172.31.0.3:2380] to cluster 8e620b738845cd7

2016-05-01 22:59:55.707754 N | etcdserver: added local member
74627c91d7ab4b54 [http://172.31.0.2:2380] to cluster 8e620b738845cd7

2016-05-01 22:59:55.707820 N | etcdserver: added member e980eb6ff82d4d42
[http://172.31.0.1:2380] to cluster 8e620b738845cd7

2016-05-01 22:59:55.707873 E | rafthttp: failed to dial 705d980456f91652
on stream Message (dial tcp 172.31.0.3:2380: getsockopt: connection
refused)

2016-05-01 22:59:55.708433 I | rafthttp: the connection with
e980eb6ff82d4d42 became active
```

```
2016-05-01 22:59:56.196750 I | raft: 74627c91d7ab4b54 is starting a new
election at term 1

2016-05-01 22:59:56.196903 I | raft: 74627c91d7ab4b54 became candidate at
term 2

2016-05-01 22:59:56.196946 I | raft: 74627c91d7ab4b54 received vote from
74627c91d7ab4b54 at term 2

2016-05-01 22:59:56.949201 I | raft: raft.node: 74627c91d7ab4b54 elected
leader e980eb6ff82d4d42 at term 112

2016-05-01 22:59:56.961883 I | etcdserver: published {Name:ip-172-31-0-2
ClientURLs:[http://10.0.0.2:2379]} to cluster 8e620b738845cd7

2016-05-01 22:59:56.966981 N | etcdserver: set the initial cluster
version to 2.1
```

After starting member 2, we can see that the current cluster version is `2.1`. The following error message shows the connection to peer `705d980456f91652` is unhealthy. By observing the log, we can find that member `705d980456f91652` is pointing to `http://172.31.0.3:2380`. Let's start up the last member `ip-172-31-0-3`:

```
# etcd -name ip-172-31-0-3 \
       -initial-advertise-peer-urls http://172.31.0.3:2380 \
       -listen-peer-urls http://172.31.0.3:2380 \
       -listen-client-urls http://0.0.0.0:2379 \
       -advertise-client-urls http://172.31.0.3:2379 \
       -initial-cluster-token mytoken \
       -initial-cluster ip-172-31-0-1=http://172.31.0.1:2380,ip-172-31-0-
2=http://172.31.0.2:2380, ip-172-31-0-3=http://172.31.0.3:2380 -initial-
cluster-state new

2016-05-01 19:02:19.106540 I | etcdserver: starting member
705d980456f91652 in cluster 8e620b738845cd7

2016-05-01 19:02:19.106590 I | raft: 705d980456f91652 became follower at
term 0

2016-05-01 19:02:19.106608 I | raft: newRaft 705d980456f91652 [peers: [],
term: 0, commit: 0, applied: 0, lastindex: 0, lastterm: 0]

2016-05-01 19:02:19.106615 I | raft: 705d980456f91652 became follower at
term 1

2016-05-01 19:02:19.118330 I | etcdserver: starting server... [version:
2.2.5, cluster version: to_be_decided]

2016-05-01 19:02:19.120729 N | etcdserver: added local member
705d980456f91652 [http://10.0.0.75:2380] to cluster 8e620b738845cd7

2016-05-01 19:02:19.120816 N | etcdserver: added member 74627c91d7ab4b54
[http://10.0.0.204:2380] to cluster 8e620b738845cd7

2016-05-01 19:02:19.120887 N | etcdserver: added member e980eb6ff82d4d42
[http://10.0.0.205:2380] to cluster 8e620b738845cd7
```

```
2016-05-01 19:02:19.121566 I | rafthttp: the connection with
74627c91d7ab4b54 became active

2016-05-01 19:02:19.121690 I | rafthttp: the connection with
e980eb6ff82d4d42 became active

2016-05-01 19:02:19.143351 I | raft: 705d980456f91652 [term: 1] received
a MsgHeartbeat message with higher term from e980eb6ff82d4d42 [term: 112]

2016-05-01 19:02:19.143380 I | raft: 705d980456f91652 became follower at
term 112

2016-05-01 19:02:19.143403 I | raft: raft.node: 705d980456f91652 elected
leader e980eb6ff82d4d42 at term 112

2016-05-01 19:02:19.146582 N | etcdserver: set the initial cluster
version to 2.1

2016-05-01 19:02:19.151353 I | etcdserver: published {Name:ip-172-31-0-3
ClientURLs:[http://10.0.0.75:2379]} to cluster 8e620b738845cd7

2016-05-01 19:02:22.022578 N | etcdserver: updated the cluster version
from 2.1 to 2.2
```

We can see, on member 3, we successfully initiated the etcd cluster without any errors and the current cluster version is `2.2`. How about member 1 now?

```
2016-05-01 19:02:19.118910 I | rafthttp: the connection with
705d980456f91652 became active

2016-05-01 19:02:22.014958 I | etcdserver: updating the cluster version
from 2.1 to 2.2

2016-05-01 19:02:22.018530 N | etcdserver: updated the cluster version
from 2.1 to 2.2
```

With member 2 and 3 online, member 1 can now get connected and go online too. When observing the log, we can see the leader election took place in the etcd cluster:

```
ip-172-31-0-1: raft: raft.node: e980eb6ff82d4d42 (ip-172-31-0-1) elected
leader e980eb6ff82d4d42 (ip-172-31-0-1) at term 112

ip-172-31-0-2: raft: raft.node: 74627c91d7ab4b54 (ip-172-31-0-2) elected
leader e980eb6ff82d4d42 (ip-172-31-0-1) at term 112

ip-172-31-0-3: 2016-05-01 19:02:19.143380 I | raft: 705d980456f91652
became follower at term 112
```

The etcd cluster will send the heartbeat to the members in the cluster to check the health status. Note that when you need to add or remove any members in the cluster, the preceding `etcd` command needs to be rerun on all the members in order to notify that there are new members joining the cluster. In this way, all the members in the cluster are aware of all the online members; if one node goes offline, the other members will poll the failure member until it refreshes members by the `etcd` command. If we set a message from a member, we could get the same message from the other members too. If one member becomes unhealthy, the other members in the etcd cluster will still be in service and elect for a new leader.

etcd discovery

Before using the etcd discovery, you should have a discovery URL that is used to bootstrap a cluster. If you want to add or remove a member, you should use the `etcdctl` command as the runtime reconfiguration. The command line is pretty much the same as the static mechanism. What we need to do is change `-initial-cluster` to `-discovery`, which is used to specify the discovery service URL. We could use the etcd discovery service (`https://discovery.etcd.io`) to request a discovery URL:

```
// get size=3 cluster url from etcd discovery service
# curl -w "\n" 'https://discovery.etcd.io/new?size=3'
https://discovery.etcd.io/be7c1938bbde83358d8ae978895908bd
// Init a cluster via requested URL
# etcd -name ip-172-31-0-1 -initial-advertise-peer-urls
http://172.31.43.209:2380 \
 -listen-peer-urls http://172.31.0.1:2380 \
 -listen-client-urls http://0.0.0.0:2379 \
 -advertise-client-urls http://172.31.0.1:2379 \
 -discovery https://discovery.etcd.io/be7c1938bbde83358d8ae978895908bd
...
2016-05-02 00:28:08.545651 I | etcdmain: listening for peers on
http://172.31.0.1:2380
2016-05-02 00:28:08.545756 I | etcdmain: listening for client requests on
http://127.0.0.1:2379
2016-05-02 00:28:08.545807 I | etcdmain: listening for client requests on
http://172.31.0.1:2379
2016-05-02 00:28:09.199987 N | discovery: found self e980eb6ff82d4d42 in
the cluster
2016-05-02 00:28:09.200010 N | discovery: found 1 peer(s), waiting for 2
more
```

The first member has joined the cluster; wait for the other two peers. Let's start `etcd` in the second node:

```
# etcd -name ip-172-31-0-2 -initial-advertise-peer-urls
http://172.31.0.2:2380 \
 -listen-peer-urls http://172.31.0.2:2380 \
 -listen-client-urls http://0.0.0.0:2379 \
 -advertise-client-urls http://172.31.0.2:2379 \
 -discovery https://discovery.etcd.io/be7c1938bbde83358d8ae978895908bd
...
```

```
2016-05-02 00:30:12.919005 I | etcdmain: listening for peers on
http://172.31.0.2:2380

2016-05-02 00:30:12.919074 I | etcdmain: listening for client requests on
http://0.0.0.0:2379

2016-05-02 00:30:13.018160 N | discovery: found self 25fc8075ab1ed17e in
the cluster

2016-05-02 00:30:13.018235 N | discovery: found 1 peer(s), waiting for 2
more

2016-05-02 00:30:22.985300 N | discovery: found peer e980eb6ff82d4d42 in
the cluster

2016-05-02 00:30:22.985396 N | discovery: found 2 peer(s), waiting for 1
more
```

We know there are two members in etcd already and they are waiting for the last one to join. The following code starts the last node:

```
# etcd -name ip-172-31-0-3 -initial-advertise-peer-urls
http://172.31.0.3:2380 \

 -listen-peer-urls http://172.31.0.3:2380 \

 -listen-client-urls http://0.0.0.0:2379 \

 -advertise-client-urls http://172.31.0.3:2379 \

 -discovery https://discovery.etcd.io/be7c1938bbde83358d8ae978895908bd
```

After new nodes join, we can check from the logs that there is a new election taking place:

```
2016-05-02 00:31:01.152215 I | raft: e980eb6ff82d4d42 is starting a new
election at term 308

2016-05-02 00:31:01.152272 I | raft: e980eb6ff82d4d42 became candidate at
term 309

2016-05-02 00:31:01.152281 I | raft: e980eb6ff82d4d42 received vote from
e980eb6ff82d4d42 at term 309

2016-05-02 00:31:01.152292 I | raft: e980eb6ff82d4d42 [logterm: 304,
index: 9739] sent vote request to 705d980456f91652 at term 309

2016-05-02 00:31:01.152302 I | raft: e980eb6ff82d4d42 [logterm: 304,
index: 9739] sent vote request to 74627c91d7ab4b54 at term 309

2016-05-02 00:31:01.162742 I | rafthttp: the connection with
74627c91d7ab4b54 became active

2016-05-02 00:31:01.197820 I | raft: e980eb6ff82d4d42 received vote from
74627c91d7ab4b54 at term 309

2016-05-02 00:31:01.197852 I | raft: e980eb6ff82d4d42 [q:2] has received
2 votes and 0 vote rejections

2016-05-02 00:31:01.197882 I | raft: e980eb6ff82d4d42 became leader at
term 309
```

With the discovery method, we can see that the cluster can be launched without knowing the others' IPs beforehand. etcd will start a new election if new nodes join or leave, and always keep the service online with the multi-nodes setting.

See also

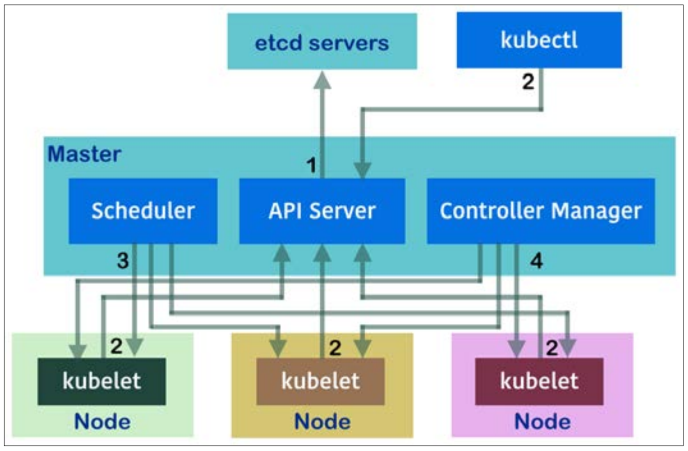
To understand the installation of a single etcd server, refer to the *Building datastore* recipe in *Chapter 1*, *Building Your Own Kubernetes*.

Building multiple masters

The master node serves as a kernel component in the Kubernetes system. Its duties include the following:

- Pushing and pulling information from the datastore and the etcd servers
- Being the portal for requests
- Assigning tasks to nodes
- Monitoring the running tasks

Three major daemons support the master fulfilling the preceding duties, which are numbered in the following image:



As you can see, the master is the communicator between workers and clients. Therefore, it will be a problem if the master node crashes. A multiple-master Kubernetes system is not only fault tolerant, but also workload-balanced. There will be no longer only one API server for accessing nodes and clients sending requests. Several API server daemons in separated master nodes would help to solve the tasks simultaneously and shorten the response time.

Getting ready

The brief concepts for building a multiple-master system are listed here:

- Add a load balancer server in front of the masters. The load balancer will become the new endpoint accessed by nodes and clients.
- Every master runs its own API server daemon.
- Only one scheduler and one controller manager are in the system, which can avoid conflict directions from different daemons while managing containers.
- `Pod master` is a new daemon installed in every master. It elects to decide the master node-running daemon scheduler and the master node-running controller manager. It could be the same master that runs both the daemons.
- Make a more flexible way to run a daemon scheduler and a controller manager as a container. Install kubelet in master and manage the daemons as pods by configuring files.

In this recipe, we are going to build a two-master system, which has similar methods while scaling more masters.

How to do it...

Now, we will guide you step by step in building a multiple-master system. Before this, you have to deploy a load balancer server for masters.

> To learn about deploying the load balancer and to build the system on AWS, please check the *Building the Kubernetes infrastructure in AWS* recipe in *Chapter 6*, *Building Kubernetes on AWS* on how to build a master load balancer.

Preparing multiple master nodes

First, install another master node in your previous Kubernetes system, which should be in the same environment as the original master. Then, stop the daemon services of scheduler and controller manager in both the masters:

- For the systemd-controlled system, stop the services directly using the commands `systemctl kube-scheduler stop` and `systemctl kube-controller-manager stop`

- For the init service-controlled system, stop the master service first. Next, delete or comment on the lines about scheduler and controller manager in the initialization script:

```
// Checking current daemon processes on master server
# service kubernetes-master status
kube-apiserver (pid 3137) is running...
kube-scheduler (pid 3138) is running...
kube-controller-manager (pid 3136) is running...
# service kubernetes-master stop
Shutting down /usr/local/bin/kube-controller-manager:       [  OK
]
Shutting down /usr/local/bin/kube-scheduler:                [  OK
]
Shutting down /usr/local/bin/kube-apiserver:                [  OK
]
// Or, for "hypercube" command with init script, we block out
scheduler and controller-manager. Just leave apiserver daemon in
master node.
// Put comment on the scheduler and controller manager daemons
// the variable $prog is /usr/local/bin/hyperkube
# cat /etc/init.d/kubernetes-master
(ignored above parts)
       # Start daemon.
       echo $"Starting apiserver: "
       daemon $prog apiserver \
       --service-cluster-ip-range=${CLUSTER_IP_RANGE} \
       --insecure-port=8080 \
       --secure-port=6443 \
       --address=0.0.0.0 \
       --etcd_servers=${ETCD_SERVERS} \
       --cluster_name=${CLUSTER_NAME} \
       > ${logfile}-apiserver.log 2>&1 &

#       echo $"Starting controller-manager: "
#       daemon $prog controller-manager \
#       --master=${MASTER} \
#       > ${logfile}-controller-manager.log 2>&1 &
#
#       echo $"Starting scheduler: "
#       daemon $prog scheduler \
#       --master=${MASTER} \
#       > ${logfile}-scheduler.log 2>&1 &
(ignored below parts)
# service kubernetes-master start
Starting apiserver:
```

At this step, you have two masters serving in the system with two processes of the API server.

Setting up kubelet in master

Because we are going to install the daemons' scheduler and controller manager as pods, a kubelet process is a must-have daemon. Download the latest (version `1.1.4`) kubelet binary file (`https://storage.googleapis.com/kubernetes-release/release/v1.1.4/bin/linux/amd64/kubelet`) and put it under the directory of the system's binary files:

```
# wget https://storage.googleapis.com/kubernetes-release/release/v1.1.4/
bin/linux/amd64/kubelet
# chmod 755 kubelet
# mv kubelet /usr/local/bin/
```

Alternatively, for the RHEL system, you can download `kubelet` from the YUM repository:

```
# yum install kubernetes-node
```

Later, we will configure the `kubelet` daemon with specific parameters and values:

Tag Name	Value	Purpose
`--api-servers`	`127.0.0.1:8080`	To communicate with the API server in local.
`--register-node`	`false`	Avoid registering this master, local host, as a node.
`--allow-privileged`	`true`	To allow containers to request the privileged mode, which means containers have the ability to access the host device, especially, the network device in this case.
`--config`	`/etc/kubernetes/manifests`	To manage local containers by the template files under this specified directory.

If your system is monitored by `systemctl`, put the preceding parameters in the configuration files:

- In `/etc/kubernetes/config`:
  - Modify `KUBE_MASTER` to `--master=127.0.0.1:8080`:

```
KUBE_LOGTOSTDERR="--logtostderr=true"
KUBE_LOG_LEVEL="--v=0"
KUBE_ALLOW_PRIV="--allow_privileged=false"
KUBE_MASTER="--master=127.0.0.1:8080"
```

- In `/etc/kubernetes/kubelet`:
  - Put the tag `--api-servers` to variable `KUBELET_API_SERVER`.
  - Put the other three tags to variable `KUBELET_ARGS`:

```
KUBELET_ADDRESS="--address=0.0.0.0"
KUBELET_HOSTNAME="--hostname_override=127.0.0.1"
KUBELET_API_SERVER="--api_servers=127.0.0.1:8080"
KUBELET_ARGS="--register-node=false --allow-privileged=true
--config /etc/kubernetes/manifests"
```

On the other hand, modify your script file of `init` service management and append the tags after the daemon `kubelet`. For example, we have the following settings in `/etc/init.d/kubelet`:

```
# cat /etc/init.d/kubelet
prog=/usr/local/bin/kubelet
lockfile=/var/lock/subsys/`basename $prog`
hostname=`hostname`
logfile=/var/log/kubernetes.log

start() {
       # Start daemon.
       echo $"Starting kubelet: "
       daemon $prog \
               --api-servers=127.0.0.1:8080 \
               --register-node=false \
               --allow-privileged=true \
               --config=/etc/kubernetes/manifests \
               > ${logfile} 2>&1 &
     (ignored)
```

It is fine to keep your kubelet service in the stopped state, since we will start it after the configuration files of scheduler and the controller manager are ready.

Getting the configuration files ready

We need three templates as configuration files: pod master, scheduler, and controller manager. These files should be put at specified locations.

Pod master handles the elections to decide which master runs the scheduler daemon and which master runs the controller manager daemon. The result will be recorded in the etcd servers. The template of pod master is put in the kubelet config directory, making sure that the pod master is created right after kubelet starts running:

```
# cat /etc/kubernetes/manifests/podmaster.yaml
apiVersion: v1
kind: Pod
metadata:
 name: podmaster
 namespace: kube-system
spec:
 hostNetwork: true
 containers:
 - name: scheduler-elector
   image: gcr.io/google_containers/podmaster:1.1
   command: ["/podmaster", "--etcd-servers=<ETCD_ENDPOINT>",
"--key=scheduler", "--source-file=/kubernetes/kube-scheduler.yaml",
"--dest-file=/manifests/kube-scheduler.yaml"]
   volumeMounts:
   - mountPath: /kubernetes
     name: k8s
     readOnly: true
   - mountPath: /manifests
     name: manifests
 - name: controller-manager-elector
   image: gcr.io/google_containers/podmaster:1.1
   command: ["/podmaster", "--etcd-servers=<ETCD_ENDPOINT>",
"--key=controller", "--source-file=/kubernetes/kube-controller-manager.
yaml", "--dest-file=/manifests/kube-controller-manager.yaml"]
   terminationMessagePath: /dev/termination-log
   volumeMounts:
   - mountPath: /kubernetes
     name: k8s
     readOnly: true
   - mountPath: /manifests
     name: manifests
 volumes:
 - hostPath:
     path: /srv/kubernetes
   name: k8s
 - hostPath:
     path: /etc/kubernetes/manifests
   name: manifests
```

In the configuration file of pod master, we will deploy a pod with two containers, the two electors for different daemons. The pod `podmaster` is created in a new namespace called `kube-system` in order to separate pods for daemons and applications. We will need to create a new namespace prior to creating resources using templates. It is also worth mentioning that the path `/srv/kubernetes` is where we put the daemons' configuration files. The content of the files is like the following lines:

```
# cat /srv/kubernetes/kube-scheduler.yaml
apiVersion: v1
kind: Pod
metadata:
 name: kube-scheduler
 namespace: kube-system
spec:
 hostNetwork: true
 containers:
 - name: kube-scheduler
   image: gcr.io/google_containers/kube-scheduler:34d0b8f8b31e27937327961
528739bc9
   command:
   - /bin/sh
   - -c
   - /usr/local/bin/kube-scheduler --master=127.0.0.1:8080 --v=2 1>>/var/
log/kube-scheduler.log 2>&1
```

```
   livenessProbe:
     httpGet:
       path: /healthz
       port: 10251
     initialDelaySeconds: 15
     timeoutSeconds: 1
   volumeMounts:
   - mountPath: /var/log/kube-scheduler.log
     name: logfile
   - mountPath: /usr/local/bin/kube-scheduler
     name: binfile
 volumes:
 - hostPath:
     path: /var/log/kube-scheduler.log
   name: logfile
 - hostPath:
     path: /usr/local/bin/kube-scheduler
   name: binfile
```

There are some special items set in the template, such as namespace and two mounted files. One is a log file; the streaming output can be accessed and saved in the local side. The other one is the execution file. The container can make use of the latest kube-scheduler on the local host:

```
# cat /srv/kubernetes/kube-controller-manager.yaml
apiVersion: v1
kind: Pod
metadata:
 name: kube-controller-manager
 namespace: kube-system
spec:
 containers:
 - command:
   - /bin/sh
   - -c
   - /usr/local/bin/kube-controller-manager --master=127.0.0.1:8080
--cluster-cidr=<KUBERNETES_SYSTEM_CIDR> --allocate-node-cidrs=true --v=2
1>>/var/log/kube-controller-manager.log 2>&1
```

```
    image: gcr.io/google_containers/kube-controller-manager:fda24638d51a48
baa13c35337fcd4793
    livenessProbe:
      httpGet:
        path: /healthz
        port: 10252
      initialDelaySeconds: 15
      timeoutSeconds: 1
    name: kube-controller-manager
    volumeMounts:
    - mountPath: /srv/kubernetes
      name: srvkube
      readOnly: true
    - mountPath: /var/log/kube-controller-manager.log
      name: logfile
    - mountPath: /usr/local/bin/kube-controller-manager
      name: binfile
  hostNetwork: true
  volumes:
  - hostPath:
      path: /srv/kubernetes
    name: srvkube
  - hostPath:
      path: /var/log/kube-controller-manager.log
    name: logfile
  - hostPath:
      path: /usr/local/bin/kube-controller-manager
    name: binfile
```

The configuration file of the controller manager is similar to the one of the scheduler. Remember to provide the CIDR range of your Kubernetes system in the daemon command.

For the purpose of having your templates work successfully, there are still some preconfigurations required before you start the pod master:

- Create empty log files. Otherwise, instead of the file format, the container will regard the path as a directory and cause the error of pod creation:

  ```
  // execute these commands on each master
  # touch /var/log/kube-scheduler.log
  # touch /var/log/kube-controller-manager.log
  ```

- Create the new namespace. The new namespace is separated from the default one. We are going to put the pod for system usage in this namespace:

  ```
  // Just execute this command in a master, and other masters can
  share this update.
  # kubectl create namespace kube-system
  // Or
  # curl -XPOST -d'{"apiVersion":"v1","kind":"Namespace","metadata":
  {"name":"kube-system"}}' "http://127.0.0.1:8080/api/v1/namespaces"
  ```

Starting the kubelet service and turning daemons on!

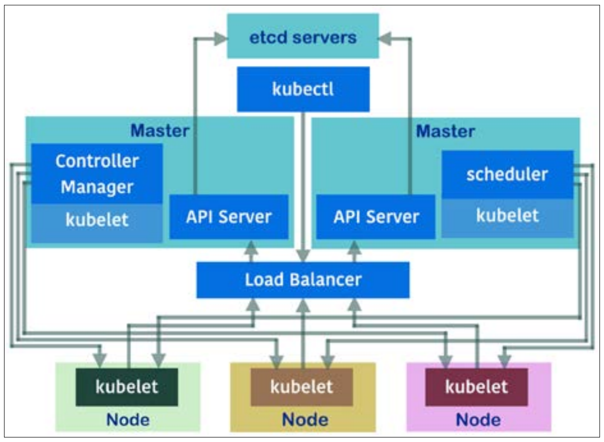
Before starting kubelet for our pod master and two master-owned daemons, please make sure you have Docker and flanneld started first:

```
# Now, it is good to start kubelet on every masters
# service kubelet start
```

Wait for a while; you will get a pod master running on each master and you will finally get a pair of scheduler and controller manager:

```
# Check pods at namespace "kube-system"
# kubectl get pod --namespace=kube-system
NAME                                   READY     STATUS     RESTARTS     AGE
kube-controller-manager-kube-master1   1/1       Running    0            3m
kube-scheduler-kube-master2            1/1       Running    0            3m
podmaster-kube-master1                 2/2       Running    0            1m
podmaster-kube-master2                 2/2       Running    0            1m
```

Congratulations! You have your multiple-master Kubernetes system built up successfully. And the structure of the machines looks like following image:



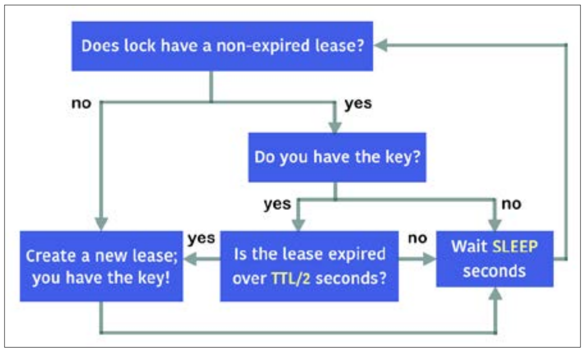
You can see that now, a single node does not have to deal with the whole request load. Moreover, the daemons are not crowded in a master; they can be distributed to different masters and every master has the ability to do the recovery. Try to shut down one master; you will find that your scheduler and controller manager are still providing services.

How it works...

Check the log of the container pod master; you will get two kinds of messages, one for who is holding the key and one without a key on hand:

```
// Get the log with specified container name
# kubectl logs podmaster-kube-master1 -c scheduler-elector
--namespace=kube-system
I0211 15:13:46.857372       1 podmaster.go:142] --whoami is empty,
defaulting to kube-master1
I0211 15:13:47.168724       1 podmaster.go:82] key already exists, the
master is kube-master2, sleeping.
I0211 15:13:52.506880       1 podmaster.go:82] key already exists, the
master is kube-master2, sleeping.
(ignored)
# kubectl logs podmaster-kube-master1 -c controller-manager-elector
--namespace=kube-system
I0211 15:13:44.484201       1 podmaster.go:142] --whoami is empty,
defaulting to kube-master1
I0211 15:13:50.078994       1 podmaster.go:73] key already exists, we are
the master (kube-master1)
I0211 15:13:55.185607       1 podmaster.go:73] key already exists, we are
the master (kube-master1)
(ignored)
```

The master with the key should take charge of the specific daemon and the said scheduler or controller manager. This current high-availability solution for the master is realized by the lease-lock method in etcd:


Does lock have a non-expired lease?
no
yes
Do you have the key?
yes
no
Create a new lease; you have the key!
yes
Is the lease expired over TTL/2 seconds?
no
Wait SLEEP seconds

The preceding loop image indicates the progress of the lease-lock method. Two time periods are important in this method: **SLEEP** is the period for checking lock, and **Time to Live** (**TTL**) is the period of lease expiration. We can say that if the daemon-running master crashed, the worst case for the other master taking over its job requires the time **SLEEP** + **TTL**. By default, **SLEEP** is 5 seconds and **TTL** is 30 seconds.

You can still take a look at the source code of pod master for more concepts (`podmaster.go`: `https://github.com/kubernetes/contrib/blob/master/pod-master/podmaster.go`).

See also

Before you read this recipe, you should have the basic concept of single master installation. Refer to the related recipes mentioned here and get an idea of how to build a multiple-master system automatically:

- The *Configuring master* recipe in *Chapter 1*, *Building Your Own Kubernetes*
- *Clustering etcd*
- The *Building the Kubernetes infrastructure in AWS* recipe in *Chapter 6*, *Building Kubernetes on AWS*

5
Building a Continuous Delivery Pipeline

In this chapter, we will cover the following topics:

- Moving monolithic to microservices
- Integrating with Jenkins
- Working with the private Docker registry
- Setting up the Continuous Delivery pipeline

Introduction

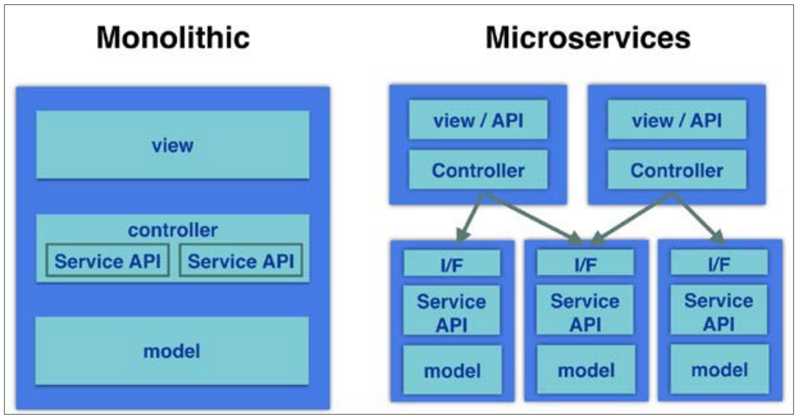
Kubernetes is a perfect match with applications featuring the microservices architecture. However, most of the old applications are all built in monolithic style. We will give you the idea about how to move from monolithic to the microservices world. As for microservices, deployment will become a burden if you are doing it manually. We will then learn how to build up our own Continuous Delivery pipeline by coordinating Jenkins, the Docker registry, and Kubernetes.

Moving monolithic to microservices

Typically, application architecture was the monolithic design that contains **Model-View-Controller** (**MVC**) and every component within a single big binary. Monolithic has some benefits, such as less latency within components, all in one straightforward packaging, and being easy to deploy and test.

However, a monolithic design has some downsides because the binary will be getting bigger and bigger. You always need to take care of the side effects when adding or modifying the code, therefore, making release cycles longer.

Containers and Kubernetes give more flexibility in using microservices for your application. The microservices architecture is very simple that can be divided into some modules or some service classes with MVC together.



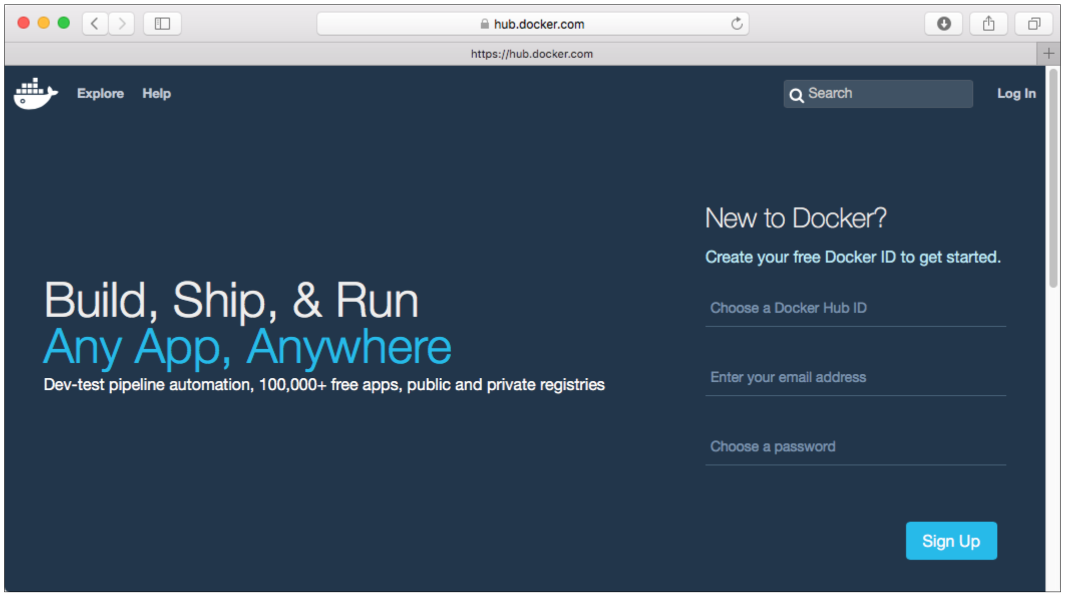
Monolithic and microservices design

Each microservice provides **Remote Procedure Call** (**RPC**) using RESTful or some standard network APIs to other microservices. The benefit is that each microservice is independent. There are minimal side effects when adding or modifying the code. Release the cycle independently, so it perfectly fits with the Agile software development methodology and allows to reuse these microservices to construct another application that builds the microservices ecosystem.

Getting ready

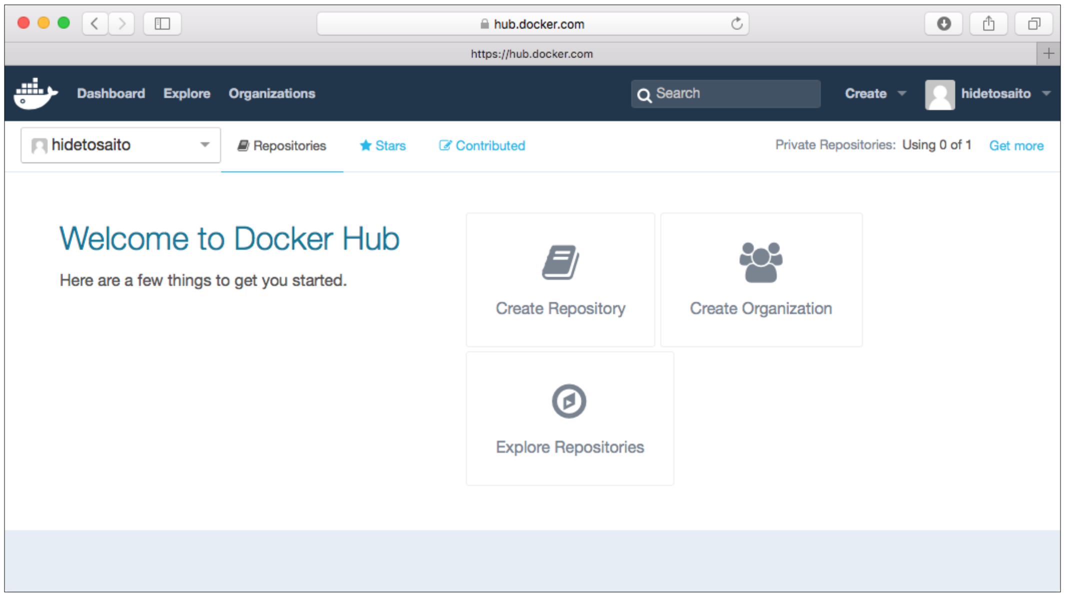
Prepare the simple microservices program. In order to push and pull your microservices, please register to Docker Hub (`https://hub.docker.com/`) to create your free Docker Hub ID in advance:

Attention: If you push the Docker image to Docker Hub, it will be public; anyone can pull your image. Therefore, don't put any confidential information into the image.



Docker Hub registration page

Once you successfully log in to your Docker Hub ID, you will be redirected to your **Dashboard** page as follows:



After logging to Docker Hub

How to do it...

Prepare both microservices and the Frontend WebUI as a Docker image. Then, deploy them using the Kubernetes replication controller and service.

Microservices

1. Here is the simple microservice using Python Flask (`http://flask.pocoo.org/`):

```
$ cat entry.py
from flask import Flask, request

app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"

@app.route("/power/<int:base>/<int:index>")
def power(base, index):
    return "%d" % (base ** index)

@app.route("/addition/<int:x>/<int:y>")
def add(x, y):
    return "%d" % (x+y)

@app.route("/substraction/<int:x>/<int:y>")
def substract(x, y):
    return "%d" % (x-y)


if __name__ == "__main__":
    app.run(host='0.0.0.0')
```

2. Prepare a `Dockerfile` as follows in order to build the Docker image:

```
$ cat Dockerfile
FROM ubuntu:14.04
```

```
# Update packages
RUN apt-get update -y

# Install Python Setuptools
RUN apt-get install -y python-setuptools git telnet curl

# Install pip
RUN easy_install pip

# Bundle app source
ADD . /src
WORKDIR /src

# Add and install Python modules
RUN pip install Flask

# Expose
EXPOSE  5000

# Run
CMD ["python", "entry.py"]
```

3. Then, use the `docker build` command to build the Docker image as follows:

[If you publish the Docker image, you should use `Docker Hub ID/image name` as the Docker image name.]

```
//name as "your_docker_hub_id/my-calc"
$ sudo docker build -t hidetosaito/my-calc .
Sending build context to Docker daemon 3.072 kB
Step 1 : FROM ubuntu:14.04
 ---> 6cc0fc2a5ee3
Step 2 : RUN apt-get update -y
 ---> Using cache
```

```
(snip)

Step 8 : EXPOSE 5000
 ---> Running in 7c52f4bfe373
 ---> 28f79bb7481f
Removing intermediate container 7c52f4bfe373
Step 9 : CMD python entry.py
 ---> Running in 86b39c727572
 ---> 20ae465bf036
Removing intermediate container 86b39c727572
Successfully built 20ae465bf036

//verity your image
$ sudo docker images
REPOSITORY            TAG                 IMAGE ID
CREATED             VIRTUAL SIZE
hidetosaito/my-calc   latest              20ae465bf036        19
seconds ago      284 MB
ubuntu                14.04               6cc0fc2a5ee3        3
weeks ago          187.9 MB
```

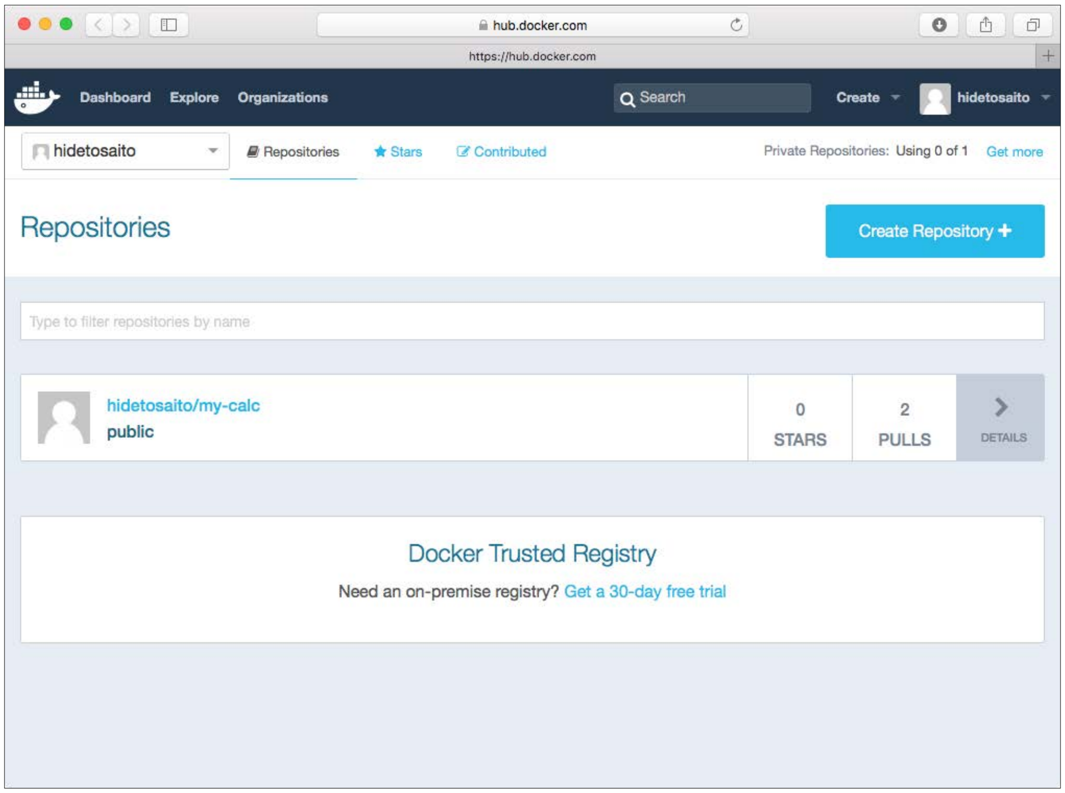
4. Then, use the `docker login` command to log in to Docker Hub:

```
//type your username, password and e-mail address in Docker hub
$ sudo docker login
Username: hidetosaito
Password:
Email: hideto.saito@yahoo.com
WARNING: login credentials saved in /home/ec2-user/.docker/config.
json
Login Succeeded
```

5. Finally, use the `docker push` command to register to your Docker Hub repository as follows:

```
//push to your docker index
$ sudo docker push hidetosaito/my-calc
The push refers to a repository [docker.io/hidetosaito/my-calc]
(len: 1)
20ae465bf036: Pushed
(snip)
92ec6d044cb3: Pushed
latest: digest: sha256:203b81c5a238e228c154e0b53a58e60e6eb3d156329
3483ce58f48351031a474 size: 19151
```

On accessing Docker Hub, you can see your microservices in the repository:



Microservice image on Docker Hub

Frontend WebUI

1. Here is the simple Frontend WebUI that is also using Python Flask:

```
import os
import httplib
from flask import Flask, request, render_template
app = Flask(__name__)
@app.route("/")
def index():
    return render_template('index.html')

@app.route("/add", methods=['POST'])
def add():
    #
    # from POST parameters
    #
    x = int(request.form['x'])
    y = int(request.form['y'])

    #
    # from Kubernetes Service(environment variables)
    #
    my_calc_host = os.environ['MY_CALC_SERVICE_SERVICE_HOST']
    my_calc_port = os.environ['MY_CALC_SERVICE_SERVICE_PORT']

    #
    # remote procedure call to MicroServices(my-calc)
    #
    client = httplib.HTTPConnection(my_calc_host, my_calc_port)
    client.request("GET", "/addition/%d/%d" % (x, y))
    response = client.getresponse()
    result = response.read()

    return render_template('index.html',
        add_x=x, add_y=y, add_result=result)

if __name__ == "__main__":
    app.debug = True
    app.run(host='0.0.0.0')
```

Kubernetes service generates the Kubernetes service name and port number as an environment variable to the other pods. Therefore, the environment variable's name and the Kubernetes service name must be consistent.

In this scenario, the `my-calc` service name must be `my-calc-service`.

2. Frontend WebUI uses the Flask HTML template; it is similar to PHP and JSP such that `entry.py` will pass the parameter to the template (`index.html`) to render the HTML:

```
<html>
<body>
<div>
    <form method="post" action="/add">
        <input type="text" name="x" size="2"/>
        <input type="text" name="y" size="2"/>
        <input type="submit" value="addition"/>
    </form>

    {% if add_result %}
    <p>Answer : {{ add_x }} + {{ add_y }} = {{ add_result }}</p>
    {% endif %}
</div>
</body>
</html>
```

3. `Dockerfile` is exactly the same as microservices. So, eventually, the file structure will be as follows. Note that `index.html` is a template file; therefore, put it under the templates directory:

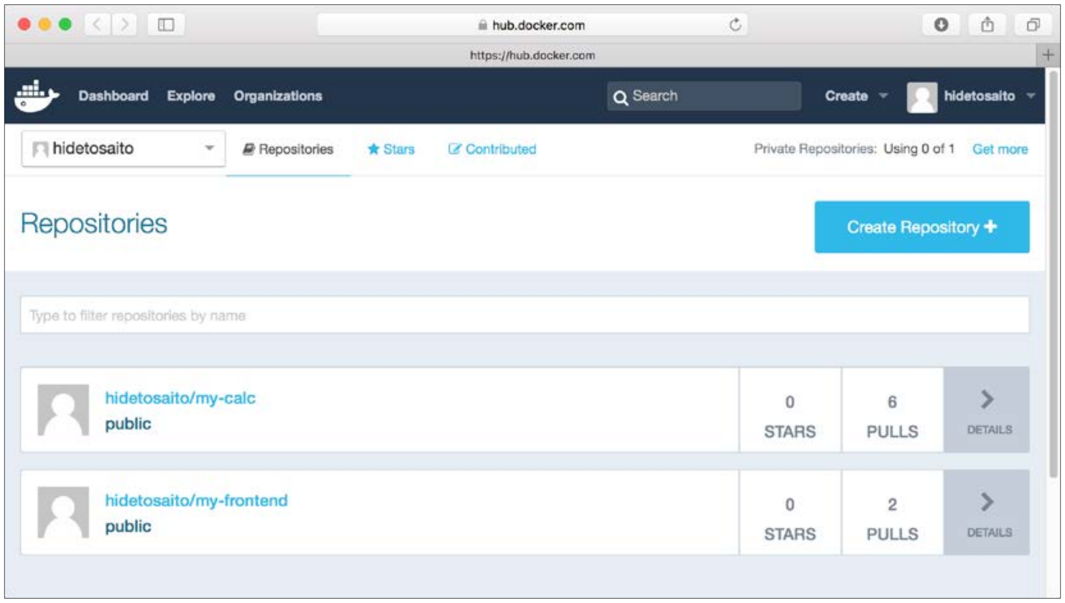
```
/Dockerfile
/entry.py
/templates/index.html
```

4. Then, build a Docker image and push to Docker Hub as follows:

In order to push your image to Docker Hub, you need to log in using the `docker login` command. It is needed only once; the system checks `~/.docker/config.json` to read from there.

```
//build frontend Webui image
$ sudo docker build -t hidetosaito/my-frontend .
```

```
//login to docker hub, if not login yet
$ sudo docker login

//push frontend webui image
$ sudo docker push hidetosaito/my-frontend
```



Repositories

hidetosaito/my-calc public	0 STARS	6 PULLS
hidetosaito/my-frontend public	0 STARS	2 PULLS

Microservices and Frontend WebUI image on Docker Hub

How it works...

Launch both microservices and the Frontend WebUI.

Microservices

Microservices (`my-calc`) uses the Kubernetes replication controller and service, but it needs to communicate to other pods only. In other words, there's no need to expose it to the outside Kubernetes network. Therefore, the service type is set as `ClusterIP`:

```
# cat my-calc.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-calc-rc
spec:
  replicas: 2
  selector:
        app: my-calc
  template:
    metadata:
      labels:
        app: my-calc
    spec:
      containers:
      - name: my-calc
        image: hidetosaito/my-calc
---
apiVersion: v1
kind: Service
metadata:
  name: my-calc-service

spec:
  ports:
    - protocol: TCP
      port: 5000
  type: ClusterIP
  selector:
     app: my-calc
```

Use the `kubectl` command to load the `my-calc` pods as follows:

```
$ sudo kubectl create -f my-calc.yaml
replicationcontroller "my-calc-rc" created
service "my-calc-service" created
```

Frontend WebUI

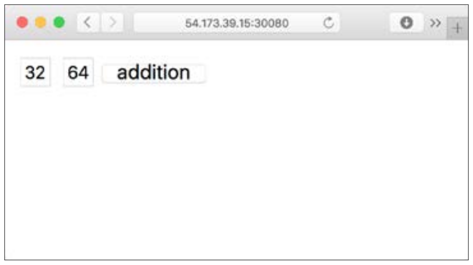
Frontend WebUI also uses the replication controller and service, but it exposes the port (TCP port `30080`) in order to access it from an external web browser:

```
$ cat my-frontend.yaml
apiVersion: v1
kind: ReplicationController
```

```
metadata:
  name: my-frontend-rc
spec:
  replicas: 2
  selector:
        app: my-frontend
  template:
    metadata:
      labels:
        app: my-frontend
    spec:
      containers:
      - name: my-frontend
        image: hidetosaito/my-frontend
---
apiVersion: v1
kind: Service
metadata:
  name: my-frontend-service

spec:
  ports:
    - protocol: TCP
      port: 5000
      nodePort: 30080
  type: NodePort
  selector:
     app: my-frontend

$ sudo kubectl create -f my-frontend.yaml
replicationcontroller "my-frontend-rc" created
service "my-frontend-service" created
```

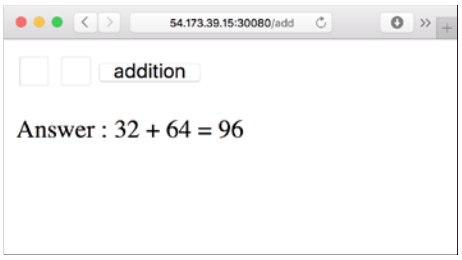
You have exposed your service to an external port on all the nodes in your cluster. If you want to expose this service to the external Internet, you may need to set up firewall rules for the service port(s) (TCP port `30080`) to serve traffic. Refer to `http://releases.k8s.io/release-1.1/docs/user-guide/services-firewalls.md` for more details.

Let's try to access `my-frontend` using a web browser. You can access any Kubernetes node's IP address; specify the port number `30080` as follows:



Access to the Frontend WebUI

When you click on the **addition** button, it will forward a parameter to microservices (`my-calc`). Microservices compute the addition (yes, just an addition!) and then return the result back to the Frontend WebUI as follows:

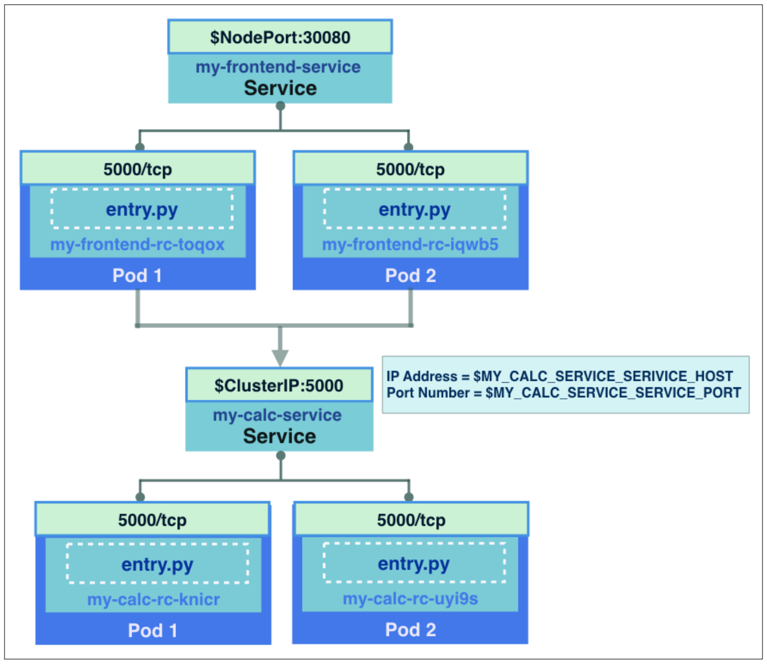


Answer : 32 + 64 = 96

Get a result from microservices and render the HTML

So now, it is easy to adjust the number of replicas for the Frontend WebUI and microservices independently. For example, WebUI replicas range from 2 to 8 and microservice replicas range from 2 to 16.

Also, if there's a need to fix some bugs, for example, there's a frontend need to validate the input parameter to check whether it is numeric or string (yes, if you type string and then submit, it will show an error!); it will not affect the build and deploy the cycle against microservices:



The frontend WebUI and microservices diagram

In addition, if you want to add an additional microservice, for example, subtract microservices, you may need to create another Docker image and deploy with another replication controller and service, so it will be independent from the current microservices.

Then, you can keep accumulate your own microservices ecosystem to re-use for another application.

See also

This recipe described how your application aligns to the microservices architecture. The microservices and the Docker container perfectly fit the concept. The following recipe also helps in managing your microservices' container images:

- *Working with the private Docker registry*

Integrating with Jenkins

In software engineering, **Continuous Integration** (**CI**) (`https://en.wikipedia.org/wiki/Continuous_integration`) and **Continuous Delivery** (**CD**) (`https://en.wikipedia.org/wiki/Continuous_delivery`), abbreviated as CI/CD, have the idea to simplify the procedure of the traditional development process with a continuous testing mechanism in order to reduce the panic of serious confliction, namely, to solve small errors immediately once you have found it. Furthermore, through automatic tools, a product running on the CI/CD system can achieve better efficiency for bug fixes or new feature delivery. Jenkins is one of the well-known CI/CD applications. Projects in Jenkins pull codes from the code base server and test or deploy. In this recipe, we will show you how the Kubernetes system joins Jenkins servers as a CI/CD group.

Getting ready

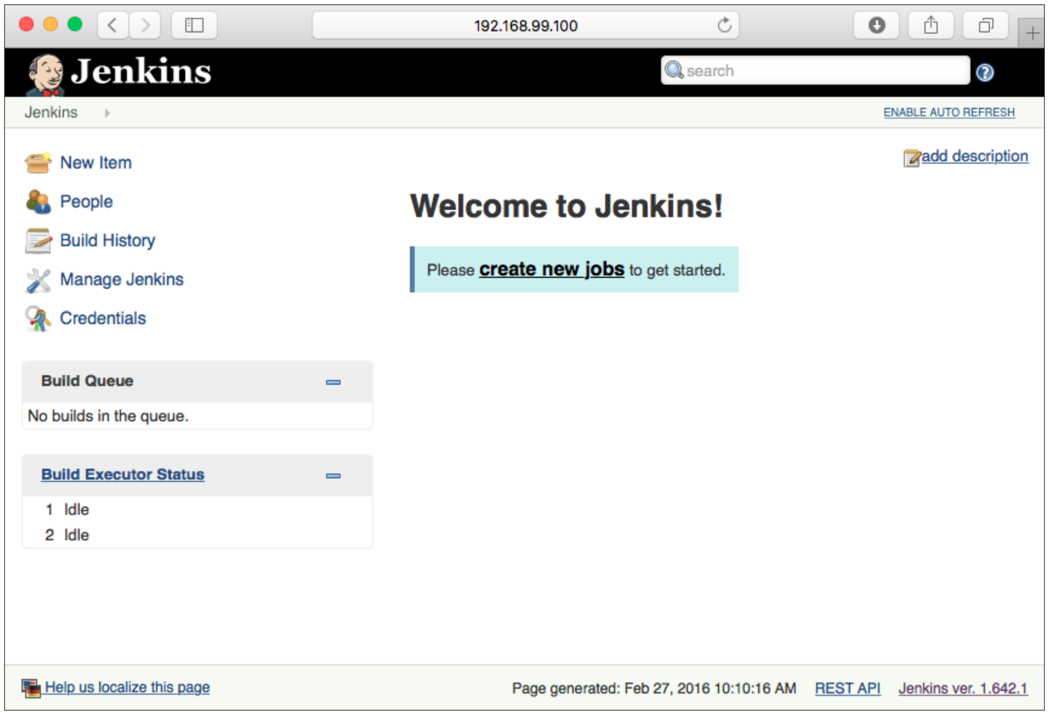
Before you start this recipe, prepare a Docker Hub account (`https://hub.docker.com`) if you don't have your Docker registry; we will put the images built by Jenkins here. It is also where Kubernetes pulls images from. Next, make sure both the Kubernetes servers and your Jenkins servers are ready. You can set up your own standalone Jenkins server through the Docker image as well; the details are indicated here.

Install a Jenkins server which can build a Docker program

First, you need a Docker-installed machine. Then, you can go ahead to create a Jenkins server using this command:

```
// Pull the Jenkins image from Docker Hub
$ docker run -p 8080:8080 -v /your/jenkins_home/:/var/jenkins_home -v
$(which docker):/bin/docker jenkins
```

Port `8080` is the portal of a website. It is recommended to assign a host directory for mounting the Jenkins home directory. Therefore, you can keep the data even with the container shutdown. We will also mount the Docker binary file to our Jenkins container, since this Jenkins official image didn't install the `docker` command. After you push this command, logs for installation will show on your terminal. Once you see this information, **INFO: Jenkins is fully up and running**, it is good for you to check the web console of Jenkins using `DOCKER_MACHINE_IP_ADDRESS:8080`:



After this, you have to install the Git and Docker plugins for Jenkins. Installing plugins is the way to customize your Jenkins server. Here, these two plugins can make you fulfill the workflow from the code in your laptop to the containers on the Kubernetes system. You can refer to the following steps:

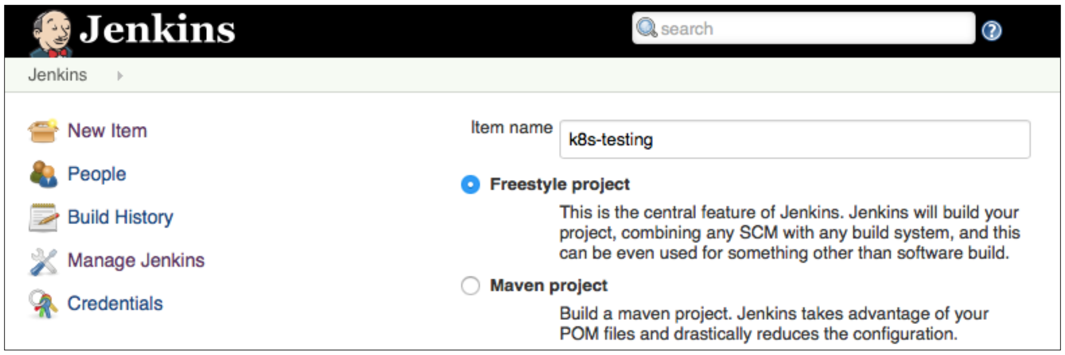
1. Click on **Manage Jenkins** on the left menu.
2. Pick **Manage Plugins**.
3. Choose the tag **Available**.
4. Key in **Git Plugin** and check it; do the same for **CloudBees Docker Build and Publish Plugin**.
5. Start the installation and reboot.

How to do it...

We are going to make Kubernetes the working station for running testing containers or deploying containers officially. Currently, there are no Jenkins plugins for deploying pods on the Kubernetes systems. Therefore, we are going to call the Kubernetes API for our usage. In order to run a program test, we can use the Kubernetes job mentioned in *Ensuring flexible usage of your containers* recipe in *Chapter 3*, *Playing with Containers*, which will create the job-like pod that could be terminated once finished. It is suitable for testing and verification. On the other hand, we can directly create the replication controller and service for official deployment.

Create your Jenkins project

As a program build, we will create a single Jenkins project for each program. Now, you can click on **New Item** on the menu on the left-hand side. In this recipe, choose **Freestyle project**; it is good enough for the building examples later on. Name the project and click on **OK**:



Item name: k8s-testing

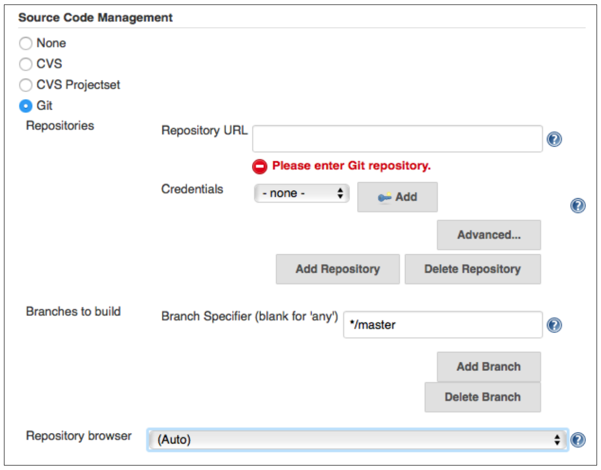
[✓ Freestyle project]
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

Maven project
Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.

Then, you will see the following blocks of categories on the configuration page:

- **Advanced Project Options**
- **Source Code Management**
- **Build Triggers**
- **Build**
- **Post-build Actions**

These settings can make your project more flexible and more close to the one you need. However, we will only focus on the **Source Code Management** and **Build** parts to meet our requirements. **Source Code Management** is where we will define our codebase location. In the later sections, we will choose **Git** and put the corresponding repository:

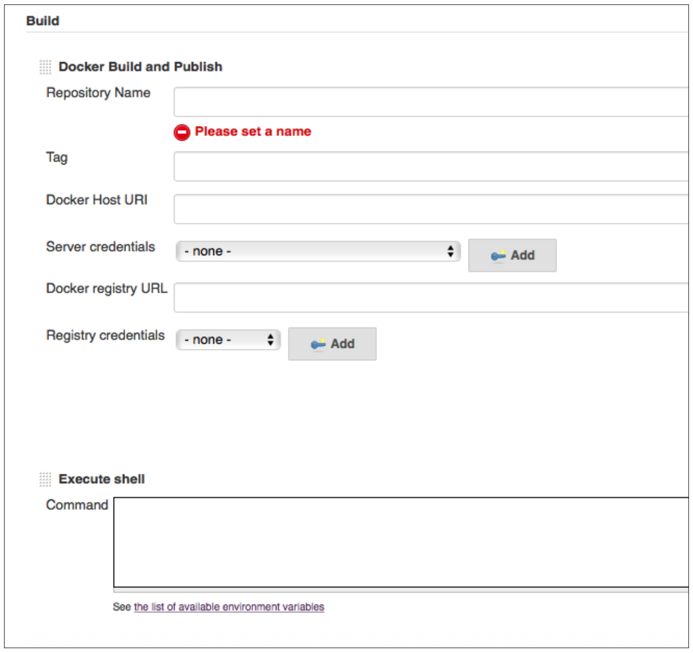


Source Code Management

None [✓ Git] CVS, CVS Projectset

Repositories	Repository URL:
	Please enter Git repository.
	Credentials: - none -
Branches to build	Branch Specifier (blank for 'any'): */master
Repository browser	(Auto)

In the category **Build**, several steps, such as **Docker Build and Publish** and **Execute shell**, will help us build Docker images, push images to the Docker registry, and trigger the Kubernetes system to run the programs:



Build

Docker Build and Publish

Repository Name	
	Please set a name
Tag	
Docker Host URI	
Server credentials	- none -
Docker registry URL	
Registry credentials	- none -

Execute shell

Command

See the list of available environment variables

The following two scenarios will demonstrate how to set the configurations for running your programs on the Kubernetes systems.

Run a program testing

Kubernetes' job is to handle some programs that have terminal conditions, such as unit testing. Here, we have a short dockerized program, which is suitable for this situation. Feel free to check it on GitHub: `https://github.com/kubernetes-cookbook/sleeper`. As indicated in `Dockerfile`, this tiny program will be killed after 10 seconds of running and doing nothing. The following steps will guide you set up the project's configurations:

1. At **Source Code Management**, choose **Git**, put the **HTTPS URL** of your GitHub repository at **Repository URL**. For example, `https://github.com/kubernetes-cookbook/sleeper.git`. And you will find that the **WARNING** message has disappeared for a public repository; or you will have to add a credential.
2. At **Build**, we need the following two build steps:
    1. Add **Docker Build and Publish** first to build your program as an image. A repository name is a required item. In this case, we will use Docker Hub as a registry; please name your repository as `DOCKERHUB_ACCOUNT:YOUR_CUSTOMIZED_NAME`. For example, `nosus:sleeper`. Adding a tag, such as `v$BUILD_NUMBER`, could tag your Docker image with the Jenkins build number. Leave **Docker Host URI** and **Server credentials** empty if your Jenkins servers already installed the Docker package. But if you follow the instructions on the previous page to install the Jenkins server as a container, please check the following tips for detailed settings of these two items. Leave **Docker registry URL** empty, since we used Docker Hub as a Docker registry. However, set a new credential for your Docker Hub accessing permission.
    2. Next, add an **Execute shell** block for calling the Kubernetes API. We put two API calls for our purpose: one is to create a Kubernetes job using the JSON format template and the other is to query whether the job completes successfully or not:

```
#run a k8s job

curl -XPOST -d'{"apiVersion":"extensions/
v1beta1","kind": "Job","metadata":{"name":"sleeper
"}, "spec": {"selector": {"matchLabels": {"image":
"ubuntu","test": "jenkins"}},"template": {"metadata":
{"labels": {"image": "ubuntu","test": "jenkins"}},"spec":
{"containers": [{"name": "sleeper","image": "nosus/
sleeper"}],"restartPolicy": "Never"}}}}' http://YOUR_
KUBERNETES_MASTER_ENDPOINT/apis/extensions/v1beta1/
namespaces/default/jobs

#check status

count=1

returnValue=-1
```

```
while [ $count -lt 60 ]; do
  curl -XGET http://YOUR_KUBERNETES_MASTER_ENDPOINT/apis/
extensions/v1beta1/namespaces/default/jobs/sleeper | grep
"\"succeeded\": 1" && returnValue=0 && break
  sleep 3
count=$(($count+1))
done


return $returnValue
```

We will also add a `while` loop for a 3-minute timeout limitation. Periodically checking the status of the job is a simple way to know whether it is completed. If you fail to get the message **succeeded: 1**, judge the job as a failure.

Set Docker Host URI and Server Credential for a containerized Jenkins server

If you use `docker-machine` to build your Docker environment (for example, an OS X user), type this command in your terminal:

```
$ docker-machine env
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.100:2376"
export DOCKER_CERT_PATH="/Users/YOUR_ACCOUNT/.
docker/machine/machines/default"
export DOCKER_MACHINE_NAME="default"
```

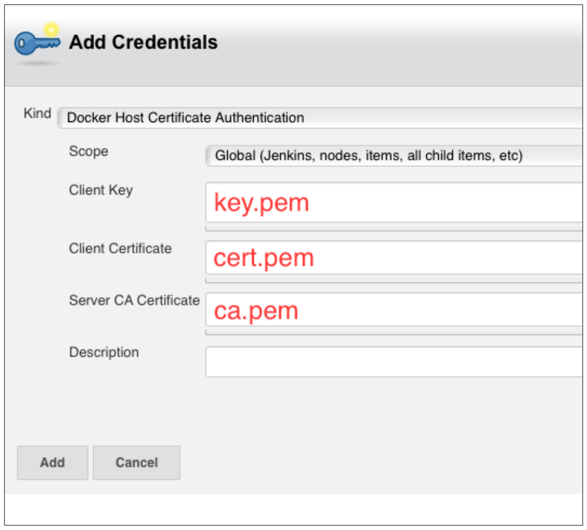
Run this command to configure your shell:

```
# eval $(docker-machine env)
```

By default, you will get the preceding information. Please paste the value of `$DOCKER_HOST` in the item **Docker Host URI**. Then, check your directory at `$DOCKER_CERT_PATH`:

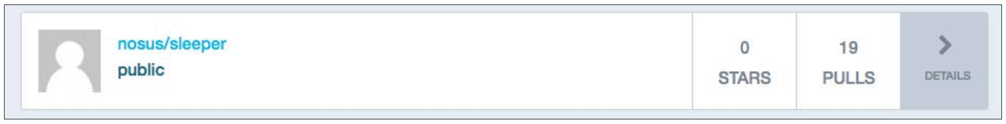
```
$ ls /Users/carol/.docker/machine/machines/default
boot2docker.iso cert.pem        default          id_
rsa             key.pem         server.pem
ca.pem             config.json     disk.vmdk        id_
rsa.pub         server-key.pem
```

Certain files listed here are capable for you to get permission. Please click on the **Add** button beside **Registry credentials** and paste the content of a part of the previous files, as shown in the following screenshot:



Kind: Docker Host Certificate Authentication
Scope: Global (Jenkins, nodes, items, all child items, etc)
Client Key: key.pem
Client Certificate: cert.pem
Server CA Certificate: ca.pem
Description:

After you finish project configurations, you can click on **Save** and then click on **Build Now** to check the result. You may find your image is pushed to Docker Hub as well:



nosus/sleeper
public
0 STARS
19 PULLS

Deploying a program

Deploying a dockerized program through the Jenkins server has similar configurations in the project's settings. We will prepare a simple nginx program for you to try on! Open a new Jenkins project for new settings as follows:

1. At **Source Code Management**, choose **Git**; put the value `https://github.com/kubernetes-cookbook/nginx-demo.git` in **Repository URL**.
2. At **Build**, we will need two build steps as well. They are as follows:
    1. Add a **Docker Build and Publish** first; we are going to put `nosus:nginx-demo` as the repository name. Specify appropriate values at **Docker Host URI** and **Server credentials**.

2. We need an **Execute shell** block for calling the Kubernetes API. There are two API calls put in the space: first one is for creating a pod and the other one is for creating a service to expose the pod:

```
#create a pod
curl -XPOST -d'{"apiVersion": "v1","kind":
"Pod","metadata": {"labels":{"app": "nginx"},"name":
"nginx-demo"},"spec": {"containers": [{"image": "nosus/
nginx-demo","imagePullPolicy": "Always","name":
"nginx-demo","ports": [{"containerPort": 80,"name":
"http"}]}],"restartPolicy": "Always"}}' http:// YOUR_
KUBERNETES_MASTER_ENDPOINT/api/v1/namespaces/default/pods
#create a service
curl -XPOST -d'{"apiVersion": "v1","kind":
"Service","metadata": {"name": "nginx-demo"},"spec":
{"ports": [{"port": 8081,"protocol": "TCP","targetPort":
80}],"selector": {"app": "nginx"},"type": "NodePort"}}'
http://YOUR_KUBERNETES_MASTER_ENDPOINT /api/v1/namespaces/
default/services
```

Feel free to check the endpoint of the Kubernetes service you just created using Jenkins! It is even better if you add a Git server webhook with Jenkins. You can directly get the last result after each code is pushed!

How it works...

The Jenkins server communicates with the Kubernetes system through its RESTful API. You can also take a look at more functionalities via the URL: `http://YOUR_KUBERNETES_MASTER_ENDPOINT:KUBE_API_PORT/swagger_ui`. Hey! The API list is just kept inside your server!

On the other hand, it is possible for you to install the plugin called **HTTP Request Plugin** (`https://wiki.jenkins-ci.org/display/JENKINS/HTTP+Request+Plugin`) to fulfill the Kubernetes API calls. We will not explain this one, since the `POST` function of the current version failed to use the JSON format payload. You can still try the other types of API calls with this plugin.

Currently, there are no plugins that are able to help deploy the Kubernetes tasks. That is why the building procedure is such a headache for the long `curl` commands. It is also an inspiration that you can combine other systems or services with your Kubernetes system through the RESTful API.

There's more...

You may find that in the section *Deploying a program*, if we build the same project again, the Kubernetes API call will return an error response, replying that the pod and service already exist. In this situation, Kubernetes does not have any rolling update API for the live updating of our program.

Still, on the aspect of infrastructure, there are two ideas for Kubernetes integration:

- A Jenkins plugin called **Kubernetes Plugin** (`https://wiki.jenkins-ci.org/display/JENKINS/Kubernetes+Plugin`) helps to build Jenkins slaves dynamically.
- You can try to make your Kubernetes master as a Jenkins slave! As a result, it is wonderful to create pods without complicated API calls. There is no problem to use a rolling update!

In the version 1.2 of Kubernetes, there is a new resource type deployment that controls pods and replica sets, which should be the new solution for managing your pods. Deployment has the following features:

- Desired state and available state to indicate whether the pods are ready to use
- Rolling update for the pods for any modification
- It is capable to roll back to the previous revision of deployment

Resource deployment is in the API version extensions `/v1beta1`. Kubernetes supports its API call for both update and rollback. Please take the following API calls as a reference:

```
// Create a deployment
curl -XPOST  -d "{\"metadata\":{\"name\":\"nginx-123\"},\"spec\":{\"repl
icas\":2,\"template\":{\"metadata\":{\"labels\":{\"app\":\nginx\"}},\"sp
ec\":{\"containers\":[{\"name\":\"nginx-deployment\",\"image\":\"nosus/
nginx-demo:v$BUILD_NUMBER\",\"ports\":[{\"containerPort\": 80}]}]}}}}"
YOUR_KUBERNETES_MASTER_ENDPOINT /v1beta1/namespaces/default/deployments
```

A patch call of API is used for update deployments. Here, we will change the image version. Some details about patch operation can be found at `https://github.com/kubernetes/kubernetes/blob/master/docs/devel/api-conventions.md`:

```
// Update a deployment
curl -H "Content-Type: application/strategic-merge-patch+json" -XPATCH
-d '{"spec":{"template":{"spec":{"containers":[{"name":"nginx-
deployment","image":"nosus/nginx-demo:v1"}]}}}}' YOUR_KUBERNETES_MASTER_
ENDPOINT /apis/extensions/v1beta1/namespaces/default/deployments/nginx-
deployment
// Rollback the deployment
```

```
curl -H "Content-Type: application/json" -XPOST -d '{"name":"nginx-
deployment","rollbackTo":{"revision":0}}' YOUR_KUBERNETES_MASTER_ENDPOINT
/apis/extensions/v1beta1/namespaces/default/deployments/nginx-deployment/
rollback
```

In the last API call, we rollback the deployment to the original version, indicated as version `0`. Just try to manage the new resource type deployment by yourself!

See also

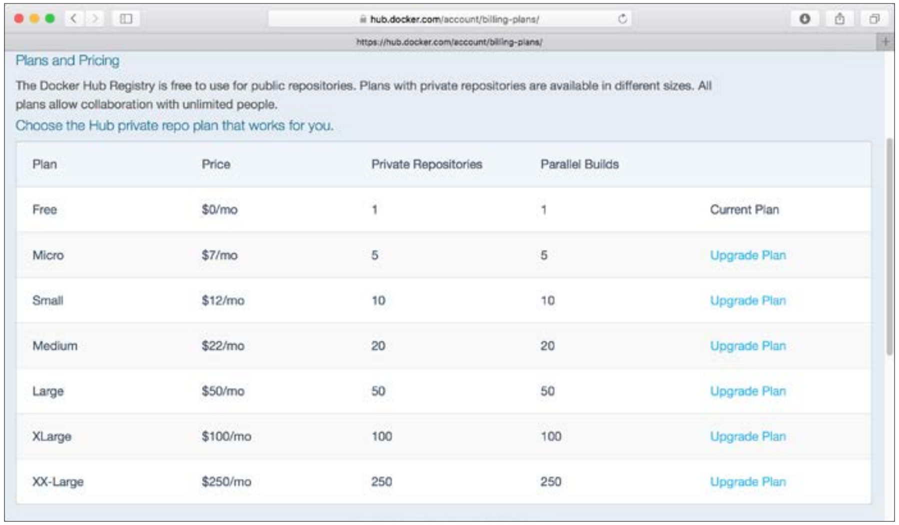
Please continue to read the next recipe for more CI/CD concepts. It will definitely help you while pushing your programs on the Kubernetes system. Also refer to the following recipes:

- *Setting up the Continuous Delivery pipeline*
- The *Working with a RESTful API* and *Authentication and authorization* recipes in *Chapter 7*, *Advanced Cluster Administration*

Working with the private Docker registry

Once you start maintaining your own Docker image, you might need to have some private Docker registry to put some sensitive information into an image or your organization policy.

Docker Hub offers the private repository, which only the authenticated user can push and pull images, and is not visible to other users. However, there is only one quota for a free Docker Hub account. You may pay to increase the private repositories quota, but if you adopt the microservices architecture, you will need a large number of private repositories:



hub.docker.com/account/billing-plans/

Plans and Pricing

The Docker Hub Registry is free to use for public repositories. Plans with private repositories are available in different sizes. All plans allow collaboration with unlimited people.

Choose the Hub private repo plan that works for you.

Plan	Price	Private Repositories	Parallel Builds	
Free	$0/mo	1	1	Current Plan
Micro	$7/mo	5	5	Upgrade Plan
Small	$12/mo	10	10	Upgrade Plan
Medium	$22/mo	20	20	Upgrade Plan
Large	$50/mo	50	50	Upgrade Plan
XLarge	$100/mo	100	100	Upgrade Plan
XX-Large	$250/mo	250	250	Upgrade Plan

Docker Hub private repositories price

There are some ways to set up your own private Docker registry that unlimited Docker image quota locates inside of your network.

Getting ready

The easiest way to set up your Docker registry is use an official Docker registry image (`https://docs.docker.com/registry/`). Run the `docker pull` command to download the Docker registry image as follows:

```
$ docker pull registry:2
2: Pulling from library/registry
f32095d4ba8a: Pull complete
9b607719a62a: Pull complete
973de4038269: Pull complete
2867140211c1: Pull complete
8da16446f5ca: Pull complete
fd8c38b8b68d: Pull complete
136640b01f02: Pull complete
e039ba1c0008: Pull complete
c457c689c328: Pull complete
Digest: sha256:339d702cf9a4b0aa665269cc36255ee7ce424412d56bee9ad8a247afe8
c49ef1
Status: Downloaded newer image for registry:2

//create Docker image datastore under /mnt/docker/images
$ sudo mkdir /mnt/docker/images

//launch registry that expose the 5000/tcp to 8888/tcp on host
$ sudo docker run -p 8888:5000 -v /mnt/docker/images:/var/lib/registry
registry:2
```

It will store the images to `/mnt/docker/images` on the host machine. It is highly recommended to consider to mount a network data volume such as NFS or use Docker volume

How to do it...

Let's create your simple Docker image based on `nginx`:

1. Firstly, prepare `index.html` as follows:

   ```
   $ cat index.html
   <html>
       <head><title>My Image</title></head>

       <body>
           <h1>Hello Docker !</h1>
       </body>

   </html>
   ```

2. Also, prepare `Dockerfile` as follows to build your Docker image:

   ```
   $ cat Dockerfile
   FROM nginx
   COPY index.html /usr/share/nginx/html
   ```

3. Then, build a Docker image name as `<your name>/mynginx` as follows:

   ```
   $ ls
   Dockerfile  index.html

   $ docker build -t hidetosaito/mynginx .
   Sending build context to Docker daemon 3.072 kB
   Step 1 : FROM nginx
    ---> 9737f81306ee
   Step 2 : COPY index.html /usr/share/nginx/html
    ---> 74dd7902a931
   Removing intermediate container eefc2bb17e24
   Successfully built 74dd7902a931
   ```

   At this moment, the `mynginx` image is stored on this host only.

4. Now, it's time to push to your own private registry. First of all, it needs to tag the image name as `<private_registry:port number>/<your name>/mynginx` as follows:

   ```
   $ docker tag hidetosaito/mynginx ip-10-96-219-25:8888/hidetosaito/
   mynginx
   ```

```
$ docker images
REPOSITORY                                       TAG
IMAGE ID             CREATED                VIRTUAL SIZE
ip-10-96-219-25:8888/hidetosaito/mynginx   latest
b69b2ab1f31b        7 minutes ago        134.6 MB
hidetosaito/mynginx                            latest
b69b2ab1f31b        7 minutes ago        134.6 MB
```

> You may see that `IMAGE ID` are the same, because they are the same image.

5. Then, push to the private registry using the `docker push` command as follows:

```
$ docker push ip-10-96-219-25:8888/hidetosaito/mynginx
The push refers to a repository [ip-10-96-219-25:8888/hidetosaito/
mynginx] (len: 1)
b69b2ab1f31b: Pushed
ae8e1e9c54b3: Pushed
18de280c0e54: Pushed
cd0794b5fd94: Pushed
f32095d4ba8a: Pushed
latest: digest: sha256:7ac04fdaedad1cbcc8c92fb2ff099a6509f4f29b0f6
94ae044a0cffc8ffe58b4 size: 15087
```

   Now, your `mynginx` image has been stored in your private registry. Let's deploy this image using Kubernetes.

6. Prepare the YAML file, which contains command to use `nginx` from the private registry and use the Kubernetes service to expose to TCP port `30080`:

```
# cat my-nginx-with-service.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: mynginx
spec:
  replicas: 2
  selector:
        app: mynginx
  template:
        metadata:
          labels:
            app: mynginx
        spec:
          containers:
          - name: mynginx
            image: ip-10-96-219-25:8888/hidetosaito/mynginx
---
apiVersion: v1
kind: Service
metadata:
  name: mynginx-service

spec:
  ports:
    - protocol: TCP
      port: 80
      nodePort: 30080
  type: NodePort
  selector:
     app: mynginx
```

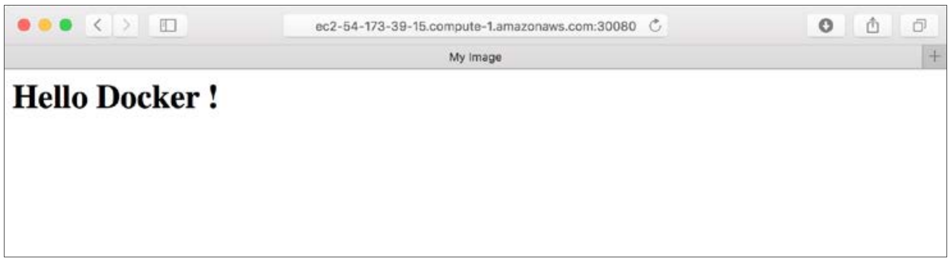
7. Then, use the `kubectl create` command to load this YAML file:

```
# kubectl create -f my-nginx-with-service.yaml
replicationcontroller "mynginx" created
```

You have exposed your service to an external port on all the nodes in your cluster. If you want to expose this service to the external Internet, you may need to set up firewall rules for the service port(s) (TCP port `30080`) to serve traffic. Refer to `http://releases.k8s.io/release-1.1/docs/user-guide/services-firewalls.md` for more details:

```
service "mynginx-service" created
```

Then, access any Kubernetes node on TCP port `30080`; you may see `index.html` as follows:

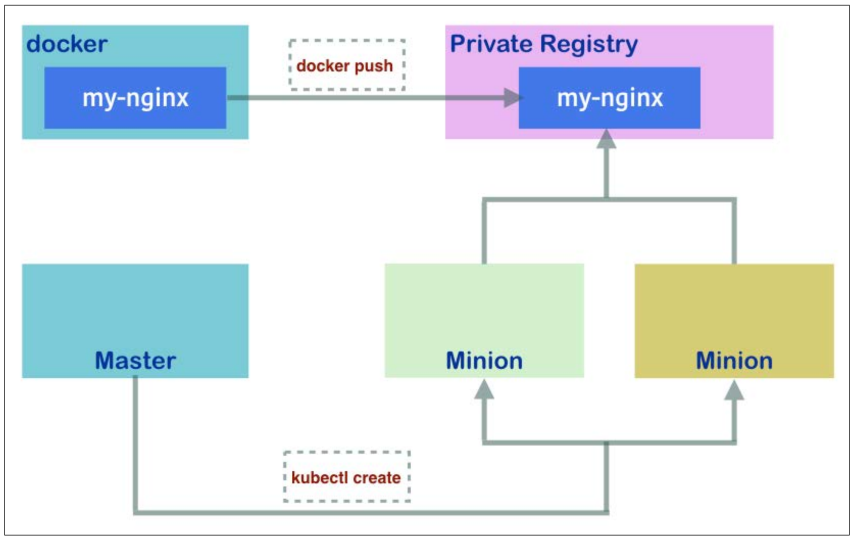


Hello Docker !

How it works...

On running the `docker push` command, it uploads your Docker image to the private registry. Then, when the `kubectl create` command is run, the Kubernetes node performs `docker pull` from the private registry.

Using the private registry is the easiest way to propagate your Docker image to all the Kubernetes nodes:



Alternatives

The official Docker registry image is the standard way to set up your private registry. However, from the management and maintenance points of view, you might need to make more effort. There is an alternative way to construct your own Docker private registry.

Docker Trusted Registry

Docker Trusted Registry is an enterprise version of Docker registry. It comes with Web Console, LDAP integration, and so on. To know more about Docker Trusted Registry, please refer to the following link:

`https://www.docker.com/products/docker-trusted-registry`

Nexus Repository Manager

Nexus Repository Manager is one of the popular repository managers; it supports Java Maven, Linux apt/yum, Docker, and more. You can integrate all the software repository into the Nexus Repository Manager. Read more about Nexus Repository here:

`http://www.sonatype.com/nexus/solution-overview/nexus-repository`

Amazon EC2 Container Registry

Amazon Web Services (**AWS**) also provides a managed Docker registry service. It is integrated with **Identity Access Management** (**IAM**) and the charges are based on storage usage and data transfer usage, instead of the number of images. To know more about it, please refer to following link:

`https://aws.amazon.com/ecr/`

See also

This recipe described how to set up your own Docker registry. The private registry brings you more flexibility and security for your own images. The following recipes help to understand the need for private registry:

- *Moving monolithic to microservices*
- The *Working with volumes* recipe in *Chapter 2*, *Walking through Kubernetes Concepts*

Setting up the Continuous Delivery pipeline

Continuous Delivery is a concept that was first introduced in the book: *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation* written by *Jez Humble* and *David Farley*. By automation of the test, build, and deployment, the pace of software release can be the time to market. It also helps in the collaboration between developers, operations, and testers, reducing communication effort and bugs. CD pipeline aims to be a reliable and repeatable process and tools of delivering software.

Kubernetes is one of the destinations in CD pipeline. This section will describe how to deliver your new release software into Kubernetes by Jenkins and Kubernetes deployment.

Getting ready

Knowing Jenkins is prerequisite to this section. For more details on how to build and setup Jenkins from scratch, please refer to *Integrating with Jenkins* section in this chapter. We will use the sample *Flask* (`http://flask.pocoo.org`) app `my-calc` mentioned in *Moving monolithic to microservices* section. Before setting up our Continuous Delivery pipeline with Kubernetes, we should know what Kubernetes deployment is. Deployment in Kubernetes could create a certain number of pods and replication controller replicas. When a new software is released, you could then roll updates or recreate the pods that are listed in the deployment configuration file, which can ensure your service is always alive.

Just like Jobs, deployment is a part of the extensions API group and still in the `v1beta` version. To enable a deployment resource, set the following command in the API server configuration when launching. If you have already launched the server, just modify the `/etc/kubernetes/apiserver` configuration file and restart the `kube-apiserver` service. Please note that, for now, it still supports the `v1beta1` version:

```
--runtime-config=extensions/v1beta1/deployments=true
```

After the API service starts successfully, we could start building up the service and create the sample `my-calc` app. These steps are required, since the concept of Continuous Delivery is to deliver your software from the source code, build, test and into your desired environment. We have to create the environment first.

Once we have the initial `docker push` command in the Docker registry, let's start creating a deployment named `my-calc-deployment`:

```
# cat my-calc-deployment.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: my-calc-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: my-calc
    spec:
      containers:
      - name: my-calc
        image: msfuko/my-calc:1
        ports:
        - containerPort: 5000

// create deployment resource
# kubectl create -f deployment.yaml
deployment "my-calc-deployment" created
```

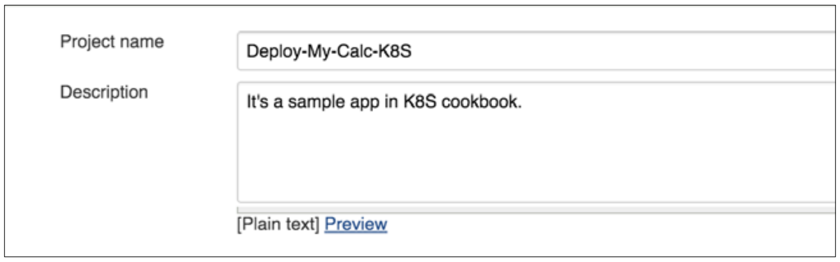
Also, create a service to expose the port to the outside world:

```
# cat deployment-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-calc
spec:
  ports:
    - protocol: TCP
      port: 5000
  type: NodePort
  selector:
     app: my-calc
// create service resource
# kubectl create -f deployment-service.yaml
You have exposed your service on an external port on all nodes in your
cluster. If you want to expose this service to the external internet, you
may need to set up firewall rules for the service port(s) (tcp:31725) to
serve traffic.
service "my-calc" created
```

How to do it...

To set up the Continuous Delivery pipeline, perform the following steps:

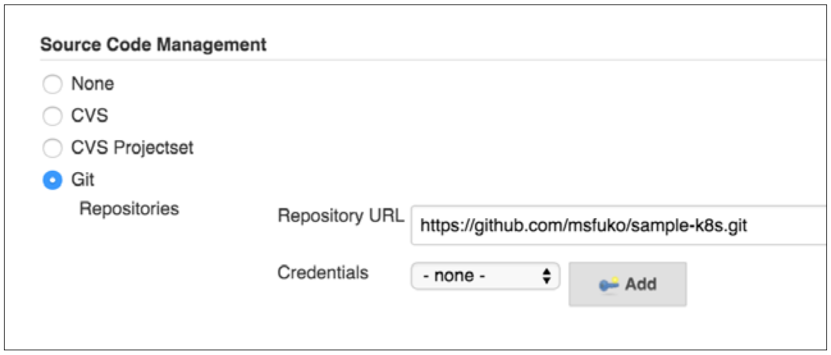
1. At first, we'll start a Jenkins project named `Deploy-My-Calc-K8S` as shown in the following screenshot:



Project name: Deploy-My-Calc-K8S

Description: It's a sample app in K8S cookbook.

2. Then, import the source code information in the **Source Code Management** section:



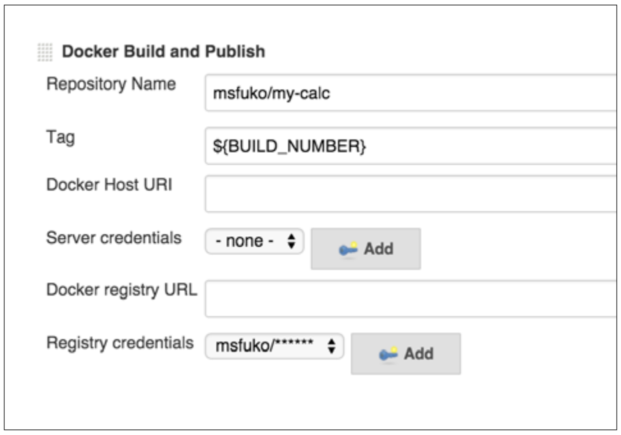
Source Code Management

None [✓ Git] CVS, CVS Projectset

Repositories

Repository URL	https://github.com/msfuko/sample-k8s.git
Credentials	- none -

3. Next, add the targeted Docker registry information into the **Docker Build and Publish** plugin in the **Build** step:



Docker Build and Publish

Repository Name	msfuko/my-calc
Tag	${BUILD_NUMBER}
Docker Host URI	
Server credentials	- none -
Docker registry URL	
Registry credentials	msfuko/******

4. At the end, add the **Execute Shell** section in the **Build** step and set the following command:

```
curl -XPUT -d'{"apiVersion":"extensions/v1beta1","kind":"Depl
oyment","metadata":{"name":"my-calc-deployment"},"spec":{"repl
icas":3,"template":{"metadata":{"labels":{"app":"my-calc"}},"s-
pec":{"containers":[{"name":"my-calc","image":"msfuko/my-
calc:${BUILD_NUMBER}","ports":[{"containerPort":5000}]}]}}}}'
http://54.153.44.46:8080/apis/extensions/v1beta1/namespaces/
default/deployments/my-calc-deployment
```

Let's explain the command here; it's actually the same command with the following configuration file, just using a different format and launching method. One is by using the RESTful API, another one is by using the `kubectl` command.

5. The `${BUILD_NUMBER} tag` is an environment variable in Jenkins, which will export as the current build number of the project:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: my-calc-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: my-calc
    spec:
      containers:
      - name: my-calc
        image: msfuko/my-calc:${BUILD_NUMBER}
        ports:
         containerPort: 5000
```

6. After saving the project and we could start our build. Click on **Build Now**. Then, Jenkins will pull the source code from your Git repository, building and pushing the image. At the end, call the RESTful API of Kubernetes:

```
# showing the log in Jenkins about calling API of Kubernetes
...
[workspace] $ /bin/sh -xe /tmp/hudson3881041045219400676.sh
+ curl -XPUT -d'{"apiVersion":"extensions/v1beta1","kind":"Depl
oyment","metadata":{"name":"my-cal-deployment"},"spec":{"repli
cas":3,"template":{"metadata":{"labels":{"app":"my-cal"}},"spe-
c":{"containers":[{"name":"my-cal","image":"msfuko/my-cal:1","p
orts":[{"containerPort":5000}]}]}}}}' http://54.153.44.46:8080/
apis/extensions/v1beta1/namespaces/default/deployments/my-cal-
deployment
  % Total    % Received % Xferd  Average Speed   Time    Time
Time  Current
                                 Dload  Upload   Total   Spent
Left  Speed
```

```
  0      0     0      0     0      0       0       0 --:--:-- --:--:--
--:--:--     0
100  1670  100  1407  100   263   107k  20534 --:--:-- --:--:--
--:--:--  114k
{
  "kind": "Deployment",
  "apiVersion": "extensions/v1beta1",
  "metadata": {
    "name": "my-calc-deployment",
    "namespace": "default",
    "selfLink": "/apis/extensions/v1beta1/namespaces/default/
deployments/my-calc-deployment",
    "uid": "db49f34e-e41c-11e5-aaa9-061300daf0d1",
    "resourceVersion": "35320",
    "creationTimestamp": "2016-03-07T04:27:09Z",
    "labels": {
      "app": "my-calc"
    }
  },
  "spec": {
    "replicas": 3,
    "selector": {
      "app": "my-calc"
    },
    "template": {
      "metadata": {
        "creationTimestamp": null,
        "labels": {
          "app": "my-calc"
        }
      },
      "spec": {
        "containers": [
          {
            "name": "my-calc",
            "image": "msfuko/my-calc:1",
            "ports": [
              {
                "containerPort": 5000,
                "protocol": "TCP"
              }
            ],
            "resources": {},
            "terminationMessagePath": "/dev/termination-log",
            "imagePullPolicy": "IfNotPresent"
          }
        ],
        "restartPolicy": "Always",
        "terminationGracePeriodSeconds": 30,
        "dnsPolicy": "ClusterFirst"
      }
    },
    "strategy": {
      "type": "RollingUpdate",
      "rollingUpdate": {
        "maxUnavailable": 1,
        "maxSurge": 1
      }
    },
    "uniqueLabelKey": "deployment.kubernetes.io/podTemplateHash"
  },
  "status": {}
}
Finished: SUCCESS
```

7. Let's check it using the `kubectl` command line after a few minutes:

```
// check deployment status
# kubectl get deployments
NAME                UPDATEDREPLICAS   AGE
my-cal-deployment   3/3               40m
```

We can see that there's a deployment named `my-cal-deployment`.

8. Using `kubectl describe`, you could check the details:

```
// check the details of my-cal-deployment
# kubectl describe deployment my-cal-deployment
Name:         my-cal-deployment
Namespace:      default
CreationTimestamp:    Mon, 07 Mar 2016 03:20:52 +0000
Labels:        app=my-cal
Selector:      app=my-cal
Replicas:      3 updated / 3 total
StrategyType:      RollingUpdate
RollingUpdateStrategy:       1 max unavailable, 1 max surge, 0 min
ready seconds
OldReplicationControllers:  <none>
NewReplicationController:  deploymentrc-1448558234 (3/3 replicas
created)
Events:
  FirstSeen  LastSeen  Count  From        SubobjectPath  Reason
Message
  ─────────  ────────  ─────  ────        ─────────────  ──────
───────
  46m    46m    1  {deployment-controller }      ScalingRC  Scaled
up rc deploymentrc-3224387841 to 3
  17m    17m    1  {deployment-controller }      ScalingRC  Scaled
up rc deploymentrc-3085188054 to 3
  9m    9m    1  {deployment-controller }      ScalingRC  Scaled
up rc deploymentrc-1448558234 to 1
  2m    2m    1  {deployment-controller }      ScalingRC  Scaled
up rc deploymentrc-1448558234 to 3
```

We could see one interesting setting named `RollingUpdateStrategy`. We have `1 max unavailable`, `1 max surge`, and `0 min ready seconds`. It means that we could set up our strategy to roll the update. Currently, it's the default setting; at the most, one pod is unavailable during the deployment, one pod could be recreated, and zero seconds to wait for the newly created pod to be ready. How about replication controller? Will it be created properly?

```
// check ReplicationController
# kubectl get rc
CONTROLLER                CONTAINER(S)   IMAGE(S)            SELECTOR
REPLICAS   AGE
deploymentrc-1448558234   my-cal         msfuko/my-cal:1         app=my-
cal,deployment.kubernetes.io/podTemplateHash=1448558234    3          1m
```

We could see previously that we have three replicas in this RC with the name `deploymentrc-${id}`. Let's also check the pod:

```
// check Pods
# kubectl get pods
NAME                          READY     STATUS          RESTARTS   AGE
deploymentrc-1448558234-qn45f   1/1       Running           0          4m
deploymentrc-1448558234-4utub   1/1       Running       0          12m
deploymentrc-1448558234-iz9zp   1/1       Running       0          12m
```

We could find out deployment trigger RC creation, and RC trigger pods creation. Let's check the response from our app `my-calc`:

```
# curl http://54.153.44.46:31725/
Hello World!
```

Assume that we have a newly released application. We'll make **Hello world!** to be **Hello Calculator!**. After pushing the code into GitHub, Jenkins could be either triggered by the SCM webhook, periodically run, or triggered manually:

```
[workspace] $ /bin/sh -xe /tmp/hudson877190504897059013.sh
+ curl -XPUT -d{"apiVersion":"extensions/v1beta1","kind":"Deployment","me
tadata":{"name":"my-calc-deployment"},"spec":{"replicas":3,"template":{"m
etadata":{"labels":{"app":"my-calc"}},"spec":{"containers":[{"name":"my-
calc","image":"msfuko/my-calc:2","ports":[{"containerPort":5000}]}]}}}}
http://54.153.44.46:8080/apis/extensions/v1beta1/namespaces/default/
deployments/my-calc-deployment
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed


  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:-
-     0
100  1695  100  1421  100   274  86879  16752 --:--:-- --:--:-- --:--:--
88812
{
  "kind": "Deployment",
  "apiVersion": "extensions/v1beta1",
  "metadata": {
    "name": "my-calc-deployment",
    "namespace": "default",
    "selfLink": "/apis/extensions/v1beta1/namespaces/default/deployments/
my-calc-deployment",
    "uid": "db49f34e-e41c-11e5-aaa9-061300daf0d1",
    "resourceVersion": "35756",
    "creationTimestamp": "2016-03-07T04:27:09Z",
    "labels": {
      "app": "my-calc"
    }
  },
  "spec": {
    "replicas": 3,
    "selector": {
      "app": "my-calc"
    },
    "template": {
      "metadata": {
        "creationTimestamp": null,
        "labels": {
          "app": "my-calc"
        }
      },
      "spec": {
        "containers": [
          {
            "name": "my-calc",
            "image": "msfuko/my-calc:2",
            "ports": [
              {
                "containerPort": 5000,
                "protocol": "TCP"
              }
            ],
            "resources": {},
            "terminationMessagePath": "/dev/termination-log",
            "imagePullPolicy": "IfNotPresent"
          }
        ],
        "restartPolicy": "Always",
        "terminationGracePeriodSeconds": 30,
        "dnsPolicy": "ClusterFirst"
      }
    },
    "strategy": {
      "type": "RollingUpdate",
      "rollingUpdate": {
        "maxUnavailable": 1,
        "maxSurge": 1
      }
    },
    "uniqueLabelKey": "deployment.kubernetes.io/podTemplateHash"
  },
  "status": {}
}
Finished: SUCCESS
```

How it works...

Let's continue the last action. We built a new image with the `$BUILD_NUMBER` tag and triggered Kubernetes to replace a replication controller by a deployment. Let's observe the behavior of the replication controller:

```
# kubectl get rc
CONTROLLER                 CONTAINER(S)   IMAGE(S)              SELECTOR
REPLICAS   AGE
deploymentrc-1705197507    my-calc        msfuko/my-calc:1      app=my-
calc,deployment.kubernetes.io/podTemplateHash=1705197507   3          13m
deploymentrc-1771388868    my-calc        msfuko/my-calc:2      app=my-
calc,deployment.kubernetes.io/podTemplateHash=1771388868   0          18s
```

We can see deployment create another RC named `deploymentrc-1771388868`, whose pod number is currently 0. Wait a while and let's check it again:

```
# kubectl get rc
CONTROLLER                 CONTAINER(S)   IMAGE(S)              SELECTOR
REPLICAS   AGE
```

```
deploymentrc-1705197507    my-calc         msfuko/my-calc:1        app=my-
calc,deployment.kubernetes.io/podTemplateHash=1705197507   1          15m
deploymentrc-1771388868    my-calc         msfuko/my-calc:2        app=my-
calc,deployment.kubernetes.io/podTemplateHash=1771388868   3          1m
```

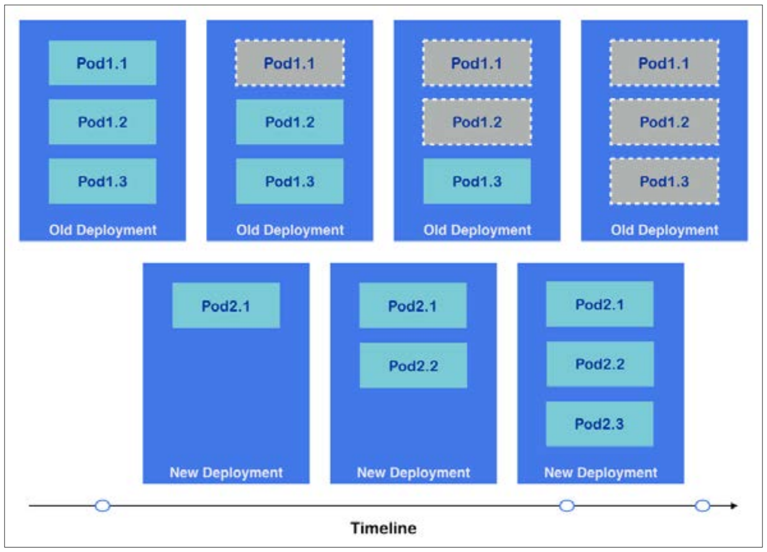
The number of pods in RC with the old image `my-calc:1` reduces to 1 and the new image increase to 3:

```
# kubectl get rc
CONTROLLER                 CONTAINER(S)    IMAGE(S)                SELECTOR
REPLICAS    AGE
deploymentrc-1705197507    my-calc         msfuko/my-calc:1        app=my-
calc,deployment.kubernetes.io/podTemplateHash=1705197507   0          15m
deploymentrc-1771388868    my-calc         msfuko/my-calc:2        app=my-
calc,deployment.kubernetes.io/podTemplateHash=1771388868   3          2m
```

After a few seconds, the old pods are all gone and the new pods replace them to serve users. Let's check the response by the service:

```
# curl http://54.153.44.46:31725/
Hello Calculator!
```

The pods have been rolling updates to the new image individually. Following is the illustration on how it works. Based on `RollingUpdateStrategy`, Kubernetes replaces pods one by one. After the new pod launches successfully, the old pod is destroyed. The bubble in the timeline arrow shows the timing of the logs we got on the previous page. At the end, the new pods will replace all the old pods:



There's more...

Deployment is still in the beta version, while some functions are still under development, for example, deleting a deployment resource and recreating strategy support. However, it gives the chance to Jenkins to make the Continuous Delivery pipeline available. It's pretty easy and makes sure all the services are always online to update. For more details of the RESTful API, please refer to `http://YOUR_KUBERNETES_MASTER_ENDPOINT:KUBE_API_PORT/swagger-ui/#!/v1beta1/listNamespacedDeployment`.

See also

By deployment, we could achieve the goals of rolling the update. However, `kubectl` also provides a `rolling-update` command, which is really useful, too. Check out the following recipes:

- The *Updating live containers* and *Ensuring flexible usage of your containers* recipes in *Chapter 3*, *Playing with Containers*
- *Moving monolithic to microservices*
- *Integrating with Jenkins*
- The *Working with a RESTful API* and *Authentication and authorization* recipes in *Chapter 7*, *Advanced Cluster Administration*

6
Building Kubernetes on AWS

In this chapter, we will cover the following topics:

- Building the Kubernetes infrastructure in AWS
- Managing applications using AWS OpsWorks
- Auto-deploying Kubernetes through Chef recipes
- Using AWS CloudFormation for fast provisioning

Introduction

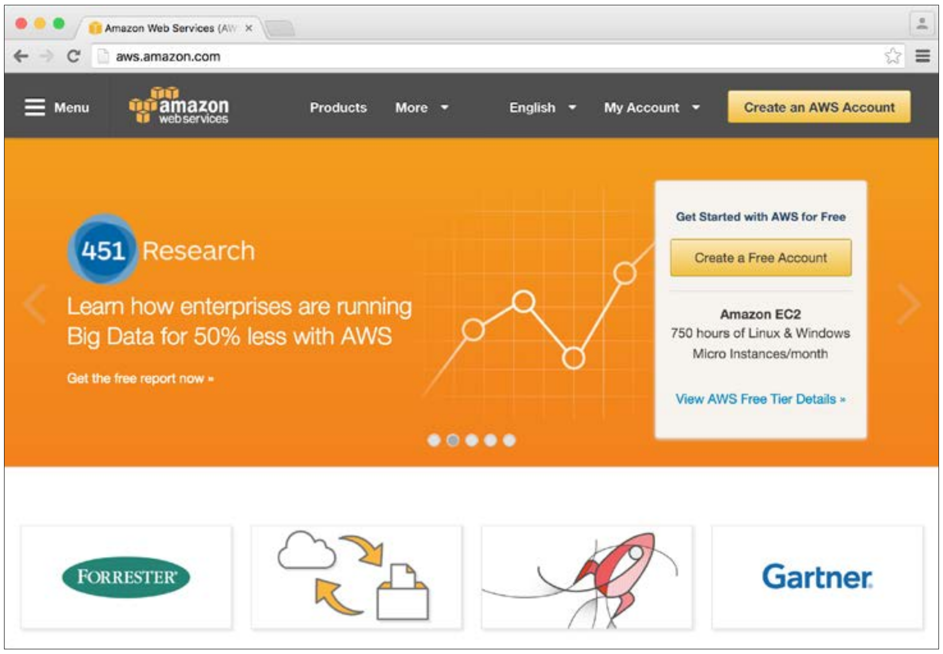
In this chapter, you will learn how to build up all the components on popular public cloud Amazon Web Services. However, we like our infrastructure as code. The infrastructure can be built repeatedly in a reliable way. You will learn how to manage an application's life cycle using AWS OpsWorks, which is powered by Chef. Finally, we will leverage what we learned and build all the infrastructure via a text file in the JSON format!

Building the Kubernetes infrastructure in AWS

Amazon Web Services (AWS) is the most popular cloud service. You can launch several virtual machines on the Amazon datacenter. This section covers sign-up, setting up AWS infrastructure, and launching Kubernetes on AWS.

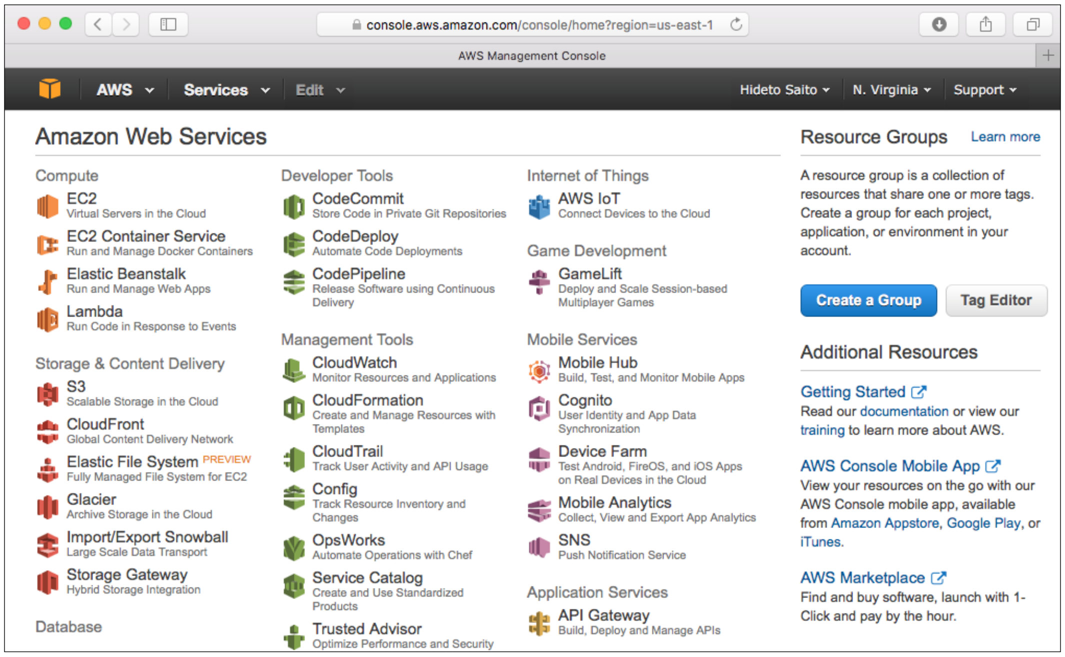
Getting ready

You must sign up to AWS to create an account. Access `http://aws.amazon.com` to put in your information and credit card number:


Amazon Web Services (AW
aws.amazon.com
Menu
amazon web services
Products
More
English
My Account
Create an AWS Account
451 Research
Learn how enterprises are running Big Data for 50% less with AWS
Get the free report now »
Get Started with AWS for Free
Create a Free Account
Amazon EC2
750 hours of Linux & Windows Micro Instances/month
View AWS Free Tier Details »
FORRESTER
Gartner

AWS registration

After registration, you may need to wait up to 24 hours in order to validate your account. After this, you will see the following page after logging on to the AWS console:


console.aws.amazon.com/console/home?region=us-east-1

AWS Management Console

AWS | Services | Edit | Hideto Saito | N. Virginia | Support

Amazon Web Services

Compute
- EC2 — Virtual Servers in the Cloud
- EC2 Container Service — Run and Manage Docker Containers
- Elastic Beanstalk — Run and Manage Web Apps
- Lambda — Run Code in Response to Events

Storage & Content Delivery
- S3 — Scalable Storage in the Cloud
- CloudFront — Global Content Delivery Network
- Elastic File System PREVIEW — Fully Managed File System for EC2
- Glacier — Archive Storage in the Cloud
- Import/Export Snowball — Large Scale Data Transport
- Storage Gateway — Hybrid Storage Integration

Database

Developer Tools
- CodeCommit — Store Code in Private Git Repositories
- CodeDeploy — Automate Code Deployments
- CodePipeline — Release Software using Continuous Delivery

Management Tools
- CloudWatch — Monitor Resources and Applications
- CloudFormation — Create and Manage Resources with Templates
- CloudTrail — Track User Activity and API Usage
- Config — Track Resource Inventory and Changes
- OpsWorks — Automate Operations with Chef
- Service Catalog — Create and Use Standardized Products
- Trusted Advisor — Optimize Performance and Security

Internet of Things
- AWS IoT — Connect Devices to the Cloud

Game Development
- GameLift — Deploy and Scale Session-based Multiplayer Games

Mobile Services
- Mobile Hub — Build, Test, and Monitor Mobile Apps
- Cognito — User Identity and App Data Synchronization
- Device Farm — Test Android, FireOS, and iOS Apps on Real Devices in the Cloud
- Mobile Analytics — Collect, View and Export App Analytics
- SNS — Push Notification Service

Application Services
- API Gateway — Build, Deploy and Manage APIs

Resource Groups — Learn more

A resource group is a collection of resources that share one or more tags. Create a group for each project, application, or environment in your account.

Create a Group | Tag Editor

Additional Resources

Getting Started
Read our documentation or view our training to learn more about AWS.

AWS Console Mobile App
View your resources on the go with our AWS Console mobile app, available from Amazon Appstore, Google Play, or iTunes.

AWS Marketplace
Find and buy software, launch with 1-Click and pay by the hour.

AWS console

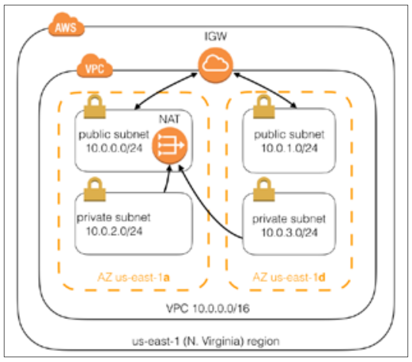
How to do it...

AWS supports multiple region datacenters; you may choose the nearest and cheapest region. Inside the region, there are several **Availability Zones** (**AZ**), which are physically isolated locations that are available.

Once you choose a region, you can set up the **Virtual Private Cloud** (**VPC**) on your own network, such as `10.0.0.0/16`. Inside VPC, you can also define public and private subnets that will do the following:

- Public subnet : Allows you to assign a public IP address and access from/to public Internet via Internet Gateway
- Private subnet : Assigns a private IP address only; can't access from public Internet, outgoing access to Internet through NAT
- Between public subnet and private subnet are accessible

Each subnet must be located in single AZ. Therefore, it would better to create multiple public subnets and multiple private subnets in order to avoid a **single point of failure** (**SPOF**).



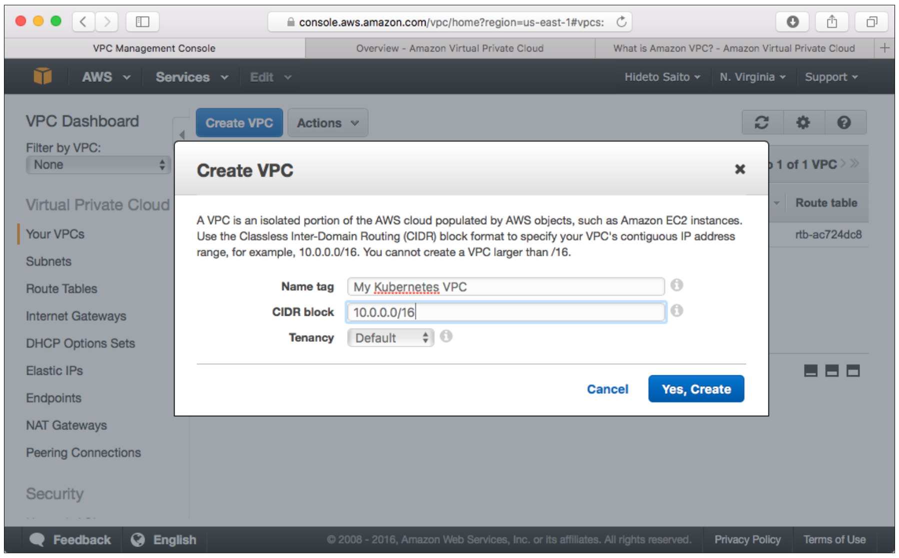
Typical VPC and subnet design

It would better to consider multiAZ for NAT; however, in this cookbook, we will skip it, as it is not necessary. For more details about the NAT gateway, please follow the link `http://docs.aws.amazon.com/AmazonVPC/latest/UserGuide/vpc-nat-gateway.html`.

Let's create this VPC on your AWS.

VPC and subnets

1. On the AWS console, access VPC and click on **Create VPC**. Then, input the name tag as **My Kubernetes VPC** and CIDR as **10.0.0.0/16**:



Create VPC

A VPC is an isolated portion of the AWS cloud populated by AWS objects, such as Amazon EC2 instances. Use the Classless Inter-Domain Routing (CIDR) block format to specify your VPC's contiguous IP address range, for example, 10.0.0.0/16. You cannot create a VPC larger than /16.

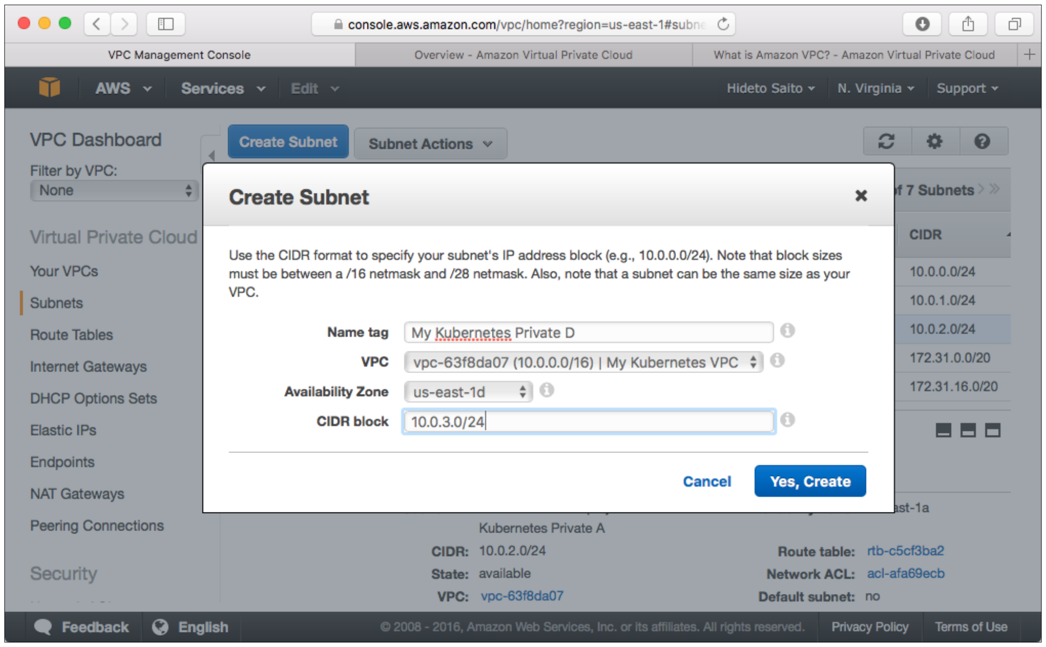
Name tag	My Kubernetes VPC
CIDR block	10.0.0.0/16
Tenancy	Default

Create VPC window

Under the VPC, subnets for both public and private on multiAZ are mentioned in the following table:

Name Tag	CIDR Block	Availability Zone	Auto-Assign Public IP
`My Kubernetes Public A`	`10.0.0.0/24`	`us-east-1a`	Yes
`My Kubernetes Public D`	`10.0.1.0/24`	`us-east-1d`	Yes
`My Kubernetes Private A`	`10.0.2.0/24`	`us-east-1a`	No (Default)
`My Kubernetes Private D`	`10.0.3.0/24`	`us-east-1d`	No (Default)

2. Click on **Subnets** on the left navigation link. Then, click on the **Create Subnet** button. Fill out the information and choose the **VPC** and **Availability Zone** for each subnet. Repeat this four times to create four subnets:



Create Subnet

Use the CIDR format to specify your subnet's IP address block (e.g., 10.0.0.0/24). Note that block sizes must be between a /16 netmask and /28 netmask. Also, note that a subnet can be the same size as your VPC.

Name tag	My Kubernetes Private D
VPC	vpc-63f8da07 (10.0.0.0/16) | My Kubernetes VPC
Availability Zone	us-east-1d
CIDR block	10.0.3.0/24

CIDR
10.0.0.0/24
10.0.1.0/24
10.0.2.0/24
172.31.0.0/20
172.31.16.0/20

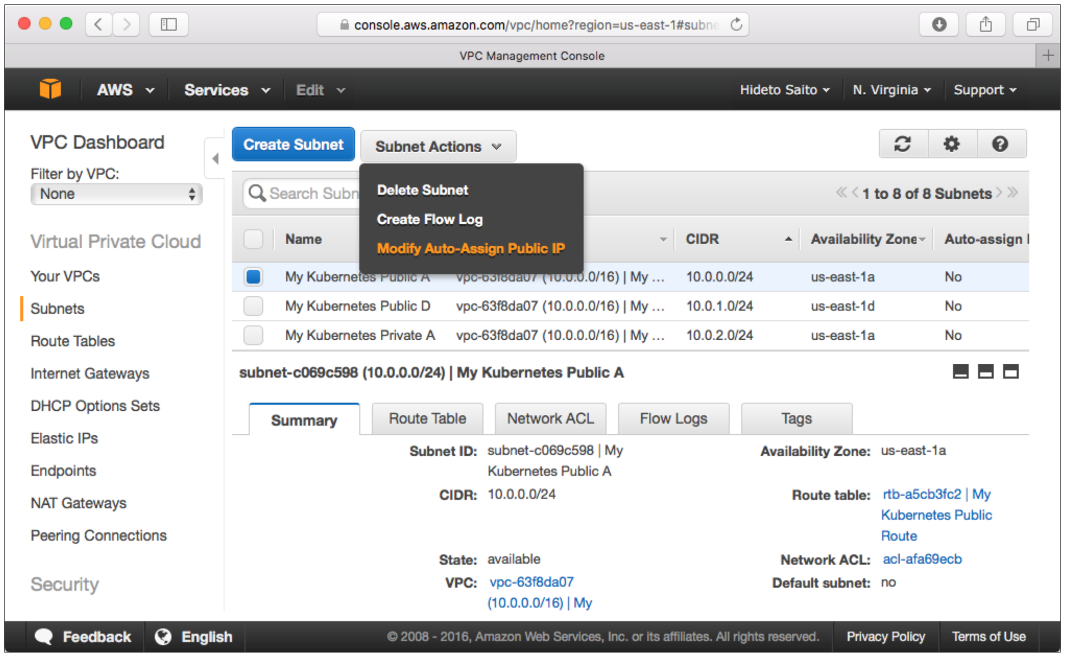
ast-1a

Kubernetes Private A

CIDR:	10.0.2.0/24	Route table:	rtb-c5cf3ba2
State:	available	Network ACL:	acl-afa69ecb
VPC:	vpc-63f8da07	Default subnet:	no

Creating subnet

3. Select the public subnet and click on the **Subnet Actions** button. Then, choose **Modify Auto-Assign Public IP** to enable public IP auto-assignment.



Name		CIDR	Availability Zone	Auto-assign
My Kubernetes Public A	vpc-63f8da07 (10.0.0.0/16) | My ...	10.0.0.0/24	us-east-1a	No
My Kubernetes Public D	vpc-63f8da07 (10.0.0.0/16) | My ...	10.0.1.0/24	us-east-1d	No
My Kubernetes Private A	vpc-63f8da07 (10.0.0.0/16) | My ...	10.0.2.0/24	us-east-1a	No

subnet-c069c598 (10.0.0.0/24) | My Kubernetes Public A

Subnet ID: subnet-c069c598 | My Kubernetes Public A
Availability Zone: us-east-1a
CIDR: 10.0.0.0/24
Route table: rtb-a5cb3fc2 | My Kubernetes Public Route
State: available
Network ACL: acl-afa69ecb
VPC: vpc-63f8da07 (10.0.0.0/16) | My
Default subnet: no

Set Auto-Assign Public IP

Internet Gateway and NAT

Each subnet should have a gateway that goes to an external network. There are two types of gateway, as follows:

- **Internet Gateway** (**IGW**): It allows you to access from/to the Internet (bidirectional) for a server that has a public IP address
- **Network Address Translation** (**NAT**): It allows you to access the Internet (one direction) for a server that has a private IP address

Public subnets associate with an Internet Gateway; on the other hand, private subnets can go to the Internet through NAT. Let's create IGW and NAT as follows:

Type	**Associate to**
Internet Gateway	VPC
NAT Gateway	Public Subnet A

Route Table

1. After creating IGW and NAT, you need to adjust the route table to set the default gateway to IGW or NAT, as follows:

Route Table Name	Route Destination	Association

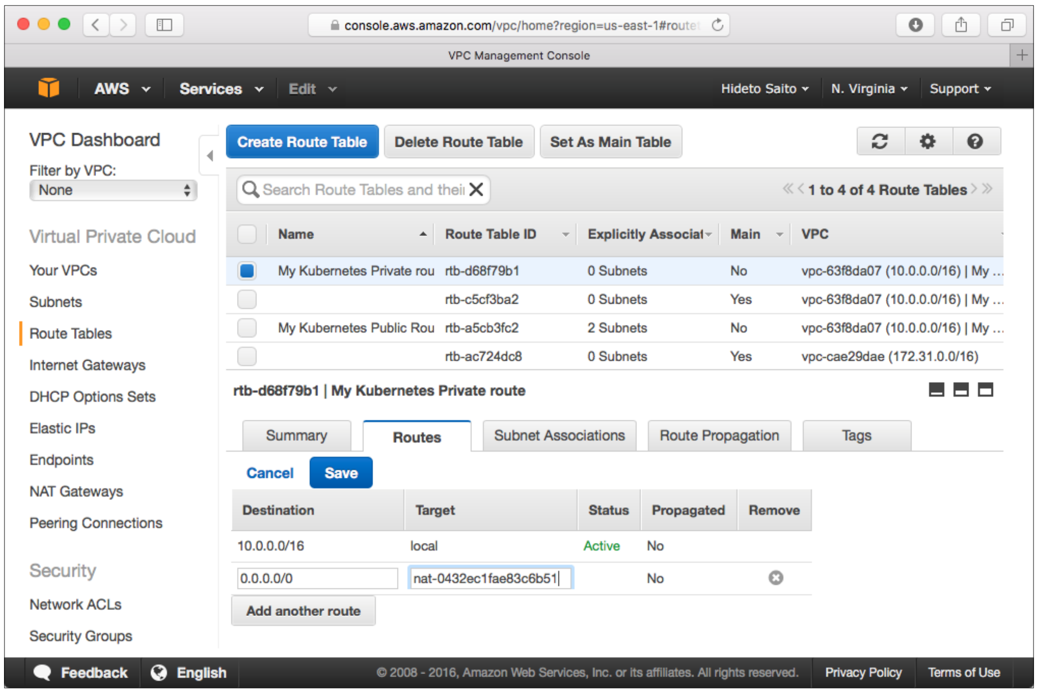
`My Kubernetes Public Route`	`10.0.0.0/16 local`
	`0.0.0.0/0 IGW`
		Public Subnet A
		Public Subnet D

`My Kubernetes Private Route`	`10.0.0.0/16 local`
	`0.0.0.0/0 NAT`
		Private Subnet A
		Private Subnet D

2. On the AWS console, click on **Route Tables** on the left navigation pane. Then, click on the **Create Route Table** button. Fill out **Table Name** and choose the VPN that you created. Repeat this procedure twice for a public route and private route.
3. After creating routes, you need to add the default route as either Internet Gateway (IGW) or NAT.

   For a public route, click on the **Routes** tab and click on **Edit**. Then, add the default route as `0.0.0.0/0` and Target as the IGW ID.

   For a private route, click on the **Routes** tab and click on **Edit**. Then, add the default route as `0.0.0.0/0` and Target as the NAT Gateway ID.



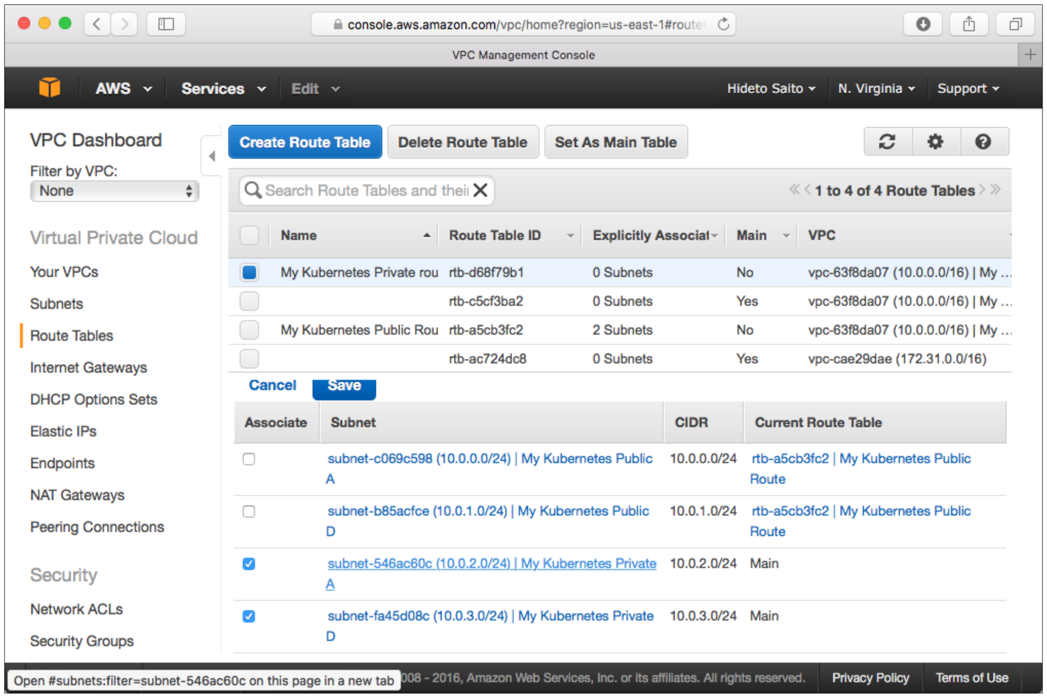
Name	Route Table ID	Explicitly Associat	Main	VPC
My Kubernetes Private rou	rtb-d68f79b1	0 Subnets	No	vpc-63f8da07 (10.0.0.0/16) | My ...
	rtb-c5cf3ba2	0 Subnets	Yes	vpc-63f8da07 (10.0.0.0/16) | My ...
My Kubernetes Public Rou	rtb-a5cb3fc2	2 Subnets	No	vpc-63f8da07 (10.0.0.0/16) | My ...
	rtb-ac724dc8	0 Subnets	Yes	vpc-cae29dae (172.31.0.0/16)

rtb-d68f79b1 | My Kubernetes Private route

Destination	Target	Status	Propagated	Remove
10.0.0.0/16	local	Active	No	
0.0.0.0/0	nat-0432ec1fae83c6b51		No	⊗

Set default route to NAT

4. Finally, click on the **Subnet Associations** tab and then on the **Edit** button. Then, choose public subnets for a public route and private subnets for a private route, as follows:



	Name	Route Table ID	Explicitly Associat	Main	VPC
☑	My Kubernetes Private rou	rtb-d68f79b1	0 Subnets	No	vpc-63f8da07 (10.0.0.0/16) | My ...
☐		rtb-c5cf3ba2	0 Subnets	Yes	vpc-63f8da07 (10.0.0.0/16) | My ...
☐	My Kubernetes Public Rou	rtb-a5cb3fc2	2 Subnets	No	vpc-63f8da07 (10.0.0.0/16) | My ...
☐		rtb-ac724dc8	0 Subnets	Yes	vpc-cae29dae (172.31.0.0/16)

Associate	Subnet	CIDR	Current Route Table
☐	subnet-c069c598 (10.0.0.0/24) | My Kubernetes Public A	10.0.0.0/24	rtb-a5cb3fc2 | My Kubernetes Public Route
☐	subnet-b85acfce (10.0.1.0/24) | My Kubernetes Public D	10.0.1.0/24	rtb-a5cb3fc2 | My Kubernetes Public Route
☑	subnet-546ac60c (10.0.2.0/24) | My Kubernetes Private A	10.0.2.0/24	Main
☑	subnet-fa45d08c (10.0.3.0/24) | My Kubernetes Private D	10.0.3.0/24	Main

Open #subnets:filter=subnet-546ac60c on this page in a new tab 008 - 2016, Amazon Web Services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use

Associate route table to subnet

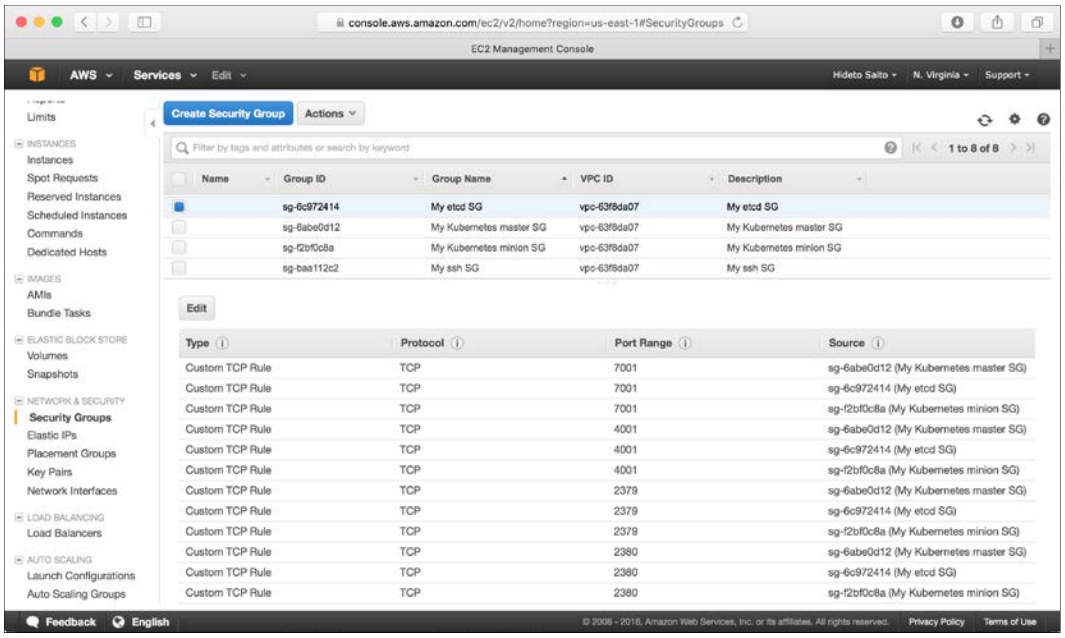
Security group

Security group is a kind of firewall to set up a rule that allows either inbound traffic or outbound traffic. For Kubernetes, there are some known traffic rules that should be set as follows:

Rule name	Inbound Protocol and port number	Source
`My Kubernetes master SG`	▸ `8080/tcp`	▸ `My Kubernetes node`
`My Kubernetes node SG`	▸ `30000-32767/tcp (Service)`	▸ `0.0.0.0/0`

Rule name	**Inbound Protocol and port number**	**Source**
`My etcd SG`	▸ `7001/tcp` ▸ `4001/tcp` ▸ `2379/tcp` ▸ `2380/tcp`	▸ `My etcd SG` ▸ `My Kubernetes master SG` ▸ `My Kubernetes node SG`
`My flannel SG`	▸ `8285/udp` ▸ `8472/udp`	▸ `My flannel SG`
`My ssh SG`	▸ `22/tcp`	▸ `0.0.0.0/0`

On the AWS console, click on **Security Groups** on the left navigation pane, create five **Security Groups**, and add Inbound Rules, as follows:



Name	Group ID	Group Name	VPC ID	Description
	sg-6c972414	My etcd SG	vpc-63f8da07	My etcd SG
	sg-6abe0d12	My Kubernetes master SG	vpc-63f8da07	My Kubernetes master SG
	sg-f2bf0c8a	My Kubernetes minion SG	vpc-63f8da07	My Kubernetes minion SG
	sg-baa112c2	My ssh SG	vpc-63f8da07	My ssh SG

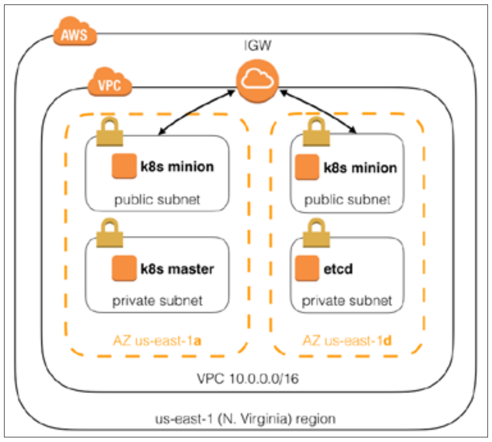
Type	Protocol	Port Range	Source
Custom TCP Rule	TCP	7001	sg-6abe0d12 (My Kubernetes master SG)
Custom TCP Rule	TCP	7001	sg-6c972414 (My etcd SG)
Custom TCP Rule	TCP	7001	sg-f2bf0c8a (My Kubernetes minion SG)
Custom TCP Rule	TCP	4001	sg-6abe0d12 (My Kubernetes master SG)
Custom TCP Rule	TCP	4001	sg-6c972414 (My etcd SG)
Custom TCP Rule	TCP	4001	sg-f2bf0c8a (My Kubernetes minion SG)
Custom TCP Rule	TCP	2379	sg-6abe0d12 (My Kubernetes master SG)
Custom TCP Rule	TCP	2379	sg-6c972414 (My etcd SG)
Custom TCP Rule	TCP	2379	sg-f2bf0c8a (My Kubernetes minion SG)
Custom TCP Rule	TCP	2380	sg-6abe0d12 (My Kubernetes master SG)
Custom TCP Rule	TCP	2380	sg-6c972414 (My etcd SG)
Custom TCP Rule	TCP	2380	sg-f2bf0c8a (My Kubernetes minion SG)

Creating a Security Group

How it works...

Once you create your own VPC and related subnets and security groups, you can launch the EC2 instance to set up your own Kubernetes cluster. Note that the EC2 instances should be launched and associated with subnets and security groups as follows:

Instance	Subnet	Security Group
etcd	Private	▸ `My etcd SG` ▸ `My ssh SG`
Kubernetes node (with flannel)	Public	▸ `My flannel SG` ▸ `My Kubernetes node SG` ▸ `My ssh SG`
Kubernetes master (with flannel)	Private	▸ `My flannel SG` ▸ `My Kubernetes master SG` ▸ `My ssh SG`



Minimal configuration of the Kubernetes cluster in the AWS VPC

See also

In this recipe, you learned how to register Amazon Web Services and how to create your own infrastructure. AWS provides a huge number of services; however, it has a lot of documentation and related webinars online. It is recommended that you read and watch to understand the best practices to use the AWS infrastructure. Following is a list of good resources:

- `https://www.youtube.com/user/AmazonWebServices`
- `https://aws.amazon.com/blogs/aws/`
- `http://www.slideshare.net/AmazonWebServices`

Furthermore, look at the following recipes:

- The *Building datastore* and *Creating an overlay network* recipes in *Chapter 1*, *Building Your Own Kubernetes*
- *Managing applications using AWS OpsWorks*
- *Auto-deploying Kubernetes through Chef recipes*
- *Using AWS CloudFormation for fast provisioning*

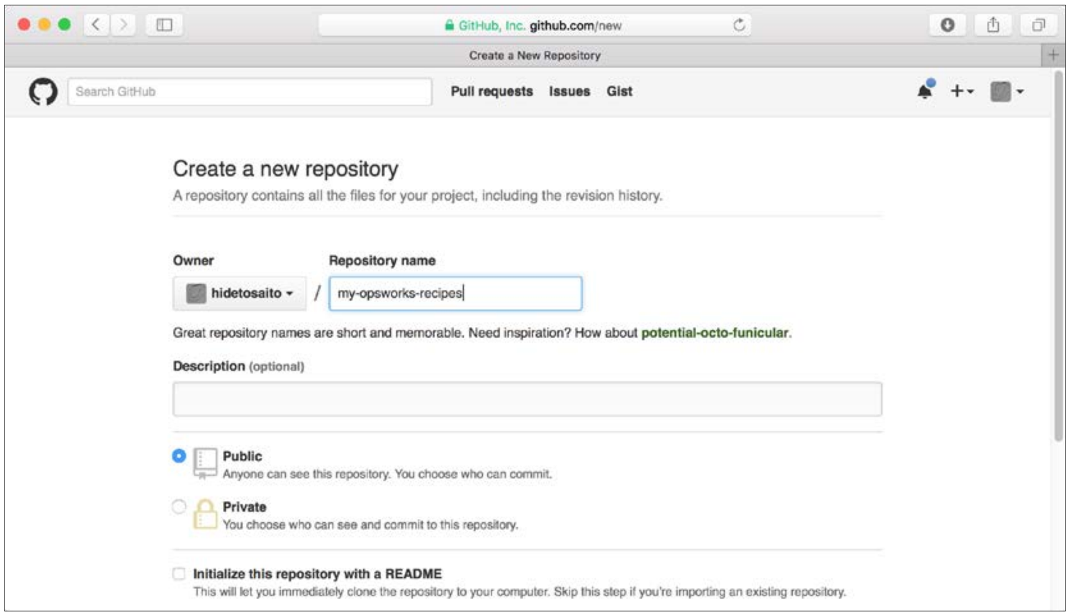
Managing applications using AWS OpsWorks

AWS OpsWorks is a comprehensive AWS EC2 and application deployment framework, which is based on Chef (`http://chef.io/`). It is easy to associate the Chef recipe and EC2 instance using the OpsWorks UI.

Getting ready

In order to upload and maintain your Chef recipes, it is recommended that you prepare a GitHub (`http://github.com`) account. After creating an account on GitHub, create a repository `my-opsworks-recipes` for OpsWorks recipes.

Just so you know, a free user on GitHub can only create a public repository.



Create a new repository

A repository contains all the files for your project, including the revision history.

Owner: hidetosaito / Repository name: my-opsworks-recipes

Great repository names are short and memorable. Need inspiration? How about potential-octo-funicular.

Description (optional)

Public — Anyone can see this repository. You choose who can commit.

Private — You choose who can see and commit to this repository.

Initialize this repository with a README — This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Creating Git repository

After creating the `my-opsworks-recipes` repository, you can access the Git repository via `http://github.com/<your username>/my-opsworks-recipes.git`.

Let's use the AWS CloudWatchLogs as a sample deployment to put a recipe into your repository, as follows:

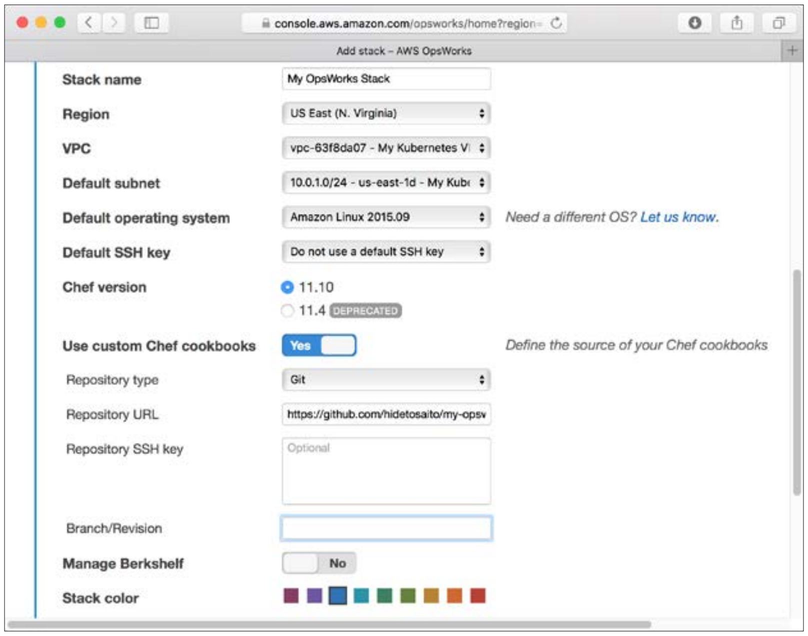
```
//Download CloudWatchLogs Cookbooks
$ curl -L -O https://s3.amazonaws.com/aws-cloudwatch/downloads/
CloudWatchLogs-Cookbooks.zip
//unzip
$ unzip CloudWatchLogs-Cookbooks.zip
//clone your GitHub repository
$ git clone https://github.com/hidetosaito/my-opsworks-recipes.git
//copy CloudWatchLogs Cookbooks into your Git repository
$ mv CloudWatchLogs-Cookbooks/logs my-opsworks-recipes/
$ cd my-opsworks-recipes/
//add recipes to Git
$ git add logs
$ git commit -a -m "initial import"
[master (root-commit) 1d9c16d] initial import
 5 files changed, 59 insertions(+)
 create mode 100755 logs/attributes/default.rb
 create mode 100644 logs/metadata.rb
 create mode 100755 logs/recipes/config.rb
 create mode 100755 logs/recipes/install.rb
 create mode 100755 logs/templates/default/cwlogs.cfg.erb
//push to GitHub.com
$ git push
Username for 'https://github.com': hidetosaito
Password for 'https://hidetosaito@github.com':
Counting objects: 12, done.
```

How to do it...

Access the AWS Web console and navigate to OpsWorks. Then, create an OpsWorks stack first and an OpsWorks layer.

The OpsWorks stack

The OpsWorks stack is the container of the OpsWorks framework. The OpsWorks stack can associate one VPC. You can use your own VPC or the default VPC. Note that, due to compatibility reasons, choose Chef `11.10` to use the `CloudWatchLogs` recipe and don't forget to enable the custom Chef cookbooks and specify your GitHub repository, as follows:



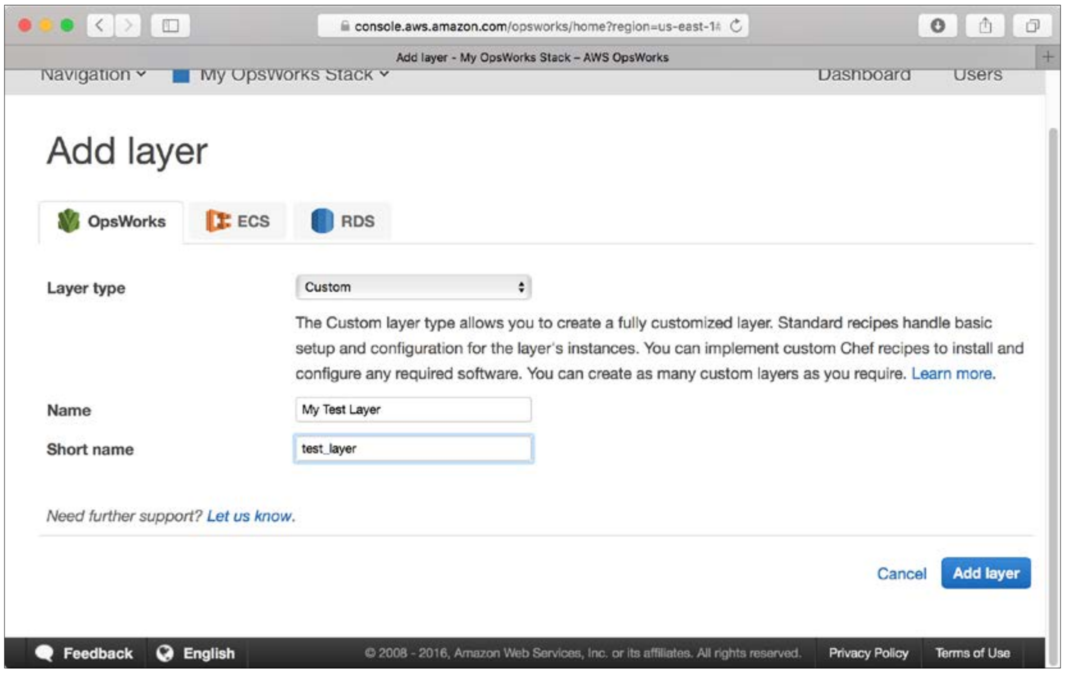
Stack name	My OpsWorks Stack	
Region	US East (N. Virginia)	
VPC	vpc-63f8da07 - My Kubernetes V	
Default subnet	10.0.1.0/24 - us-east-1d - My Kub	
Default operating system	Amazon Linux 2015.09	Need a different OS? Let us know.
Default SSH key	Do not use a default SSH key	
Chef version	● 11.10 ○ 11.4 DEPRECATED	
Use custom Chef cookbooks	Yes	Define the source of your Chef cookbooks
Repository type	Git	
Repository URL	https://github.com/hidetosaito/my-opsv	
Repository SSH key	Optional	
Branch/Revision		
Manage Berkshelf	No	
Stack color	🟪 🟪 🟦 🟦 🟩 🟩 🟨 🟧 🟥	

Creating the OpsWorks stack

The OpsWorks layer

Inside of the OpsWorks stack, you can have one or many OpsWorks layers. One layer can associate one or many Chef recipes. Let's create one custom layer and associate with the `CloudWatchLogs` recipe:

1. Access OpsWorks UI to create a custom layer and put the layer name as follows:


console.aws.amazon.com/opsworks/home?region=us-east-1
Add layer - My OpsWorks Stack – AWS OpsWorks
Navigation
My OpsWorks Stack
Dashboard
Users
Add layer
OpsWorks
ECS
RDS
Cancel
Add layer
Feedback
English
© 2008 - 2016, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

Layer type: Custom

The Custom layer type allows you to create a fully customized layer. Standard recipes handle basic setup and configuration for the layer's instances. You can implement custom Chef recipes to install and configure any required software. You can create as many custom layers as you require. Learn more.

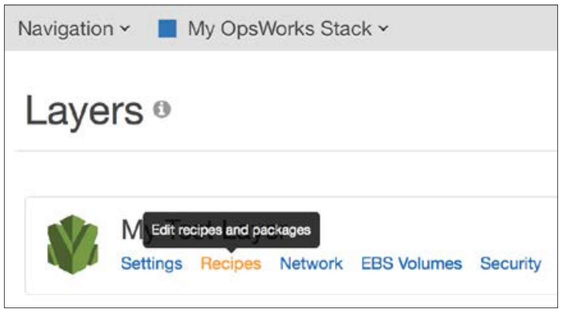
Name: My Test Layer

Short name: test_layer

Need further support? Let us know.

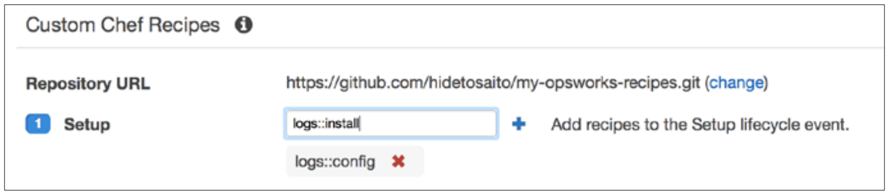
Creating the OpsWorks layer

2. Then, open the recipe page, as shown in the following screenshot:



Recipes settings page

3. Then, add the `logs::config` and `logs::install` recipes to set up a lifecycle event:



Custom Chef Recipes

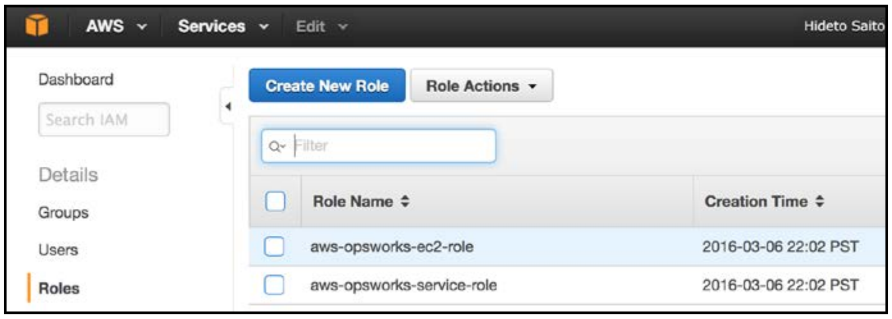
Repository URL	https://github.com/hidetosaito/my-opsworks-recipes.git (change)
1 Setup	logs::install — Add recipes to the Setup lifecycle event.
	logs::config

Associate recipes

Adjusting the IAM role

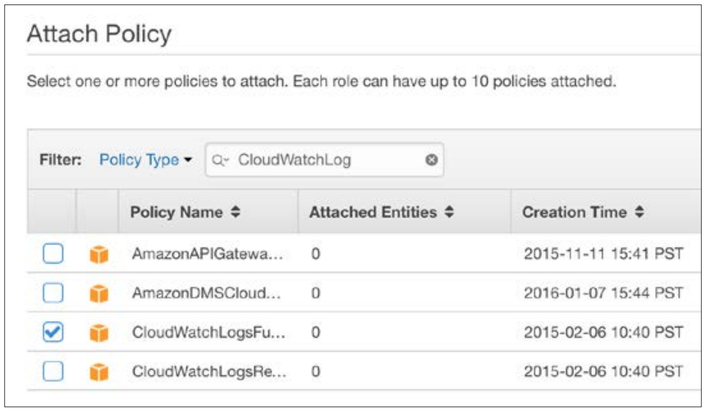
In order to send a log to the `CloudWatchLogs` service, you need to grant permission to the IAM role. Access the AWS IAM console and choose `aws-opsworks-ec2-role`.

The `aws-opsworks-ec2-role` role is created with the OpsWorks stack by default. If you use another role, you would need to change the OpsWorks stack setting to choose your role.



Role Name	Creation Time
aws-opsworks-ec2-role	2016-03-06 22:02 PST
aws-opsworks-service-role	2016-03-06 22:02 PST

Then, attach the `CloudWatchLogsFullAccess` policy, as shown in the following screenshot:



Attach Policy

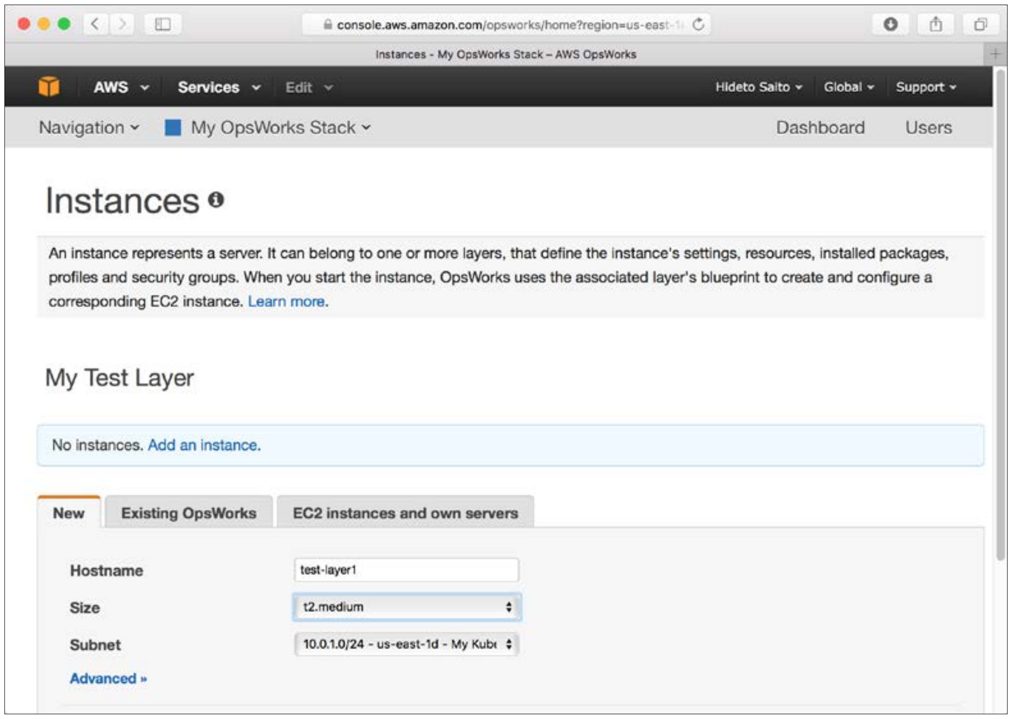
Select one or more policies to attach. Each role can have up to 10 policies attached.

		Policy Name	Attached Entities	Creation Time
☐		AmazonAPIGatewa...	0	2015-11-11 15:41 PST
☐		AmazonDMSCloud...	0	2016-01-07 15:44 PST
✓		CloudWatchLogsFu...	0	2015-02-06 10:40 PST
☐		CloudWatchLogsRe...	0	2015-02-06 10:40 PST

This role will be used when you launch the OpsWorks instance, which uses the `CloudWatchLogs` recipe.

The OpsWorks instance

The OpsWorks instance is a managed EC2 instance, which is associated with the OpsWorks layer. It will automatically set up the Chef Environment and cook your recipe. Let's access the AWS OpsWorks console again and choose the OpsWorks layer to launch an instance, as follows:


console.aws.amazon.com/opsworks/home?region=us-east-1
Instances - My OpsWorks Stack – AWS OpsWorks
AWS
Services
Edit
Hideto Saito
Global
Support
Navigation
My OpsWorks Stack
Dashboard
Users

Instances

An instance represents a server. It can belong to one or more layers, that define the instance's settings, resources, installed packages, profiles and security groups. When you start the instance, OpsWorks uses the associated layer's blueprint to create and configure a corresponding EC2 instance. Learn more.

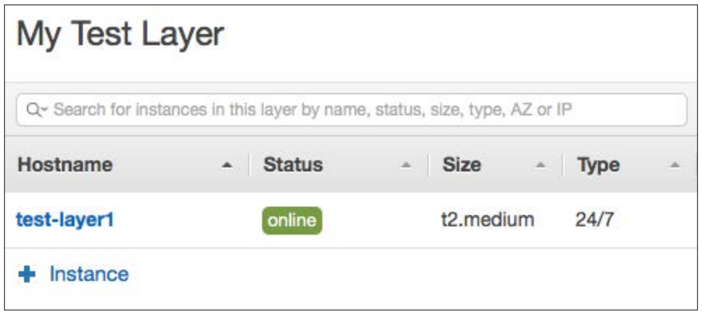
My Test Layer

No instances. Add an instance.

New
Existing OpsWorks
EC2 instances and own servers

Hostname	test-layer1
Size	t2.medium
Subnet	10.0.1.0/24 - us-east-1d - My Kube

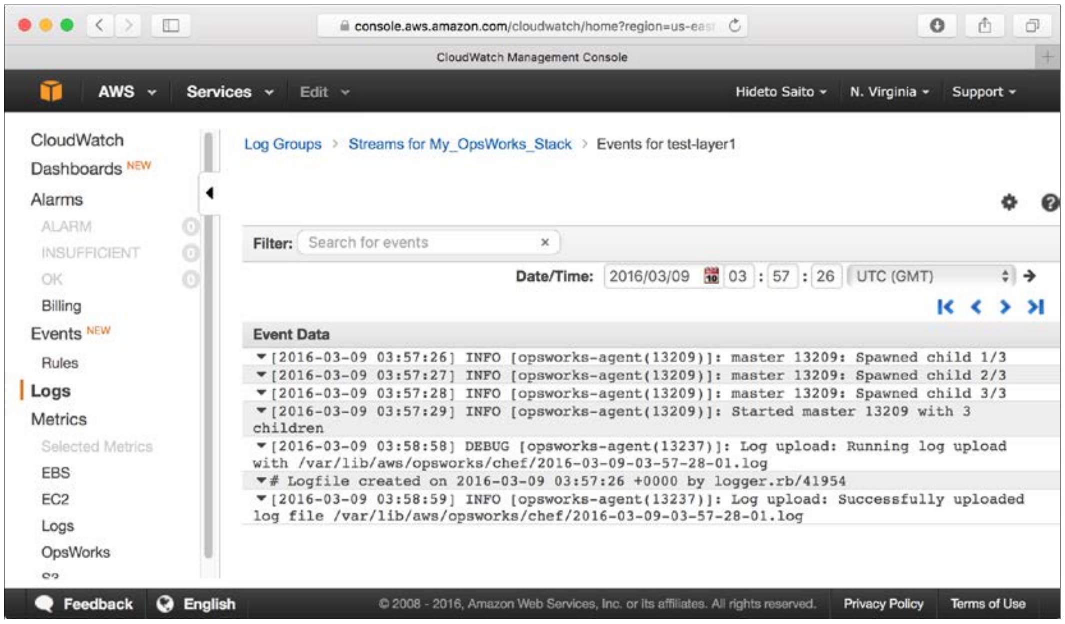
After a few minutes, your instance state will be **online** which means, the launch of an EC2 instance and installation of the `CloudWatchLogs` agent has been completed:


My Test Layer

Search for instances in this layer by name, status, size, type, AZ or IP

Hostname	Status	Size	Type
test-layer1	online	t2.medium	24/7

Now, you can see some logs on the AWS CloudWatchLogs console, as shown in the following screenshot:


console.aws.amazon.com/cloudwatch/home?region=us-east
CloudWatch Management Console
AWS
Services
Edit
Hideto Saito
N. Virginia
Support
CloudWatch
Dashboards NEW
Alarms
ALARM
INSUFFICIENT
OK
Billing
Events NEW
Rules
Logs
Metrics
Selected Metrics
EBS
EC2
Logs
OpsWorks
Log Groups > Streams for My_OpsWorks_Stack > Events for test-layer1
Filter: Search for events
Date/Time: 2016/03/09 03 : 57 : 26 UTC (GMT)
Event Data
Feedback
English
© 2008 - 2016, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

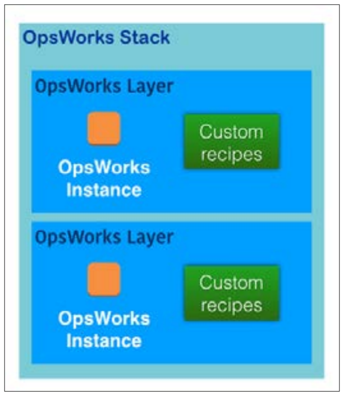
```
[2016-03-09 03:57:26] INFO [opsworks-agent(13209)]: master 13209: Spawned child 1/3
[2016-03-09 03:57:27] INFO [opsworks-agent(13209)]: master 13209: Spawned child 2/3
[2016-03-09 03:57:28] INFO [opsworks-agent(13209)]: master 13209: Spawned child 3/3
[2016-03-09 03:57:29] INFO [opsworks-agent(13209)]: Started master 13209 with 3
children
[2016-03-09 03:58:58] DEBUG [opsworks-agent(13237)]: Log upload: Running log upload
with /var/lib/aws/opsworks/chef/2016-03-09-03-57-28-01.log
# Logfile created on 2016-03-09 03:57:26 +0000 by logger.rb/41954
[2016-03-09 03:58:59] INFO [opsworks-agent(13237)]: Log upload: Successfully uploaded
log file /var/lib/aws/opsworks/chef/2016-03-09-03-57-28-01.log
```

How it works...

Once the OpsWorks instance is launched, it will refer to the associated OpsWorks layer to execute Chef recipes in the particular lifecycle event, as follows:

Lifecycle event	**Timing**
Setup	After instance has finished booting
Configure	On entering or leaving the online state, associating or disassociating an Elastic IP, attaching or detaching from Elastic Load Balancer
Deploy	Deploying the application (non custom layer)
Undeploy	When deleting an application (non custom layer)
Shutdown	Before shutdown of an instance

Again, the OpsWorks stack has one or more OpsWorks layers. In addition, each layer can associate one or more custom Chef recipes. Therefore, you should define the layer as an application role, such as frontend, backend, datastore, and so on.



For the Kubernetes setup, it should be defined as follows:

- The Kubernetes master layer
- The Kubernetes node layer
- The etcd layer

These Chef recipes will be described in the next section.

See also

In this recipe, we introduced the OpsWorks service that can define your stack and layers. In this recipe, we used the `CloudWatchLogs` Chef recipe as an example. However, Kubernetes can also be automated to install the agent via the OpsWorks Chef recipe. It is described in the following recipes of this chapter as well:

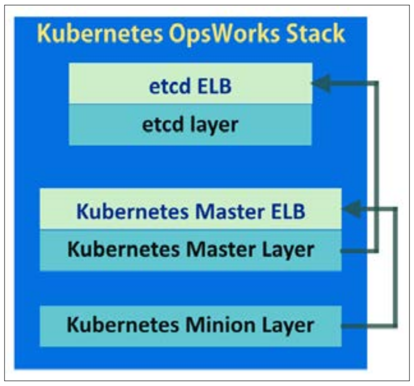
- *Building the Kubernetes infrastructure in AWS*
- *Auto-deploying Kubernetes through Chef recipes*
- *Using AWS CloudFormation for fast provisioning*

Auto-deploying Kubernetes through Chef recipes

To achieve fast deployment in AWS OpsWorks, we can write installation procedures in Chef recipes. Chef is a Ruby-based, auto-deployment managing tool (`https://www.chef.io`). It can help for program deployment and system configuration. In this recipe, we will show you how Chef works with the AWS OpsWorks.

Getting ready

In the following sections, we will show you how to use Chef recipes with the OpsWorks stack. Therefore, please prepare the OpsWorks environment. Based on the previous recipes in this chapter, we can build a Kubernetes stack with the following structure:



Let's consider that you have the same network settings mentioned in the recipe *Building the Kubernetes infrastructure in AWS*, which means that the VPC, subnets, route tables, and security groups are all ready for use. Then, we can apply the network environment directly in OpsWorks.

AWS region should be the same for resource utility

Although OpsWorks is a global service, we cannot combine the computing resources across different regions. Be aware that you need to choose the same AWS region of the network environment to create ELB and security groups.

Creating ELB and its security groups

As you can see in the previous stack structure, it is recommended to create ELBs beyond etcd and the Kubernetes master. Because both etcd and the master could be a cluster with multiple nodes, an ELB layer will provide the portal for the other application layers and balance the load to the working nodes. First, let's create the security groups of these two ELBs:

Rule name	**Inbound Protocol and port number**	**Source**
`My ELB of etcd SG`	▸ `80/tcp`	▸ `My Kubernetes master` ▸ `My Kubernetes node`
`My ELB of Kubernetes master SG`	▸ `8080/tcp`	▸ `My Kubernetes node SG`

Next, modify the existing security groups as follows. It will make sure that the network traffic is redirected to ELB first:

Rule name	**Inbound Protocol and port number**	**Source**
`My etcd SG`	▸ `7001/tcp` ▸ `4001/tcp`	▸ `My etcd SG` ▸ `My ELB of etcd SG`
`My Kubernetes master SG`	▸ `8080/tcp`	▸ `My ELB of Kubernetes master SG`

Then, we can create the ELBs with the specified security groups. Go to the EC2 console and click on **Load balancers** on the left-hand side menu. Create new ELBs with the following configurations:

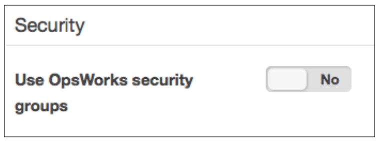
ELB name	**VPC**	**Listener Configuration (ELB Protocol:Port/ Instance Protocol: Port)**	**Subnets**	**Security Groups**	**Health check (Ping Protocol:Ping Port/Ping Path)**
`my-etcd-elb`	`My Kubernetes VPC (10.0.0.0/16)`	`HTTP:80/ HTTP:4001`	My Kubernetes Private A + My Kubernetes Private D	`My ELB of etcd SG`	HTTP:4001/ version
`my-k8s-master-elb`		`HTTP:8080/ HTTP:8080`		`My ELB of Kubernetes master SG`	HTTP:8080/ version

Except for the previous configurations, it is fine to leave other items with the default ones. You don't have to add any instances in ELBs.

Creating an OpsWorks stack

Defining an application stack in OpsWorks is simple and easy. Refer to the detailed step-by-step approach as follows:

1. Click on the **Add stack** button and you will enter the AWS OpsWorks console.
2. Fill in the following items. It is fine to leave the non-mentioned parts with default values:
    1. Choose **Chef 12 stack**.
    2. Give a stack name. For example, My Kubernetes Cluster.
    3. Assign region and VPC which you just configured for Kubernetes.
    4. For operating systems, a Linux system and the latest Amazon Linux are good for installation later on. For example, Amazon Linux 2015.09.
    5. Click on **Advanced>>** beneath the configurations and disable the block **Use OpsWorks security groups**. Since we have already set up the required security groups, this movement can prevent a lot of unnecessary security groups from being created automatically.



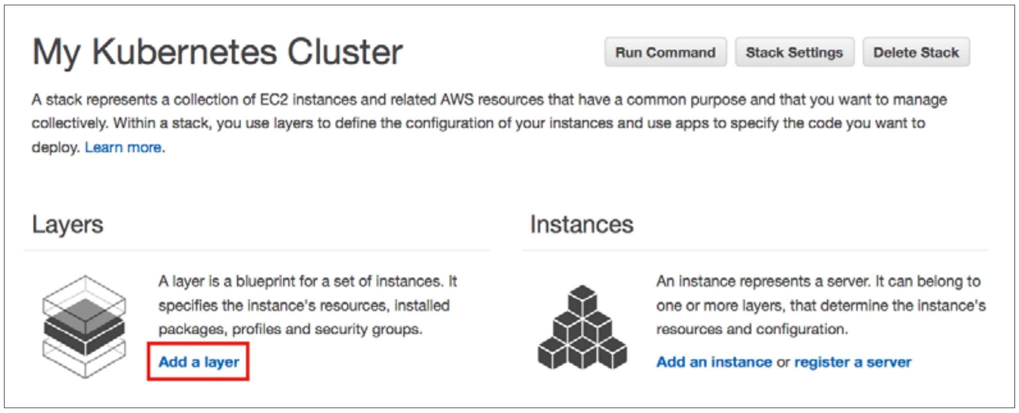
Security

Use OpsWorks security groups: No

    6. Now, go ahead and click on **Add stack** to create a Kubernetes stack.

Creating application layers

After we have the OpsWorks stack, let's create the application layers and attach ELBs:



My Kubernetes Cluster

A stack represents a collection of EC2 instances and related AWS resources that have a common purpose and that you want to manage collectively. Within a stack, you use layers to define the configuration of your instances and use apps to specify the code you want to deploy. Learn more.

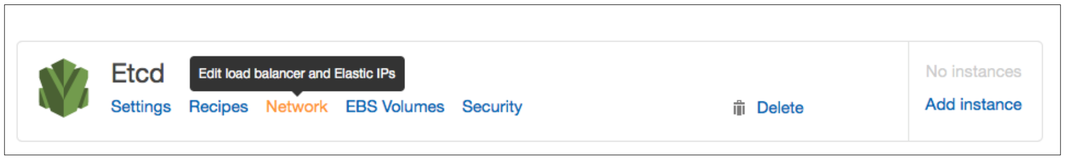
Layers

A layer is a blueprint for a set of instances. It specifies the instance's resources, installed packages, profiles and security groups.

Instances

An instance represents a server. It can belong to one or more layers, that determine the instance's resources and configuration.

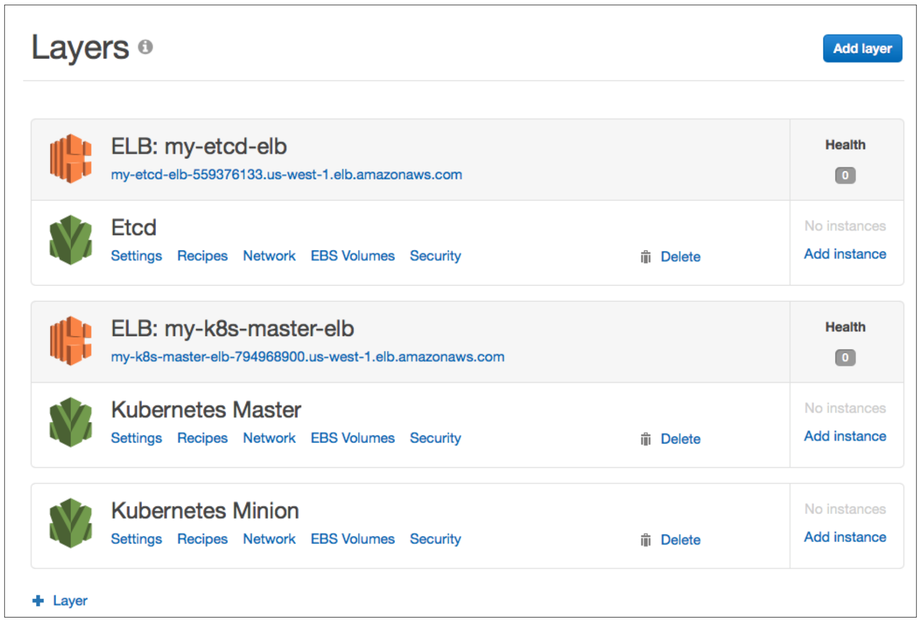
To create layers in the stack, click on **Add a layer** on the front page of the stack:


Etcd
Edit load balancer and Elastic IPs
Settings Recipes Network EBS Volumes Security
Delete
No instances
Add instance

We cannot attach an ELB to a layer at the initial step of creation. Click on **Network** for specified layer modifications after they are created. Help yourself to generate the following layers:

Layer Name	**Short Name (the name as the prefix of instance)**	**Security Group**	**Attached ELB**
Etcd	etcd	My etcd SG	my-etcd-elb
Kubernetes Master	k8s-master	My Kubernetes master SG My flannel SG (optional)	my-k8s-master-elb
Kubernetes Node	k8s-node	My Kubernetes node SG My flannel SG	

You will realize that the stack looks as follows, which is the same structure we mentioned at the beginning:


Layers
Add layer
ELB: my-etcd-elb
my-etcd-elb-559376133.us-west-1.elb.amazonaws.com
Health
0
Etcd
Settings Recipes Network EBS Volumes Security
Delete
No instances
Add instance
ELB: my-k8s-master-elb
my-k8s-master-elb-794968900.us-west-1.elb.amazonaws.com
Health
0
Kubernetes Master
Settings Recipes Network EBS Volumes Security
Delete
No instances
Add instance
Kubernetes Minion
Settings Recipes Network EBS Volumes Security
Delete
No instances
Add instance
Layer

Now, the OpsWorks stack has a basic system structure, but without customized Chef recipes. We will go through the recipe's contents and setup concepts in the next section.

How to do it...

For Kubernetes installation using the Chef recipe, we will prepare a GitHub repository with the following files and relative paths:

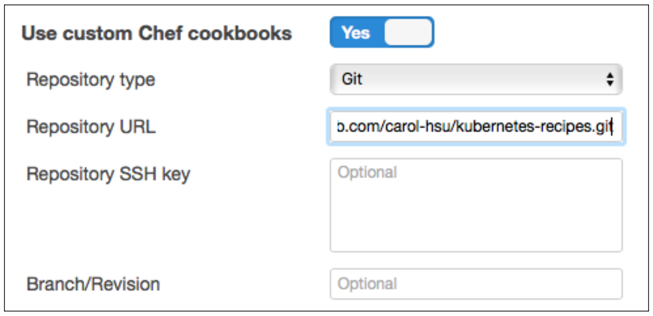
```
$ tree .
.
└── kubernetes
    ├── recipes
    │   ├── docker.rb
    │   ├── etcd-run.rb
    │   ├── etcd.rb
    │   ├── flanneld.rb
    │   ├── kubernetes-master-run.rb
    │   ├── kubernetes-master-setup.rb
    │   ├── kubernetes-node-run.rb
    │   ├── kubernetes-node-setup.rb
    │   └── kubernetes.rb
    └── templates
        └── default
            ├── docker.erb
            ├── etcd.erb
            ├── flanneld.erb
            ├── kubernetes-master.erb
            └── kubernetes-node.erb

4 directories, 14 files
```

In this section, different layers will illustrate the recipes and templates separately, but comprehensively. Create the repository and directory on the GitHub server in advance. It will help you set the customized stack configurations.

Stack configuration for custom recipes

In order to run the recipes in the stack, we are supposed to add two items in the configuration of the stack: one is the URL of the GitHub repo, which stored the recipe directory `kubernetes`, the other one is custom JSON. We may put some input parameters for the execution of recipes.

To modify the settings of the current stack, click on **Stack Settings** on the main page and then click on **Edit**. Enable **Use custom Chef Cookbooks**. You will find additional items showing up for codebase configuration. Then, put your GitHub repository URL for reference:



Use custom Chef cookbooks	Yes
Repository type	Git
Repository URL	ɔ.com/carol-hsu/kubernetes-recipes.git
Repository SSH key	Optional
Branch/Revision	Optional

You can also check our GitHub repository for more information via `https://github.com/kubernetes-cookbook/opsworks-recipes.git`.

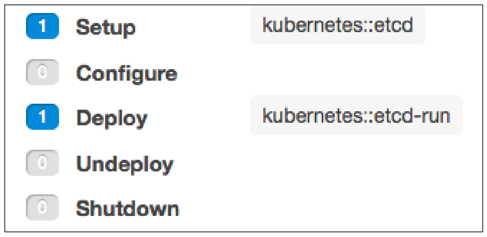
Next, at the block of **Advanced options**, please key in the following information in the item **Custom JSON**:

```
{
  "kubernetes": {
    "version":"1.1.8",
    "cluster_cidr":"192.168.0.0/16",
    "master_url":"<The DNS name of my-k8s-master-elb>"
  },
  "etcd": {
    "elb_url":"<The DNS name of my-etcd-elb>"
  }
}
```

The content of JSON is based on our Chef recipes. Users can define any key-value structure data. Usually, we put the one that may dynamically change by each deployment, for example, the version of the Kubernetes package. It would be better not to have hard code in our recipes, or some data that you don't want to show in the recipes can be made as input parameters. For example, the URL of ELB; it is a changeable value for each deployment. You possibly don't want others to know it. After configure the GitHub repository and custom JSON, we are ready to configuring the recipes in each layer.

Recipes for etcd

The lifecycle event of the etcd layer is as follows:



Setup kubernetes::etcd
Configure
Deploy kubernetes::etcd-run
Undeploy
Shutdown

We will separate the functionality of etcd recipes into two event stages: `kubernetes::etcd` is set at the **Setup** stage for etcd installation and configuration while `kubernetes::etcd-run` is at the **Deploy** stage to start the daemon:

```
$ cat ./kubernetes/recipes/etcd.rb
bash 'install_etcd' do
  user 'root'
  cwd '/tmp'
  code <<-EOH
  if [ ! -f /usr/local/bin/etcd ]; then
    wget --max-redirect 255 https://github.com/coreos/etcd/releases/
download/v2.2.5/etcd-v2.2.5-linux-amd64.tar.gz
    tar zxvf etcd-v2.2.5-linux-amd64.tar.gz
    cd etcd-v2.1.1-linux-amd64
    cp etcd etcdctl /usr/local/bin
  fi
  EOH
end

template "/etc/init.d/etcd" do
  mode "0755"
  owner "root"
  source "etcd.erb"
end
```

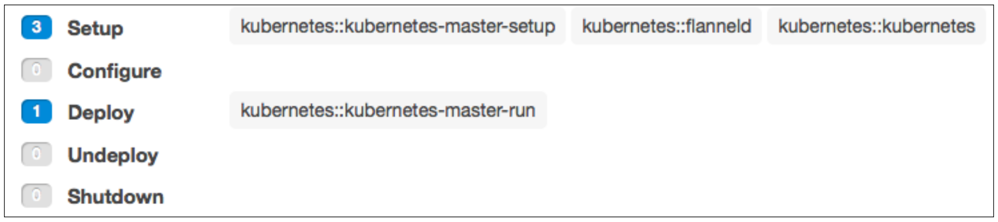
The recipe, `etcd.rb`, does the installation first. It will put the tarball at a temporary location and copy the binary file as a shared local command. To prevent the instance booting up from an already installed environment, we will add an `if` statement to check whether the system has the `etcd` command or not. There is a template `etcd.erb` working as a service configuration file. No dynamic input parameters are needed in this Chef template. It is also the same as we mentioned in the *Building datastore* recipe in *Chapter 1*, *Building Your Own Kubernetes*:

```
$ cat ./kubernetes/recipes/etcd-run.rb
service 'etcd' do
action [:enable,:start]
end
```

We have a short function in the recipe, `etcd-run.rb`, which enables and starts the etcd service. The stage, **Deploy** will run directly after **Setup**. Therefore, it is confirmed that the installation will finish before starting the service.

Recipes for the Kubernetes master

The recipes for installing the Kubernetes master are configured, as shown in the following screenshot:



3 Setup	kubernetes::kubernetes-master-setup	kubernetes::flanneld	kubernetes::kubernetes
0 Configure			
1 Deploy	kubernetes::kubernetes-master-run		
0 Undeploy			
0 Shutdown			

Just like the etcd layer, we use the **Setup** stage for installation and configuration file assignment. The recipe `kubernetes::kubernetes` is used for downloading the Kubernetes package. It will be shared to the node layer as well:

```
$ cat ./kubernetes/recipes/kubernetes.rb
bash 'install_kubernetes' do
  user 'root'
  cwd '/tmp'
  code <<-EOH
  if [[ $(ls /usr/local/bin/kubectl) ]]; then
    current_version=$(/usr/local/bin/kubectl version | awk 'NR==1' | awk
-F":\"v" '{ print $2 }' | awk -F"\"," '{ print $1 }')
    if [ "$current_version" -eq "#{node['kubernetes']['version']}" ];
then
```

```
        exit
    fi
  fi

  if [[ $(ls /usr/local/bin/kubelet) ]] ; then
    current_version=$(/usr/local/bin/kubelet --version | awk
-F"Kubernetes v" '{ print $2 }')
    if [ "$current_version" -eq "#{node['kubernetes']['version']}" ];
then
        exit
    fi
  fi
  rm -rf kubernetes/
  wget --max-redirect 255 https://github.com/GoogleCloudPlatform/
kubernetes/releases/download/v#{node['kubernetes']['version']}/
kubernetes.tar.gz -O kubernetes-#{node['kubernetes']['version']}.tar.gz
  tar zxvf kubernetes-#{node['kubernetes']['version']}.tar.gz
  cd kubernetes/server
  tar zxvf kubernetes-server-linux-amd64.tar.gz
  EOH
end
```

In this recipe, the value of the Kubernetes' version will be taken from custom JSON. We may specify the latest version of Kubernetes and enjoy the new features without modifying the recipe. Two nested `if` conditions are used to validate whether the Kubernetes binary file is deployed and updated to the version we requested. One is for the master and the other is for the node; if the condition is satisfied, package downloading will be ignored. The main Kubernetes master installation is written in the recipe `kubernetes-master-setup.rb`:

```
$ cat ./kubernetes/recipes/kubernetes-master-setup.rb
include_recipe 'kubernetes::kubernetes'

bash "master-file-copy" do
    user 'root'
    cwd '/tmp/kubernetes/server/kubernetes/server/bin'
    code <<-EOH
    if [[ $(ls /usr/local/bin/kubectl) ]]; then
        current_version=$(/usr/local/bin/kubectl version | awk 'NR==1' |
awk -F":\"v" '{ print $2 }' | awk -F"\"," '{ print $1 }')
```

```
        if [ "$current_version" -eq "#{node['kubernetes']['version']}" ];
then
            exit
        fi
    fi
    cp kubectl kube-apiserver kube-scheduler kube-controller-manager
kube-proxy /usr/local/bin/
    EOH
end

directory '/etc/kubernetes' do
    owner 'root'
    group 'root'
    mode '0755'
    subscribes :create, "bash[master-file-copy]", :immediately
    action :nothing
end

etcd_endpoint="http://#{node['etcd']['elb_url']}:80"

template "/etc/init.d/kubernetes-master" do
  mode "0755"
  owner "root"
  source "kubernetes-master.erb"
  variables({
    :etcd_server => etcd_endpoint,
    :cluster_cidr => node['kubernetes']['cluster_cidr']
  })
  subscribes :create, "bash[master-file-copy]", :immediately
    action :nothing
end
```

The first line of `kubernetes-master-setup.rb` is the solution to set the dependency within the same event stage. The resource type `include_recipe` requires you to execute the recipe `kubernetes::kubernetes` first. Then, it is able to copy the necessary binary file if the process does not exist in the version verifying condition, and next, prepare the suitable configuration file and directory for service.

Installing flanneld on the master node is an optional deployment. If so, on the Kubernetes master, we can access containers laid on flanneld:

```
$ cat ./kubernetes/recipes/flanneld.rb
bash 'install_flannel' do
  user 'root'
  cwd '/tmp'
  code <<-EOH
  if [ ! -f /usr/local/bin/flanneld ]; then
    wget --max-redirect 255 https://github.com/coreos/flannel/releases/
download/v0.5.2/flannel-0.5.2-linux-amd64.tar.gz
    tar zxvf flannel-0.5.2-linux-amd64.tar.gz
    cd flannel-0.5.2
    cp flanneld /usr/local/bin
    cp mk-docker-opts.sh /opt/
  fi
  EOH
end

template "/etc/init.d/flanneld" do
  mode "0755"
  owner "root"
  source "flanneld.erb"
  variables :elb_url => node['etcd']['elb_url']
  notifies :disable, 'service[flanneld]', :delayed
end

service "flanneld" do
  action :nothing
end
```

Especially, we will move a script file of flanneld to a specific location. This file helps to arrange the flanneld-defined network. Therefore, Docker will be based on the setting and limit its containers in the specific IP range. For the template, the value of etcd endpoints is an input parameter of the recipe. The recipe would inform the template to put the etcd ELB URL as an etcd endpoint:

```
$ cat ./kubernetes/templates/default/flanneld.erb
:
//above lines are ignored
```

```
start() {
  # Start daemon.
  echo -n $"Starting $prog: "
  daemon $prog \
    --etcd-endpoints=http://<%= @elb_url %> -ip-masq=true \
    > /var/log/flanneld.log 2>&1 &
  RETVAL=$?
  echo
  [ $RETVAL -eq 0 ] && touch $lockfile
  return $RETVAL
}
:
```

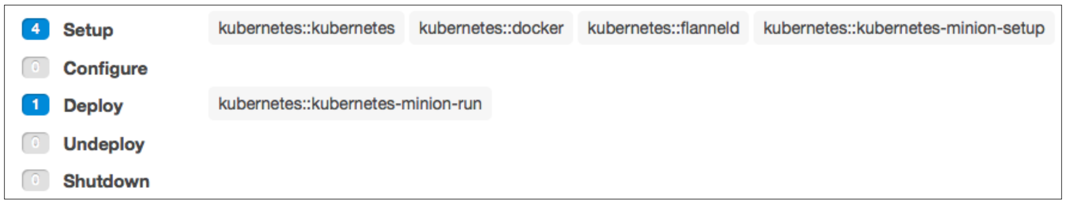
Finally, it is good for us to look at the recipe that starts the service:

```
$ cat ./kubernetes/recipes/kubernetes-master-run.rb
service "flanneld" do
  action :start
end

service "kubernetes-master" do
  action :start
end
```

It is straightforward to start these two independent daemons in this recipe.

Recipes for the Kubernetes node

With the previous experience, you now can easily understand the deployment of custom recipes and Chef functions. Then, we will go further to look at the recipes for the Kubernetes node layer:



4 Setup	kubernetes::kubernetes kubernetes::docker kubernetes::flanneld kubernetes::kubernetes-minion-setup
0 Configure	
1 Deploy	kubernetes::kubernetes-minion-run
0 Undeploy	
0 Shutdown	

Besides flanneld, we have to install Docker for running containers. However, an additional recipe `kubernetes::docker` is put at the **Setup** stage.

```
$ cat ./kubernetes/recipes/docker.rb
package "docker" do
  action :install
end

package "bridge-utils" do
  action :install
end

service "docker" do
  action :disable
end

template "/etc/sysconfig/docker" do
    mode "0644"
    owner "root"
    source "docker.erb"
end
```

We will install the necessary packages `docker` and `bridge-utils` in this recipe. But keep the Docker service stopped, since there is a service starting dependency:

```
$ cat ./kubernetes/templates/default/docker.erb
# /etc/sysconfig/docker
#
# Other arguments to pass to the docker daemon process
# These will be parsed by the sysv initscript and appended
# to the arguments list passed to docker -d

. /opt/mk-docker-opts.sh
. /run/docker_opts.env

INSECURE_REGISTRY="<YOUR_DOCKER_PRIVATE_REGISTRY>"
```

```
other_args="${DOCKER_OPTS} --insecure-registry ${INSECURE_REGISTRY}"
DOCKER_CERT_PATH=/etc/docker

# Location used for temporary files, such as those created by
# docker load and build operations. Default is /var/lib/docker/tmp
# Can be overriden by setting the following environment variable.
# DOCKER_TMPDIR=/var/tmp
```

The preceding template is called using `docker.rb`. Although there are no input parameters, it is worth mentioning that the script from flanneld will be triggered to run. It will generate the network settings for Docker and put it as the file `/run/docker_opts.env`.

Next, you will find that the node setup recipe is similar to the master one. We copy binaries, setup configuration files and directories, and keep the node service stopped:

```
$ cat ./kubernetes/recipes/kubernetes-node-setup.rb
include_recipe 'kubernetes::kubernetes'

bash "node-file-copy" do
    user 'root'
    cwd '/tmp/kubernetes/server/kubernetes/server/bin'
    code <<-EOH
    if [[ $(ls /usr/local/bin/kubelet) ]]; then
        current_version=$(/usr/local/bin/kubelet --version | awk
-F"Kubernetes v" '{ print $2 }')
        if [ "$current_version" -eq "#{node['kubernetes']['version']}" ];
then
            exit
        fi
    fi
    cp kubelet kube-proxy /usr/local/bin/
    EOH
end

directory '/var/lib/kubelet' do
    owner 'root'
    group 'root'
    mode '0755'
    subscribes :create, "bash[node-file-copy]", :immediately
    action :nothing
end

directory '/etc/kubernetes' do
    owner 'root'
    group 'root'
    mode '0755'
    subscribes :create, "bash[node-file-copy]", :immediately
    action :nothing
end

template "/etc/init.d/kubernetes-node" do
  mode "0755"
  owner "root"
  source "kubernetes-node.erb"
  variables :master_url => node['kubernetes']['master_url']
  subscribes :create, "bash[node-file-copy]", :immediately
  notifies :disable, 'service[kubernetes-node]', :delayed
   action :nothing
end

service "kubernetes-node" do
  action :nothing
end
```

On the other hand, the deploying recipe of node has more functions:

```
$ cat ./kubernetes/recipes/kubernetes-node-run.rb
service "flanneld" do
  action :start
  notifies :run, 'bash[wait_flanneld]', :delayed
end

bash 'wait_flanneld' do
  user 'root'
  cwd '/tmp'
  code <<-EOH
  tries=0
        while [ ! -f /run/flannel/subnet.env -a $tries -lt 10 ]; do
            sleep 1
            tries=$((tries + 1))
        done
  EOH

  action :nothing
  notifies :start, 'service[docker]', :delayed
end
service "docker" do
  action :nothing
  notifies :start, 'service[kubernetes-node]', :delayed
end

service "kubernetes-node" do
  action :nothing
end
```

Because of the dependence, flanneld should be started first and then Docker can be run according to the overlay network. Node service depends on running Docker, so it is the last service that needs to be started.

Starting the instances

Eventually, you have all the recipes ready to deploy a Kubernetes cluster. It is time to boot up some instances! Make sure that etcd is the earliest running instance. At that time, the Kubernetes master layer can run the master, which requires the datastore for resource information. After the master node is ready as well, create as many nodes as you want!

See also

In this recipe, you learned how to create your Kubernetes system automatically. Also, look at the following recipes:

- The *Building datastore*, *Creating an overlay network*, *Configuring master* and *Configuring nodes* recipes in *Chapter 1*, *Building Your Own Kubernetes*
- The *Clustering etcd* recipe in *Chapter 4*, *Building a High Availability Cluster*
- *Building the Kubernetes infrastructure in AWS*

- *Managing applications using AWS OpsWorks*
- *Using AWS CloudFormation for fast provisioning*
- The *Authentication and authorization* recipe in *Chapter 7*, *Advanced Cluster Administration*

Using AWS CloudFormation for fast provisioning

AWS CloudFormation is a service to make AWS resource creation easy. A simple JSON format text file could give you the power to create application infrastructure with just a few clicks. System administrators and developers can create, update, and manage their AWS resources easily without worrying about human error. In this section, we will leverage the content of the previous sections in this chapter and use CloudFormation to create them and launch instances with the Kubernetes setting automatically.

Getting ready

The unit of CloudFormation is a stack. One stack is created by one CloudFormation template, which is a text file listing AWS resources in the JSON format. Before we launch a CloudFormation stack using the CloudFormation template in the AWS console, let's get a deeper understanding of the tab names on the CloudFormation console:

Tab name	**Description**
`Overview`	Stack profile overview. Name, status and description are listed here
`Output`	The output fields of this stack
`Resources`	The resources listed in this stack
`Events`	The events when doing operations in this stack
`Template`	Text file in JSON format
`Parameters`	The input parameters of this stack
`Tags`	AWS tags for the resources
`Stack Policy`	Stack policy to use during update. This can prevent you from removing or updating resources accidentally

One CloudFormation template contains many sections; the descriptions are put in the following sample template:

```
{
   "AWSTemplateFormatVersion":"AWS CloudFormation templateversion
date",
   "Description":"stack description",
   "Metadata":{
    # put additional information for this template
   },
   "Parameters":{
    # user-specified the input of your template

   },
   "Mappings":{
    # using for define conditional parameter values and use it in the
template
   },
   "Conditions":{
    # use to define whether certain resources are created, configured
in a certain condition.
   },
   "Resources":{
    # major section in the template, use to create and configure AWS
resources
   },
   "Outputs":{
    # user-specified output
   }
}
```

We will use these three major sections:

- Parameters
- Resources
- Outputs

`Parameters` are the variable you might want to input when creating the stack, `Resources` are a major section for declaring AWS resource settings, and `Outputs` are the section you might want to expose to the CloudFormation UI so that it's easy to find the output information from a resource when a template is deployed.

Intrinsic Functions are built-in functions of AWS CloudFormation. They give you the power to link your resources together. It is a common use case that you need to link several resources together, but they'll know each other until runtime. In this case, the intrinsic function could be a perfect match to resolve this. Several intrinsic functions are provided in CloudFormation. In this case, we will use `Fn::GetAtt`, `Fn::GetAZs` and `Ref`.

The following table has their descriptions:

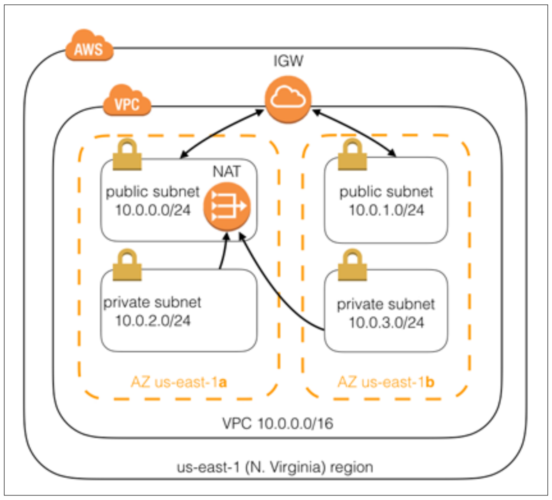
Functions	**Description**	**Usage**
`Fn::GetAtt`	Retrieve a value of an attribute from a resource	`{"Fn::GetAtt" : [ "logicalNameOfResource", "attributeName" ]}`
`Fn::GetAZs`	Return a list of AZs for the region	`{"Fn::GetAZs" : "us-east-1"}`
`Fn::Select`	Select a value from a list	`{ "Fn::Select" : [ index, listOfObjects ]}`
`Ref`	Return a value from a logical name or parameter	`{"Ref" : "logicalName"}`

How to do it...

Instead of launching a bit template with a thousand lines, we'll split it into two: one is for network resources-only, another one is application-only. Both the templates are available on our GitHub repository via `https://github.com/kubernetes-cookbook/cloudformation`.

Creating a network infrastructure

Let's review the following infrastructure listed in the *Building the Kubernetes infrastructure in AWS* section. We will create one VPC with `10.0.0.0/16` with two public subnets and two private subnets in it. Besides these, we will create one Internet Gateway and add related route table rules to a public subnet in order to route the traffic to the outside world. We will also create a NAT Gateway, which is located in the public subnet with one Elastic IP, to ensure a private subnet can get access to the Internet:



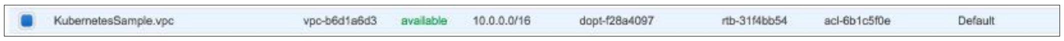
How do we do that? At the beginning of the template, we'll define two parameters: one is `Prefix` and another is `CIDRPrefix`. `Prefix` is a prefix used to name the resource we're going to create. `CIDRPrefix` is two sections of an IP address that we'd like to create; the default is `10.0`. We will also set the length constraint to it:

```
"Parameters":{
      "Prefix":{
         "Description":"Prefix of resources",
         "Type":"String",
         "Default":"KubernetesSample",
         "MinLength":"1",
         "MaxLength":"24",
         "ConstraintDescription":"Length is too long"
      },
      "CIDRPrefix":{
         "Description":"Network cidr prefix",
         "Type":"String",
         "Default":"10.0",
         "MinLength":"1",
         "MaxLength":"8",
         "ConstraintDescription":"Length is too long"
      }
   }
```

Then, we will start describing the `Resources` section. For detailed resource types and attributes, we recommend you visit the AWS Documentation via `http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-template-resource-type-ref.html`:

```
"VPC":{
         "Type":"AWS::EC2::VPC",
         "Properties":{
            "CidrBlock":{
               "Fn::Join":[
                  ".",
                  [
                     {
                        "Ref":"CIDRPrefix"
                     },
                     "0.0/16"
                  ]
               ]
            },
            "EnableDnsHostnames":"true",
            "Tags":[
               {
                  "Key":"Name",
                  "Value":{
                     "Fn::Join":[
                        ".",
                        [
                           {
                              "Ref":"Prefix"
                           },
                           "vpc"
                        ]
                     ]
                  }
               },
               {
                  "Key":"EnvName",
                  "Value":{
                     "Ref":"Prefix"
                  }
               }
            ]
         }
      }
```

Here, we'll create one resource with the logical name `VPC` and type `AWS::EC2::VPC`. Please note that the logical name is important and it cannot be duplicated in one template. You could use `{"Ref": "VPC"}` in any other resource in this template to refer to `VPCId`. The name of VPC will be `$Prefix.vpc` with CIDR `$CIDRPrefix.0.0/16`. The following image is that of a created VPC:



```
KubernetesSample.vpc    vpc-b6d1a6d3    available    10.0.0.0/16    dopt-f28a4097    rtb-31f4bb54    acl-6b1c5f0e    Default
```

Next, we'll create the first public subnet with CIDR `$CIDRPrefix.0.0/24`. Note that `{Fn::GetAZs:""}` will return a list of all the available AZs. We'll use `Fn::Select` to select the first element with index 0:

```
        "SubnetPublicA":{
            "Type":"AWS::EC2::Subnet",
            "Properties":{
               "VpcId":{
                  "Ref":"VPC"
               },
               "CidrBlock":{
                  "Fn::Join":[
                     ".",
                     [
                        {
                           "Ref":"CIDRPrefix"
                        },
                        "0.0/24"
                     ]
                  ]
               },
               "AvailabilityZone":{
                  "Fn::Select":[
                     "0",
                     {
                        "Fn::GetAZs":""
                     }
                  ]
               },
               "Tags":[
                  {
                     "Key":"Name",
                     "Value":{
                        "Fn::Join":[
                           ".",
                           [
                              {
                                 "Ref":"Prefix"
                              },
                              "public",
                              "subnet",
                              "A"
                           ]
                        ]
                     }
                  },
                  {
                     "Key":"EnvName",
                     "Value":{
                        "Ref":"Prefix"
                     }
                  }
               ]
            }
         }
```

The second public subnet and two private subnets are the same as the first one just with a different CIDR `$CIDRPrefix.1.0/24`. The difference between public and private subnets are whether they're Internet reachable or not. Typically, an instance in a public subnet will have a public IP or an Elastic IP with it that is Internet reachable. However, a private subnet cannot be reachable from the Internet, except using a bastion host or via VPN. The difference in the AWS setting is the routes in route tables. In order to let your instances communicate with the Internet, we should create an Internet Gateway to a public subnet and a NAT Gateway to a private subnet:

```
"InternetGateway":{
   "Type":"AWS::EC2::InternetGateway",
   "Properties":{
      "Tags":[
         {
            "Key":"Stack",
            "Value":{
               "Ref":"AWS::StackId"
            }
         },
         {
            "Key":"Name",
            "Value":{
               "Fn::Join":[
                  ".",
                  [
                     {
                        "Ref":"Prefix"
                     },
                     "vpc",
                     "igw"
                  ]
               ]
            }
         },
         {
            "Key":"EnvName",
            "Value":{
               "Ref":"Prefix"
            }
         }
      ]
   }
},
"GatewayAttachment":{
        "Type":"AWS::EC2::VPCGatewayAttachment",
        "Properties":{
            "VpcId":{
                "Ref":"VPC"
            },
            "InternetGatewayId":{
                "Ref":"InternetGateway"
            }
        }
    }
```

We will declare one Internet Gateway with the name `$Prefix.vpc.igw` and the logical name `InternetGateway`; we will also attach it to VPC. Then, let's create `NatGateway`. `NatGateway` needs one EIP by default, so we'll create it first and use the `DependsOn` function to tell CloudFormation that the `NatGateway` resource must be created after `NatGatewayEIP`. Note that there is `AllocationId` in the properties of `NatGateway` rather than the Gateway ID. We'll use the intrinsic function `Fn::GetAtt` to get the attribute `AllocationId` from the resource `NatGatewayEIP`:

```
    "NatGatewayEIP":{
        "Type":"AWS::EC2::EIP",
        "DependsOn":"GatewayAttachment",
        "Properties":{
            "Domain":"vpc"
        }
    },
    "NatGateway":{
        "Type":"AWS::EC2::NatGateway",
        "DependsOn":"NatGatewayEIP",
        "Properties":{
            "AllocationId":{
                "Fn::GetAtt":[
                    "NatGatewayEIP",
                    "AllocationId"
                ]
            },
            "SubnetId":{
                "Ref":"SubnetPublicA"
            }
        }
    }
```

Time to create a route table for public subnets:

```
"RouteTableInternet":{
   "Type":"AWS::EC2::RouteTable",
   "Properties":{
      "VpcId":{
         "Ref":"VPC"
      },
      "Tags":[
         {
            "Key":"Stack",
            "Value":{
               "Ref":"AWS::StackId"
            }
         },
         {
            "Key":"Name",
            "Value":{
               "Fn::Join":[
                  ".",
                  [
                     {
                        "Ref":"Prefix"
                     },
                     "internet",
                     "routetable"
                  ]
               ]
            }
         },
         {
            "Key":"EnvName",
            "Value":{
               "Ref":"Prefix"
            }
         }
      ]
   }
}
```

What about private subnets? You could use the same declaration; just change the logical name to `RouteTableNat`. After creating a route table, let's create the routes:

```
"RouteInternet":{
   "Type":"AWS::EC2::Route",
   "DependsOn":"GatewayAttachment",
   "Properties":{
      "RouteTableId":{
         "Ref":"RouteTableInternet"
      },
      "DestinationCidrBlock":"0.0.0.0/0",
      "GatewayId":{
         "Ref":"InternetGateway"
      }
   }
}
```

This route is for the route table of a public subnet. It will relocate to the `RouteTableInternet` table and route the packets to `InternetGatway` if the destination CIDR is `0.0.0.0/0`. Let's take a look at a private subnet route:

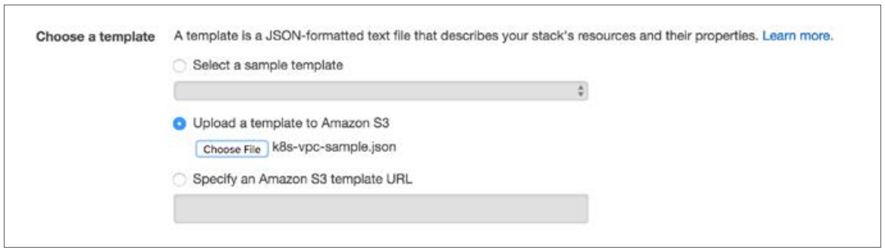
```
"RouteNat":{
   "Type":"AWS::EC2::Route",
   "DependsOn":"RouteTableNat",
   "Properties":{
      "RouteTableId":{
         "Ref":"RouteTableNat"
      },
      "DestinationCidrBlock":"0.0.0.0/0",
      "NatGatewayId":{
         "Ref":"NatGateway"
      }
   }
}
```

It is pretty much the same with `RouteInternet` but route the packets to `NatGateway` if there are any, to `0.0.0.0/0`. Wait, what's the relation between subnet and a route table? We didn't see any declaration indicate the rules in a certain subnet. We have to use `SubnetRouteTableAssociation` to define their relation. The following examples define both public subnet and private subnet; you might also add a second public/private subnet by copying them:

```
"SubnetRouteTableInternetAssociationA":{
   "Type":"AWS::EC2::SubnetRouteTableAssociation",
   "Properties":{
      "SubnetId":{
```

```
                "Ref":"SubnetPublicA"
            },
            "RouteTableId":{
                "Ref":"RouteTableInternet"
            }
        }
    },
    "SubnetRouteTableNatAssociationA":{
        "Type":"AWS::EC2::SubnetRouteTableAssociation",
        "Properties":{
            "SubnetId":{
                "Ref":"SubnetPrivateA"
            },
            "RouteTableId":{
                "Ref":"RouteTableNat"
            }
        }
    }
```

We're done for the network infrastructure. Then, let's launch it from the AWS console. First, just click and launch a stack and select the VPC sample template.



Choose a template: A template is a JSON-formatted text file that describes your stack's resources and their properties. Learn more.

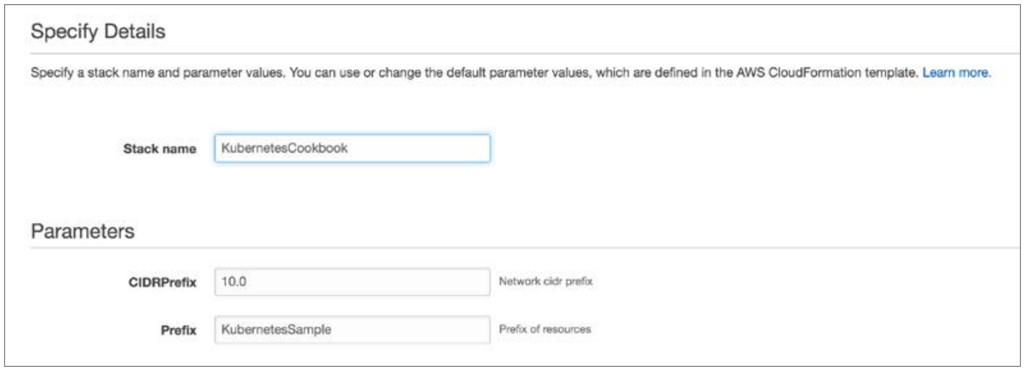
Select a sample template

[✓ Upload a template to Amazon S3]

Choose File k8s-vpc-sample.json

Specify an Amazon S3 template URL

Click on **next**; you will see the parameters' pages. It has its own default value, but you could change it at the creation/update time of the stack.



Specify Details

Specify a stack name and parameter values. You can use or change the default parameter values, which are defined in the AWS CloudFormation template. Learn more.

Stack name: KubernetesCookbook

Parameters

CIDRPrefix	10.0	Network cidr prefix
Prefix	KubernetesSample	Prefix of resources

After you click on **finish**, CloudFormation will start creating the resources you claim on the template. It will return Status as **CREATE_COMPLETE** after completion.

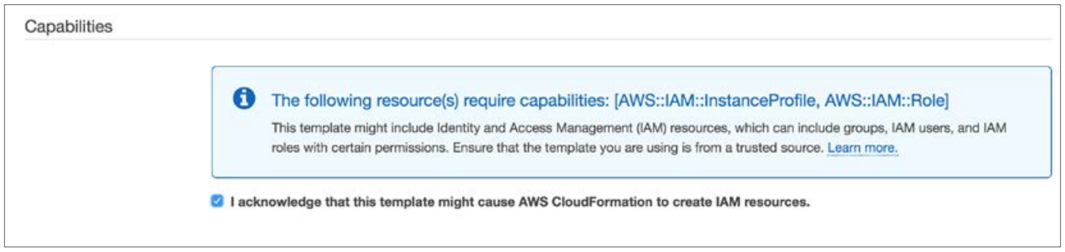
Creating OpsWorks for application management

For application management, we'll leverage OpsWorks, which is an application lifecycle management in AWS. Please refer to the previous two sections to know more about OpsWorks and Chef. Here, we'll describe how to automate creating the OpsWorks stack and related resources.

We'll have eight parameters here. Add `K8sMasterBaAccount`, `K8sMasterBaPassword`, and `EtcdBaPassword` as the basic authentication for Kubernetes master and etcd. We will also put the VPC ID and the private subnet ID here as the input, which are created in the previous sample. As parameters, we could use the type `AWS::EC2::VPC::Id` as a drop-down list in the UI. Please refer to the supported type in the AWS Documentation via `http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/parameters-section-structure.html`:

```
"Parameters":{
      "Prefix":{
         "Description":"Prefix of resources",
         "Type":"String",
         "Default":"KubernetesSample",
         "MinLength":"1",
         "MaxLength":"24",
         "ConstraintDescription":"Length is too long"
      },
      "PrivateNetworkCIDR":{
         "Default":"192.168.0.0/16",
         "Description":"Desired Private Network CIDR or Flanneld (must 
not overrap VPC CIDR)",
         "Type":"String",
         "MinLength":"9",
         "AllowedPattern":"\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}\\.\\
d{1,3}/\\d{1,2}",
         "ConstraintDescription":"PrivateNetworkCIDR must be IPv4 
format"
      },
      "VPCId":{
         "Description":"VPC Id",
         "Type":"AWS::EC2::VPC::Id"
      },
      "SubnetPrivateIdA":{
         "Description":"Private SubnetA",
         "Type":"AWS::EC2::Subnet::Id"
      },
      "SubnetPrivateIdB":{
         "Description":"Private SubnetB",
         "Default":"subnet-9007ecc9",
         "Type":"AWS::EC2::Subnet::Id"
      },
      "K8sMasterBaAccount":{
         "Default":"admin",
         "Description":"The account of basic authentication for k8s
Master",
         "Type":"String",
         "MinLength":"1",
         "MaxLength":"75",
         "AllowedPattern":"[a-zA-Z0-9]*",
         "ConstraintDescription":"Account and Password should follow
Base64 pattern"
      },
      "K8sMasterBaPassword":{
         "Default":"Passw0rd",
         "Description":"The password of basic authentication for k8s
Master",
         "Type":"String",
         "MinLength":"1",
         "MaxLength":"75",
         "NoEcho":"true",
         "AllowedPattern":"[a-zA-Z0-9]*",
         "ConstraintDescription":"Account and Password should follow
Base64 pattern"
      },
      "EtcdBaPassword":{
         "Default":"Passw0rd",
         "Description":"The password of basic authentication for
Etcd",
         "Type":"String",
         "MinLength":"1",
         "MaxLength":"71",
         "NoEcho":"true",
         "AllowedPattern":"[a-zA-Z0-9]*",
         "ConstraintDescription":"Password should follow Base64
pattern"
      }
   }
```

Before we get started with the OpsWorks stack, we need to create two IAM roles for it. One is a service role, which is used to launch instances, attaching ELB, and so on. Another is an instance role, which is to define the permission for what your OpsWorks instances can perform, to access the AWS resources. Here, we won't access any AWS resources from EC2, so we could just create a skeleton. Please note that you'll need to have IAM permission when you launch CloudFormation with IAM creation. Click on the following checkbox when launching the stack:


Capabilities

The following resource(s) require capabilities: [AWS::IAM::InstanceProfile, AWS::IAM::Role]

This template might include Identity and Access Management (IAM) resources, which can include groups, IAM users, and IAM roles with certain permissions. Ensure that the template you are using is from a trusted source. Learn more.

[✓] I acknowledge that this template might cause AWS CloudFormation to create IAM resources.

In the `SecurityGroup` section, we will define each ingress and egress to a set of machines. We'll take the Kubernetes master as an example. Since we put ELB in front of the master and ELB to retain the flexibility for future HA settings. Using ELB, we'll need to create a security group to ELB and point the ingress of the Kubernetes master that can be in touch with `8080` and `6443` from the ELB security group. Following is the example of the security group for the Kubernetes master; it opens `80` and `8080` to the outside world:

```
"SecurityGroupELBKubMaster":{
     "Type":"AWS::EC2::SecurityGroup",
     "Properties":{
        "GroupDescription":{
           "Ref":"Prefix"
        },
        "SecurityGroupIngress":[
           {
              "IpProtocol":"tcp",
              "FromPort":"80",
              "ToPort":"80",
              "CidrIp":"0.0.0.0/0"
           },
           {
              "IpProtocol":"tcp",
              "FromPort":"8080",
              "ToPort":"8080",
              "SourceSecurityGroupId":{
                 "Ref":"SecurityGroupKubNode"
              }
           }
        ],
        "VpcId":{
           "Ref":"VPCId"
        },
        "Tags":[
           {
              "Key":"Application",
              "Value":{
                 "Ref":"AWS::StackId"
              }
           },
           {
              "Key":"Name",
              "Value":{
                 "Fn::Join":[
                    "-",
                    [
                       {
                          "Ref":"Prefix"
                       },
                       "SGElbKubMaster"
                    ]
                 ]
              }
           }
        ]
     }
  },
```

Here is the example of the Kubernetes master instance set. It allows you to receive traffic from `8080` and `6443` from its ELB. We will open the SSH port to use the `kubectl` command:

```
         "SecurityGroupKubMaster":{
             "Type":"AWS::EC2::SecurityGroup",
             "Properties":{
                "GroupDescription":{
                   "Ref":"Prefix"
                },
                "SecurityGroupIngress":[
                   {
                      "IpProtocol":"tcp",
                      "FromPort":"22",
                      "ToPort":"22",
                      "CidrIp":"0.0.0.0/0"
                   },
                   {
                      "IpProtocol":"tcp",
                      "FromPort":"8080",
                      "ToPort":"8080",
                      "SourceSecurityGroupId":{
                         "Ref":"SecurityGroupELBKubMaster"
                      }
                   },
                   {
                      "IpProtocol":"tcp",
                      "FromPort":"6443",
                      "ToPort":"6443",
                      "SourceSecurityGroupId":{
                         "Ref":"SecurityGroupELBKubMaster"
                      }
                   }
                ],
                "VpcId":{
                   "Ref":"VPCId"
                },
                "Tags":[
                   {
                      "Key":"Application",
                      "Value":{
                         "Ref":"AWS::StackId"
                      }
                   },
                   {
                      "Key":"Name",
                      "Value":{
                         "Fn::Join":[
                            "-",
                            [
                               {
                                  "Ref":"Prefix"
                               },
                               "SG-KubMaster"
                            ]
                         ]
                      }
                   }
                ]
             }
          }
```

Please refer to the examples from the book about the security group setting of etcd and node. Next, we'll start creating the OpsWorks stack. `CustomJson` acts as the input of the Chef recipe. If there is anything that Chef doesn't know at the beginning, you will have to pass the parameters into `CustomJson`:

```
"OpsWorksStack":{
   "Type":"AWS::OpsWorks::Stack",
   "Properties":{
      "DefaultInstanceProfileArn":{
         "Fn::GetAtt":[
            "RootInstanceProfile",
            "Arn"
         ]
      },
      "CustomJson":{
         "kubernetes":{
            "cluster_cidr":{
               "Ref":"PrivateNetworkCIDR"
            },
            "version":"1.1.3",
            "master_url":{
               "Fn::GetAtt":[
                  "ELBKubMaster",
                  "DNSName"
               ]
            }
         },
         "ba":{
            "account":{
               "Ref":"K8sMasterBaAccount"
            },
            "password":{
               "Ref":"K8sMasterBaPassword"
            },
            "uid":1234
         },
         "etcd":{
            "password":{
               "Ref":"EtcdBaPassword"
            },
            "elb_url":{
               "Fn::GetAtt":[
                  "ELBEtcd",
                  "DNSName"
```

```
                    ]
                 }
              },
              "opsworks_berkshelf":{
                 "debug":true
              }
           },
           "ConfigurationManager":{
              "Name":"Chef",
              "Version":"11.10"
           },
           "UseCustomCookbooks":"true",
           "UseOpsworksSecurityGroups":"false",
           "CustomCookbooksSource":{
              "Type":"git",
              "Url":"https://github.com/kubernetes-cookbook/opsworks-
recipes.git"
           },
           "ChefConfiguration":{
              "ManageBerkshelf":"true"
           },
           "DefaultOs":"Red Hat Enterprise Linux 7",
           "DefaultSubnetId":{
              "Ref":"SubnetPrivateIdA"
           },
           "Name":{
              "Ref":"Prefix"
           },
           "ServiceRoleArn":{
              "Fn::GetAtt":[
                 "OpsWorksServiceRole",
                 "Arn"
              ]
           },
           "VpcId":{
              "Ref":"VPCId"
           }
        }
     },
```

After creating the stack, we can start creating each layer. Take the Kubernetes master as an example:

```
     "OpsWorksLayerKubMaster":{
        "Type":"AWS::OpsWorks::Layer",
        "Properties":{
           "Name":"Kubernetes Master",
           "Shortname":"kube-master",
           "AutoAssignElasticIps":"false",
           "AutoAssignPublicIps":"false",
           "CustomSecurityGroupIds":[
              {
                 "Ref":"SecurityGroupKubMaster"
              }
           ],
           "EnableAutoHealing":"false",
           "StackId":{
              "Ref":"OpsWorksStack"
           },
           "Type":"custom",
           "CustomRecipes":{
              "Setup":[
                 "kubernetes-rhel::flanneld",
                 "kubernetes-rhel::repo-setup",
                 "kubernetes-rhel::master-setup"
              ],
              "Deploy":[
                 "kubernetes-rhel::master-run"
              ]
           }
        }
     },
```

The run list of Chef in this layer is `["kubernetes-rhel::flanneld", "kubernetes-rhel::repo-setup", "kubernetes-rhel::master-setup"]` and `["kubernetes-rhel::master-run"]` at the deployment stage. For the run list of etcd, we'll use `["kubernetes-rhel::etcd", "kubernetes-rhel::etcd-auth"]` to perform etcd provisioning and authentication setting. For the Kubernetes nodes, we'll use `["kubernetes-rhel::flanneld", "kubernetes-rhel::docker-engine", "kubernetes-rhel::repo-setup", "kubernetes-rhel::node-setup"]` as a run list at the setup stage and `["kubernetes-rhel::node-run"]` at the deployment stage.

After setting up the layer, we can create ELB and attach it to the stack. The target of health check for the instance is `HTTP:8080/version`. It will then receive traffic from the port `80` and redirect it to the `6443` port in the master instances, and receive traffic from `8080` to the instance port `8080`:

```
      "ELBKubMaster":{
        "DependsOn":"SecurityGroupELBKubMaster",
        "Type":"AWS::ElasticLoadBalancing::LoadBalancer",
        "Properties":{
           "LoadBalancerName":{
              "Fn::Join":[
                 "-",
                 [
                    {
                       "Ref":"Prefix"
                    },
                    "Kub"
                 ]
              ]
           },
           "Scheme":"internal",
           "Listeners":[
              {
                 "LoadBalancerPort":"80",
                 "InstancePort":"6443",
                 "Protocol":"HTTP",
                 "InstanceProtocol":"HTTPS"
              },
              {
                 "LoadBalancerPort":"8080",
                 "InstancePort":"8080",
                 "Protocol":"HTTP",
                 "InstanceProtocol":"HTTP"
              }
           ],
           "HealthCheck":{
              "Target":"HTTP:8080/version",
              "HealthyThreshold":"2",
              "UnhealthyThreshold":"10",
              "Interval":"10",
              "Timeout":"5"
           },
           "Subnets":[
              {
                 "Ref":"SubnetPrivateIdA"
              },
              {
                 "Ref":"SubnetPrivateIdB"
              }
           ],
           "SecurityGroups":[
              {
                 "Fn::GetAtt":[
                    "SecurityGroupELBKubMaster",
                    "GroupId"
                 ]
              }
           ]
        }
     }
```

After creating the master ELB, let's attach it to the OpsWorks stack:

```
        "OpsWorksELBAttachKubMaster":{
           "Type":"AWS::OpsWorks::ElasticLoadBalancerAttachment",
           "Properties":{
              "ElasticLoadBalancerName":{
                 "Ref":"ELBKubMaster"
              },
              "LayerId":{
                 "Ref":"OpsWorksLayerKubMaster"
              }
           }
        }
```

That's it! The ELB of etcd is the same setting, but listen to `HTTP:4001/version` as a health check and redirect `80` traffic from the outside to the instance port `4001`. For a detailed example, please refer to our code reference. After launching the second sample template, you should be able to see the OpsWorks stacks, layers, security groups, IAM, and ELBs. If you want to launch by default with CloudFormation, just add the resource type with `AWS::OpsWorks::Instance`, specify the spec, and you are all set.

See also

In this recipe, we got an understanding on how to write and deploy an AWS CloudFormation template. Please check out the following recipes as well:

- The *Exploring architecture* recipe in *Chapter 1*, *Building Your Own Kubernetes*
- *Building the Kubernetes infrastructure in AWS*
- *Managing applications using AWS OpsWorks*
- *Auto-deploying Kubernetes through Chef recipes*

7
Advanced Cluster Administration

In this chapter, we will cover:

- Advanced settings in kubeconfig
- Setting resource in nodes
- Playing with WebUI
- Working with a RESTful API
- Authentication and authorization

Introduction

We will go through advanced topics on administration in this chapter. First, you will learn how to use kubeconfig to manage different clusters. Then, we will work on computing resources in nodes. Kubernetes provides a friendly user interface to illustrate the current status of resources, such as the replication controller, nodes, and pods. You will learn how to build and administrate it.

Next, you will learn how to work with the RESTful API that Kubernetes exposes. It will be a handy way to integrate with other systems. Finally, we want to build a secure cluster; the last section will go through how to set up authentication and authorization in Kubernetes.

Advanced settings in kubeconfig

kubeconfig is a configuration file to manage cluster, context, and authentication settings in Kubernetes. Using the kubeconfig file, we are able to set different cluster credentials, users, and namespaces to switch between clusters or contexts within a cluster. It can be configured via the command line using the `kubectl config` subcommand or a configuration file directly. In this section, we'll describe how to use `kubectl config` to manipulate kubeconfig and how to input a kubeconfig file directly.

Getting ready

Before you start to modify kubeconfig, you should clearly know what your security policies are. Using `kubectl config view`, you can check your current settings:

```
// check current kubeconfig file
# kubectl config view
apiVersion: v1
clusters: []
contexts: []
current-context: ""
kind: Config
preferences: {}
users: []
```

We can see currently we do not have any specific settings in kubeconfig.

How to do it...

Assume we have two clusters, one is under localhost `http://localhost:8080` and another is in the remote `http://remotehost:8080` named `remotehost`. In the example, we'll use localhost as the main console to switch the cluster via context changes. We then run different number of nginx into both the clusters and make sure the pods are all running:

```
// in localhost cluster
# kubectl run localnginx --image=nginx --replicas=2 --port=80
replicationcontroller "localnginx" created
// check pods are running
# kubectl get pods
NAME                  READY     STATUS        RESTARTS    AGE
```

```
localnginx-1blru   1/1       Running       0          1m
localnginx-p6cyo   1/1       Running       0          1m

// in remotehost cluster
# kubectl run remotenginx --image=nginx --replicas=4 --port=80
replicationcontroller "remotenginx" created
// check pods are running
# kubectl get pods
NAME                 READY     STATUS    RESTARTS   AGE
remotenginx-6wz5c    1/1       Running   0          1m
remotenginx-7v5in    1/1       Running   0          1m
remotenginx-c7go6    1/1       Running   0          1m
remotenginx-r1mf6    1/1       Running   0          1m
```

Setting a new credential

First, we will set up two credentials for each cluster. Use `kubectl config set-credentials <nickname>` for adding credential into kubeconfig. There are different authentication methods supported in Kubernetes. We could use a password, client-certificate, or token. In the example, we'll use HTTP basic authentication for simplifying the scenario. Kubernetes also supports client certificate and token authentications. For more information, please refer to the kubeconfig set-credential page: `http://kubernetes.io/docs/user-guide/kubectl/kubectl_config_set-credentials`:

```
// in localhost cluster, add a user `userlocal` with nickname localhost/
myself
#  kubectl config set-credentials localhost/myself --username=userlocal
--password=passwordlocal
user "localhost/myself" set.

// in localhost cluster, add a user `userremote` with nickname
remotehost/myself
#  kubectl config set-credentials remotehost/myself --username=userremote
--password=passwordremote
user "remotehost/myself" set.
```

Let's check out the current view:

```
# kubectl config view
apiVersion: v1
clusters: []
contexts: []
```

```
current-context: ""
kind: Config
preferences: {}
users:
- name: localhost/myself
  user:
    password: passwordlocal
    username: userlocal
- name: remotehost/myself
  user:
    password: passwordremote
    username: userremote
```

We can find currently that we have two sets of credentials with nicknames `localhost/myself` and `remotehost/myself`. Next, we'll set the clusters into the management.

Setting a new cluster

To set a new cluster, we will need the `kubectl config set-cluster <nickname>` command. We will need the `--server` parameter for accessing clusters. Adding `-insecure-skip-tls-verify` will not check the server's certificate. If you are setting up a trusted server with HTTPS, you will need to replace `-insecure-skip-tls-verify` to `--certificate-authority=$PATH_OF_CERT --embed-certs=true`. For more information, check out the kubeconfig set-cluster page: `http://kubernetes.io/docs/user-guide/kubectl/kubectl_config_set-cluster`:

```
// in localhost cluster: add http://localhost:8080 as localhost
# kubectl config set-cluster localhost --insecure-skip-tls-verify=true
--server=http://localhost:8080
cluster "localhost" set.

// in localhost cluster: add http://remote:8080 as localhost
# kubectl config set-cluster remotehost --insecure-skip-tls-verify=true
--server=http://remotehost:8080
cluster "remotehost" set.
```

Let's check out the current view now. The setting exactly reflects what we've set:

```
// check current view
# kubectl config view
apiVersion: v1
```

```
clusters:
- cluster:
    insecure-skip-tls-verify: true
    server: http://localhost:8080
  name: localhost
- cluster:
    insecure-skip-tls-verify: true
    server: http://remotehost:8080
  name: remotehost
contexts: []
current-context: ""
kind: Config
preferences: {}
users:
- name: localhost/myself
  user:
    password: passwordlocal
    username: userlocal
- name: remotehost/myself
  user:
    password: passwordremote
    username: userremote
```

Note that we do not associate anything between users and clusters yet. We will link them via context.

Setting and changing the current context

One context contains a cluster, namespace, and user. `kubectl` will use the specified user information and namespace to send requests to the cluster. To set up a context, we will use `kubectl config set-context <context nickname> --user=<user nickname> --namespace=<namespace> --cluster=< cluster nickname>` to create it:

```
// in localhost cluster: set a context named default/localhost/myself for
localhost cluster
# kubectl config set-context default/localhost/myself --user=localhost/
myself --namespace=default --cluster=localhost
context "default/localhost/myself" set.
```

```
// in localhost cluster: set a context named default/remotehost/myself
for remotehost cluster
# kubectl config set-context default/remotehost/myself --user=remotehost/
myself --namespace=default --cluster=remotehost
context "default/remotehost/myself" set.
```

Let's check out the current view. We can see a list of contexts is in the contexts section now:

```
# kubectl config view
apiVersion: v1
clusters:
- cluster:
    insecure-skip-tls-verify: true
    server: http://localhost:8080
  name: localhost
- cluster:
    insecure-skip-tls-verify: true
    server: http://remotehost:8080
  name: remotehost
contexts:
- context:
    cluster: localhost
    namespace: default
    user: localhost/myself
  name: default/localhost/myself
- context:
    cluster: remotehost
    namespace: default
    user: remotehost/myself
  name: default/remotehost/myself
current-context: ""
kind: Config
preferences: {}
users:
- name: localhost/myself
  user:
    password: passwordlocal
    username: userlocal
```

```
- name: remotehost/myself
  user:
    password: passwordremote
    username: userremote
```

After creating contexts, let's start to switch context in order to manage different clusters. Here, we will use the command `kubectl config use-context <context nickname>`. We'll start from the localhost one first:

```
// in localhost cluster: use the context default/localhost/myself
# kubectl config use-context default/localhost/myself
switched to context "default/localhost/myself".
```

Let's list pods to see whether it is a localhost cluster:

```
// list the pods
# kubectl get pods
NAME                READY     STATUS        RESTARTS    AGE
localnginx-1blru    1/1       Running       0           1m
localnginx-p6cyo    1/1       Running       0           1m
```

Yes, it looks fine. How about if we switch to the context with the `remotehost` setting?

```
// in localhost cluster: switch to the context default/remotehost/myself
# kubectl config use-context default/remotehost/myself
switched to context "default/remotehost/myself".
Let's list the pods to make sure it's under the remotehost context:
# kubectl get pods
NAME                 READY     STATUS     RESTARTS    AGE
remotenginx-6wz5c    1/1       Running    0           1m
remotenginx-7v5in    1/1       Running    0           1m
remotenginx-c7go6    1/1       Running    0           1m
remotenginx-r1mf6    1/1       Running    0           1m
```

All the operations we have done are in the localhost cluster. kubeconfig makes switching multiple clusters with multiple users easier.

Cleaning up kubeconfig

The kubeconfig file is stored in `$HOME/.kube/config`. If the file is deleted, the configuration is gone; if the file is restored to the directory, the configuration will be restored:

```
// clean up kubeconfig file
# rm -f ~/.kube/config

// check out current view
# kubectl config view
apiVersion: v1
clusters: []
contexts: []
current-context: ""
kind: Config
preferences: {}
users: []
```

See also

kubeconfig manages the setting of clusters, credentials, and namespaces. Check out the following recipes:

- The *Working with namespaces* recipe in *Chapter 2*, *Walking through Kubernetes Concepts*
- *Authentication and authorization*

Setting resource in nodes

Computing resource management is so important in any infrastructure. We should know our application well and preserve enough CPU and memory capacity in order to prevent running out of resources. In this section, we'll introduce how to manage node capacity in the Kubernetes nodes. Furthermore, we'll also describe how to manage pod computing resources.

Getting ready

Before you start managing computing resources, you should know your applications well in order to know the maximum resources they need. Before we start, check out the current node capacity using the `kubectl` command described in *Chapter 1*, *Building Your Own Kubernetes*:

```
// check current node capacity
# kubectl get nodes -o json | jq '.items[] | {name: .metadata.name,
capacity: .status.capacity}'
{
  "name": "kube-node1",
  "capacity": {
    "cpu": "1",
    "memory": "1019428Ki",
    "pods": "40"
  }
}
{
  "name": "kube-node2",
  "capacity": {
    "cpu": "1",
    "memory": "1019428Ki",
    "pods": "40"
  }
}
```

You should know currently, we have two nodes with `1` CPU and `1019428` bytes memory. The node capacity of the pods are `40` for each. Then, we can start planning. How much computing resource capacity is allowed to be used on a node? How much computing resource is used in running our containers?

How to do it…

When the Kubernetes scheduler schedules a pod running on a node, it will always ensure that the total limits of the containers are less than the node capacity. If a node runs out of resources, Kubernetes will not schedule any new containers running on it. If no node is available when you launch a pod, the pod will remain pending, since the Kubernetes scheduler will be unable to find any node that could run your desired pod.

Managing node capacity

Sometimes, we want to explicitly preserve some resources for other processes or future usage on the node. Let's say we want to preserve 200 MB on all my nodes. First, we'll need to create a pod and run the `pause` container in Kubernetes. Pause is a container for each pod for forwarding the traffic. In this scenario, we'll create a resource reserver pod, which is basically doing nothing with a limit of 200 MB:

```
// configuration file for resource reserver
# cat /etc/kubernetes/reserve.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: resource-reserver
spec:
  containers:
  - name: resource-reserver
    image: gcr.io/google_containers/pause:0.8.0
    resources:
      limits:
        memory: 200Mi
```

Since it's a pod infra container, we will not use kubectl to launch it. Note that we put it in the `/etc/kubernetes/` folder. You could put it under different paths; write down the path and we'll need to add it into the kubelet config file to launch it. Find the kubelet config file you specified in the *Configuring nodes* recipe in *Chapter 1*, *Building Your Own Kubernetes* and add the following argument when launching kubelet: `--config=/etc/kubernetes/reserve.yaml`. Restart kubelet. After we restart kubelet, we will see the kubelet log in the node:

```
I0325 20:44:22.937067   21306 kubelet.go:1960] Starting kubelet main
sync loop.
I0325 20:44:22.937078   21306 kubelet.go:2012] SyncLoop (ADD):
"resource-reserver-kube-node1_default"
I0325 20:44:22.937138   21306 kubelet.go:2012] SyncLoop (ADD):
"mynginx-e09bu_default"
I0325 20:44:22.963425   21306 kubelet.go:2012] SyncLoop (ADD):
"resource-reserver-kube-node1_default"
I0325 20:44:22.964484   21306 manager.go:1707] Need to restart pod
infra container for "resource-reserver-kube-node1_default" because it
is not found
```

kubelet will check whether the infra container exists and create it accordingly:

```
// check pods list
# kubectl get pods
NAME                                 READY     STATUS    RESTARTS   AGE
resource-reserver-kube-node1   1/1       Running   0          3m
```

The Kubernetes master is aware that the resource reserver pod has been created. Let's describe the details to get deeper insight:

```
# kubectl describe pods resource-reserver-kube-node1
Name:         resource-reserver-kube-node1
Namespace:      default
```

```
Image(s):      gcr.io/google_containers/pause:0.8.0
Node:        kube-node1/10.0.0.224
Start Time:      Fri, 25 Mar 2016 20:44:24 +0000
Labels:        <none>
Status:        Running
IP:        192.168.99.3
Replication Controllers:  <none>
Containers:
  resource-reserver:
    ...
    QoS Tier:
      memory:  Guaranteed
    Limits:
      memory:  200Mi
    Requests:
      memory:    200Mi
    State:    Running
      Started:    Fri, 25 Mar 2016 20:44:24 +0000
    Ready:    True
```

We can find the limits and requests that are all set as `200Mi`; it means that this container has been reserved a minimum of 200 MB and a maximum of 200 MB. Repeat the same steps in your other nodes and check the status via the `kubectl` command:

```
# kubectl get pods
NAME                              READY     STATUS    RESTARTS   AGE
resource-reserver-kube-node1   1/1       Running   0          11m
resource-reserver-kube-node2   1/1       Running   0          42m
```

Limits or requests?

The Kubernetes scheduler schedules a pod running on a node by checking the remaining computing resources. We could specify the limits or requests for each pod we launch. Limit means the maximum resources this pod can occupy. Request means the minimum resources this pod needs. We could use the following inequality to represent their relation: 0 <= request <= the resource this pod occupies <= limit <= node capacity.

Managing computing resources in a pod

The concept for managing the capacity in a pod or node is similar. They both specify the requests or limits under the container resource spec.

Let's create an nginx pod with certain requests and limits using `kubectl create -f nginx-resources.yaml` to launch it:

```
# cat nginx-resources.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
    resources:
      requests:
        cpu: 250m
        memory: 32Mi
      limits:
        cpu: 500m
        memory: 64Mi

// create the pod
# kubectl create -f nginx-resources.yaml
pod "nginx" created
```

Following are the available resources for this pod:

- CPU: 250 milli core ~ 500 milli core
- Memory: 32MB ~ 64 MB

Please note that the minimum CPU limit is set to 10 millicore. You cannot specify a value less than the minimum limit. Let's get more details via kubectl:

```
# kubectl describe pod nginx
Name:         nginx
Namespace:      default
Image(s):      nginx
Node:         kube-node1/10.0.0.224
Start Time:      Fri, 25 Mar 2016 21:12:43 +0000
Labels:         name=nginx
Status:         Running
Reason:
Message:
IP:        192.168.99.4
Replication Controllers:  <none>
Containers:
  nginx:
    ...
    QoS Tier:
      cpu:  Burstable
      memory:  Burstable
    Limits:
      memory:  64Mi
      cpu:  500m
    Requests:
      cpu:    250m
      memory:    32Mi
    State:    Running
      Started:    Fri, 25 Mar 2016 21:12:44 +0000
    Ready:    True
    Restart Count:  0
```

Everything is expected. QoS Tier is `Burstable`. Compared with `Guaranteed`, `Burstable` has a buffer to burst to the limits; however, `Guaranteed` will always reserve certain resources for the pod. Please note that if you specify too many pods with `Guaranteed`, cluster utilization would be poor, since it wastes the resources if the containers are not reaching the limits all the time.

See also

In this section, you learned how to constrain computing resources in Kubernetes. We give more control to our containers. Check out the following recipes:

- The *Preparing your environment* and *Configuring nodes* recipes in *Chapter 1*, *Building Your Own Kubernetes*
- The *Working with namespaces* recipe in *Chapter 2*, *Walking through Kubernetes Concepts*
- The *Working with configuration files* recipe in *Chapter 3*, *Playing with Containers*
- The *Monitoring master and node* recipe in *Chapter 8*, *Logging and Monitoring*

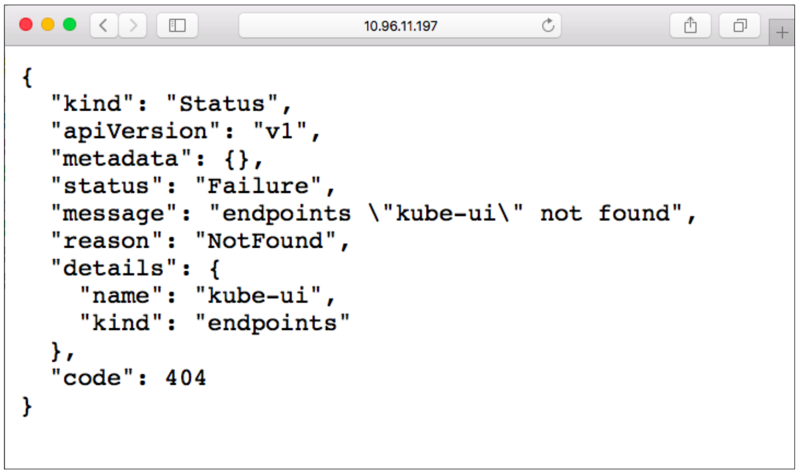
Playing with WebUI

Kubernetes has the WebUI add-on that visualizes Kubernetes' status, such as pod, replication controller, and service.

Getting ready

The Kubernetes WebUI is assigned as `http://<kubernetes master>/ui`. However, it is not launched by default, instead there are YAML files in the release binary.

> Kubernetes 1.2 introduces the dashboard. For more details, please refer to `http://kubernetes.io/docs/user-guide/ui/`.



```
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {},
  "status": "Failure",
  "message": "endpoints \"kube-ui\" not found",
  "reason": "NotFound",
  "details": {
    "name": "kube-ui",
    "kind": "endpoints"
  },
  "code": 404
}
```

Access to Kubernetes master/ui page

Let's download a release binary and launch the WebUI:

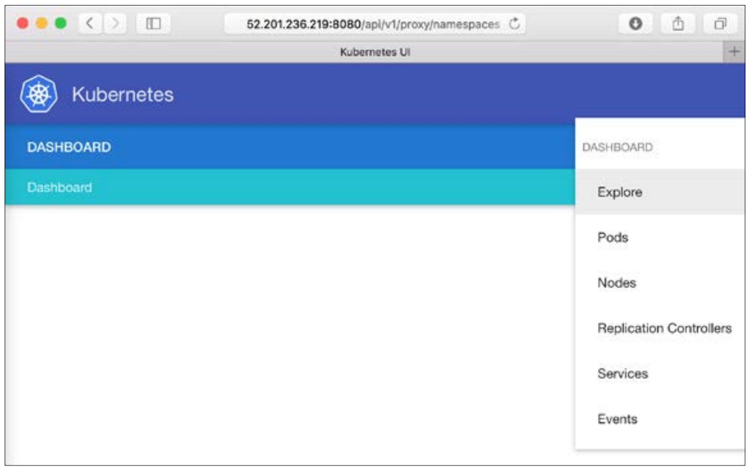
```
//Download a release binary
$ curl -L -O https://github.com/kubernetes/kubernetes/releases/download/
v1.1.4/kubernetes.tar.gz

//extract the binary
$ tar zxf kubernetes.tar.gz
//WebUI YAML file is under the cluster/addons/kube-ui directory
$ cd kubernetes/cluster/addons/kube-ui/
$ ls
kube-ui-rc.yaml    kube-ui-svc.yaml
```

How to do it...

Let's launch the kube-ui replication controller and service:

```
# kubectl create -f kube-ui-rc.yaml
replicationcontroller "kube-ui-v2" created

# kubectl create -f kube-ui-svc.yaml
service "kube-ui" created
```

Note that `kube-ui-svc` is a type of ClusterIP service; however, it is associated with Kubernetes master (/ui). You can access, from the outside, the Kubernetes network at `http://<kubernetes master>/ui`.

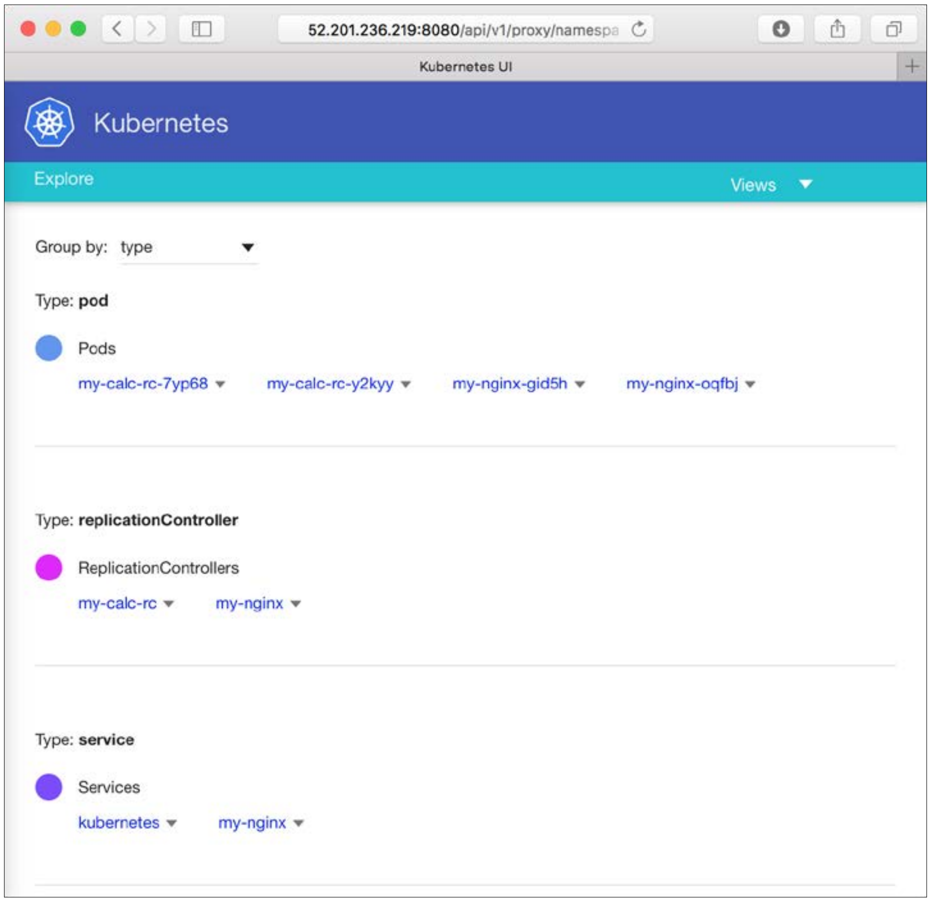


Launching the kube-ui/ui shows the dashboard screen

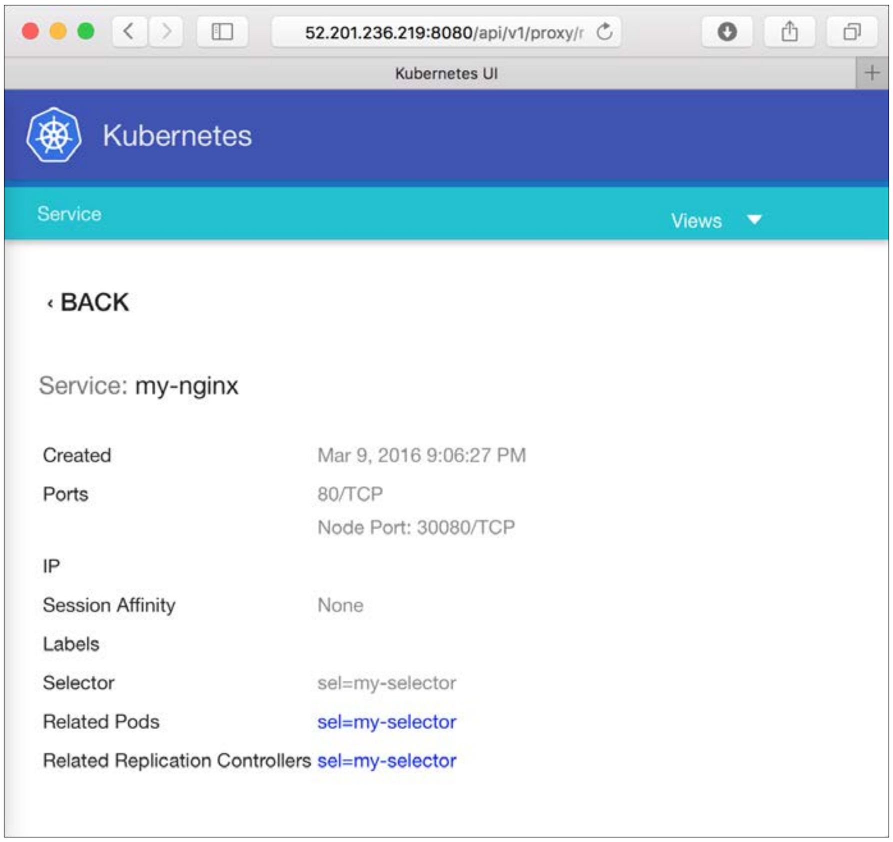
How it works...

The kube-ui replication controller accesses the API Server to get the Kubernetes cluster information same as the `kubectl` command, though read-only. However, it is useful for navigating Kubernetes' status and is easier to explore than the `kubectl` command.

The following screenshot is an **Explore** page that shows pod, replication controller, and service instances:



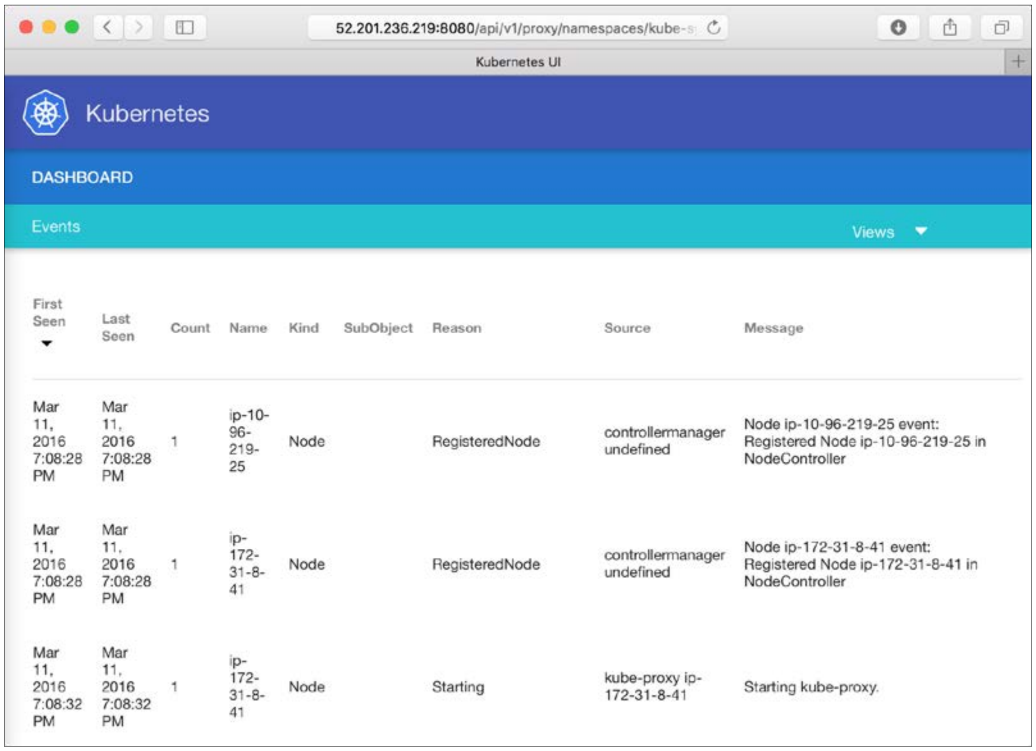
On clicking on one of the instances, it shows detailed information, as shown in the following screenshot. It shows a service, which indicates port, node port, and selectors. It is easy to find an associated replication controller and pods:



Service: my-nginx

Created	Mar 9, 2016 9:06:27 PM
Ports	80/TCP Node Port: 30080/TCP
IP	
Session Affinity	None
Labels	
Selector	sel=my-selector
Related Pods	sel=my-selector
Related Replication Controllers	sel=my-selector

Additionally, UI can also show events, as follows:



First Seen	Last Seen	Count	Name	Kind	SubObject	Reason	Source	Message
Mar 11, 2016 7:08:28 PM	Mar 11, 2016 7:08:28 PM	1	ip-10-96-219-25	Node		RegisteredNode	controllermanager undefined	Node ip-10-96-219-25 event: Registered Node ip-10-96-219-25 in NodeController
Mar 11, 2016 7:08:28 PM	Mar 11, 2016 7:08:28 PM	1	ip-172-31-8-41	Node		RegisteredNode	controllermanager undefined	Node ip-172-31-8-41 event: Registered Node ip-172-31-8-41 in NodeController
Mar 11, 2016 7:08:32 PM	Mar 11, 2016 7:08:32 PM	1	ip-172-31-8-41	Node		Starting	kube-proxy ip-172-31-8-41	Starting kube-proxy.

See also

This recipe described how to launch a web interface that will help in easily exploring Kubernetes instances, such as pods, replication controllers, and services without the `kubectl` command. Please refer to following recipes on how to get detailed information via the `kubectl` command.

- The *Working with pods*, *Working with a replication controller*, and *Working with services* recipes in *Chapter 2*, *Walking through Kubernetes Concepts*

Working with a RESTful API

The Kubernetes administrator can control the Kubernetes cluster via the `kubectl` command; it supports local and remote execution. However, some of the administrators or operators may need to integrate a program to control the Kubernetes cluster.

Kubernetes has a RESTful API that allows controlling the Kubernetes cluster via API similar to the `kubectl` command.

Getting ready

The RESTful API is open by default when we launch the API Server; you may access the RESTful API via the `curl` command, as follows:

```
//assume API server is running at localhost port number 8080
# curl http://localhost:8080/api
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ]
}
```

How to do it...

Let's create a replication controller using the following JSON format:

```
# cat nginx-rc.json
{
    "apiVersion": "v1",
    "kind": "ReplicationController",
    "metadata": {
        "name": "my-first-rc"
    },
    "spec": {
        "replicas": 2,
        "template": {
            "spec": {
                "containers": [
                    {
                        "image": "nginx",
                        "name": "my-nginx"
                    }
                ]
            },
            "metadata": {
                "labels": {
                    "app": "nginx"
                }
            }
        },
        "selector": {
            "app": "nginx"
        }
    }
}
```

Submit a request to create a replication controller, as follows:

```
# curl -XPOST -H "Content-type: application/json" -d @nginx-rc.json
http://localhost:8080/api/v1/namespaces/default/replicationcontrollers
```

Then, `kubectl get rc` command should be as follows:

```
# kubectl get rc
CONTROLLER     CONTAINER(S)    IMAGE(S)    SELECTOR     REPLICAS    AGE
my-first-rc    my-nginx        nginx       app=nginx    2           10s
```

Of course, use the `curl` command to check via a RESTful API, as follows:

```
# curl -XGET http://localhost:8080/api/v1/namespaces/default/
replicationcontrollers
```

Deletion can also be done via a RESTful API, as follows:

```
# curl -XDELETE http://localhost:8080/api/v1/namespaces/default/
replicationcontrollers/my-first-rc
```

Let's write the program that performs the same process. Following is the Python 2.7 code that creates the same replication controller:

```
# cat nginx-rc.py

import httplib
import json
k8s_master_url = "localhost"
k8s_master_port = 8080
namespace="default"

headers = {"Content-type": "applicaiton/json"}
```

```
rc = {}
rc["apiVersion"] = "v1"
rc["kind"] = "ReplicationController"
rc["metadata"] = {"name" : "my-second-rc"}
rc["spec"] = {
    "replicas": 2,
    "selector": {"app": "nginx"},
    "template": {
        "metadata": {"labels": {"app": "nginx"}},
        "spec": {
            "containers" :[
                {"name": "my-nginx", "image": "nginx"}
            ]
        }
    }
}

h1 = httplib.HTTPConnection(k8s_master_url, k8s_master_port)
h1.request("POST", "/api/v1/namespaces/%s/replicationcontrollers" %
namespace, json.dumps(rc), headers)
res = h1.getresponse()

print "return code = %d" % res.status
```

You can run this code using the Python interpreter, as follows:

```
# python nginx-rc.py
return code = 201

//HTTP return code 201 meant "Created"
```

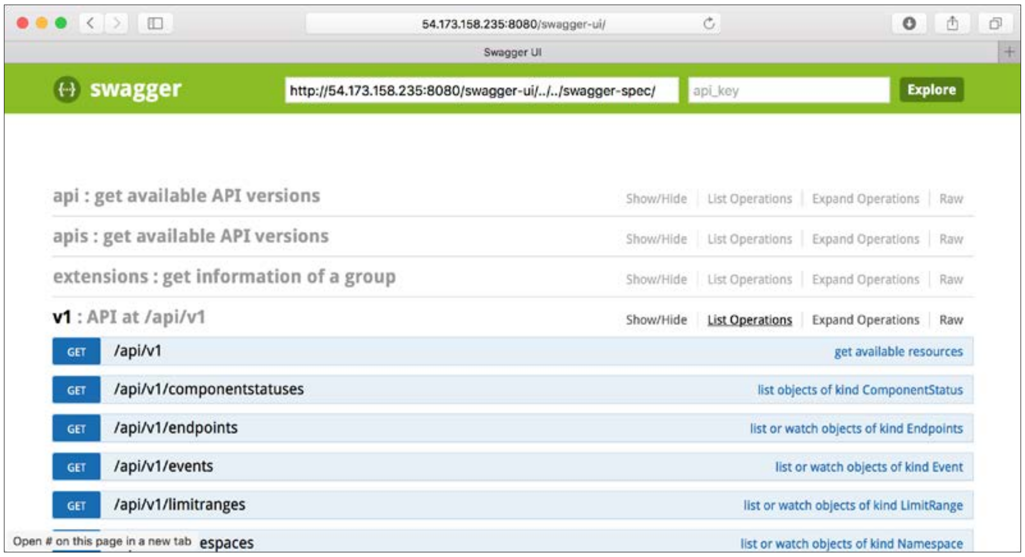
How it works...

The RESTful API allows the CRUD (Create, Read, Update, and Delete) operations, which are the same concepts behind every modern web application. For more details, please refer to `https://en.wikipedia.org/wiki/Create,_read,_update_and_delete`.

Kubernetes RESTful API examples and related HTTP methods are as follows:

Operation	**HTTP Method**	**Example**
Create	`POST`	`POST /api/v1/namespaces/default/services`
Read	`GET`	`GET /api/v1/componentstatuses`
Update	`PUT`	`PUT /api/v1/namespaces/default/replicationcontrollers/my-first-rc`
Delete	`DELETE`	`DELETE /api/v1/namespaces/default/pods/my-nginx`

The entire Kubernetes RESTful APIs are defined by Swagger (`http://swagger.io/`). You can see a detailed description via `http://<API Server IP Address>:<API Server port>/swagger-ui`.



54.173.158.235:8080/swagger-ui/

Swagger UI

swagger http://54.173.158.235:8080/swagger-ui/../../swagger-spec/ api_key Explore

api : get available API versions — Show/Hide | List Operations | Expand Operations | Raw

apis : get available API versions — Show/Hide | List Operations | Expand Operations | Raw

extensions : get information of a group — Show/Hide | List Operations | Expand Operations | Raw

v1 : API at /api/v1 — Show/Hide | List Operations | Expand Operations | Raw

GET	/api/v1	get available resources
GET	/api/v1/componentstatuses	list objects of kind ComponentStatus
GET	/api/v1/endpoints	list or watch objects of kind Endpoints
GET	/api/v1/events	list or watch objects of kind Event
GET	/api/v1/limitranges	list or watch objects of kind LimitRange
	espaces	list or watch objects of kind Namespace

Open # on this page in a new tab

See also

This recipe described how to use the Kubernetes RESTful API via a program. It is important to integrate with your automation program remotely. For detailed parameter and security enhancement, please refer to the following recipes:

- The *Working with configuration files* recipe in *Chapter 3*, *Playing with Containers*
- The *Authentication and authorization* recipe in *Chapter 7*, *Advanced Cluster administration*

Authentication and authorization

In order to use more advanced management, we can add permission rules to the Kubernetes system. Two permission types could be generated in our cluster: one is between the machines. Nodes having authentication can contact the controlling node. For example, the master who owns certification with the etcd server can store data in etcd. The other permission rule is inside the Kubernetes master. Users can be given authorization for checking and creating the resources. Applying authentication and authorization is a secure solution to prevent your data or status being accessed by others.

Getting ready

Before you start configuring your cluster with some permissions, please have your cluster installed. Nevertheless, stop every service in the system. They will be started later with the authentication enabled.

How to do it…

In this recipe, we will have a discussion on both authentication and authorization. For authentication, etcd and the Kubernetes server need to do identity verification before they respond to the requests. On the other hand, authorization restricts users by different resource access permissions. All of these contacts are based on the API connection. Later sections show you how to complete the configuration and follow the authentication.

Enabling authentication for an API call

There are several methods to block the unauthenticated communication in the Kubernetes system. We are going to introduce basic authentication mechanism. It is easier to set it on not only the Kubernetes masters, but etcd servers.

Basic authentication of etcd

First, let's try to send API requests on the etcd host. You will find that anyone can access the data by default:

```
// Create a key-value pair in etcd
# curl -X PUT -d value="Happy coding" http://localhost:4001/v2/keys/
message
{"action":"set","node":{"key":"/message","value":"Happy coding","modified
Index":4,"createdIndex":4}}
// Check the value you just push
# curl http://localhost:4001/v2/keys/message
{"action":"get","node":{"key":"/message","value":"Happy coding","modified
Index":4,"createdIndex":4}}
```

```
// Remove the value
# curl -X DELETE http://localhost:4001/v2/keys/message
{"action":"delete","node":{"key":"/message","modifiedIndex":5,"createdInd
ex":4},"prevNode":{"key":"/message","value":"Happy coding","modifiedIndex
":4,"createdIndex":4}}
```

Without authentication, neither reading nor writing functions can be protected. The way to enable basic authentication of etcd is through the RESTful API as well. The procedure is as follows:

- Add a password for the admin account `root`
- Enable basic authentication
- Stop both read and write permissions of the guest account

Make sure the etcd service is running. We will transfer the preceding logics into the following commands:

```
// Send the API request for setup root account
# curl -X PUT -d "{\"user\":\"root\",\"password\":\"<YOUR_ETCD_PASSWD>\",
\"roles\":[\"root\"]}" http://localhost:4001/v2/auth/users/root
{"user":"root","roles":["root"]}
// Enable authentication
# curl -X PUT http://localhost:4001/v2/auth/enable
// Encode "USERACCOUNT:PASSWORD" string in base64 format, and record in a
value
# AUTHSTR=$(echo -n "root:<YOUR_ETCD_PASSWD>" | base64)
// Remove all permission of guest account. Since we already enable
authentication, use the authenticated root.
# curl -H "Authorization: Basic $AUTHSTR" -X PUT -d "{\"role\":\"guest
\",\"revoke\":{\"kv\":{\"read\":[\"*\"],\"write\":[\"*\"]}}}" http://
localhost:4001/v2/auth/roles/guest
{"role":"guest","permissions":{"kv":{"read":[],"write":[]}}}
```

Now, for validation, try to check anything in etcd through the API:

```
# curl http://localhost:4001/v2/keys
{"message":"Insufficient credentials"}
```

Because we didn't specify any identity for this API call, it is regarded as a request from a guest. No authorization for even viewing data.

For long-term usage, we would put user profiles of etcd in the Kubernetes master's configuration. Check your configuration file of the Kubernetes API server. In the RHEL server, it is the file `/etc/kubernetes/apiserver`; or for the other Linux server, just the one for service, `/etc/init.d/kubernetes-master`. You can find a flag for the etcd server called `--etcd-servers`. Based on the previous settings, the value we attached is a simple URL with a port. It could be `http://ETCD_ELB_URL:80`. Add an account root and its password in a plain text format to the value, which is the authorization header for the HTTP request. The new value for flag `--etcd-servers` would become `http://root:YOUR_ETCD_PASSWD@ETCD_ELB_URL:80`. Afterwards, your Kubernetes API server daemon will work well with an authentication-enabled etcd endpoint.

Basic authentication of the Kubernetes master

Before we set up authentication in the Kubernetes master, let's check the endpoint of the master first:

```
# curl https://K8S_MASTER_HOST_IP:SECURED_PORT --insecure
```

or

```
# curl http://K8S_MASTER_ELB_URL:80
{
  "paths": [
    "/api",
    "/api/v1",
    "/apis",
    "/apis/extensions",
    "/apis/extensions/v1beta1",
    "/healthz",
    "/healthz/ping",
    "/logs/",
    "/metrics",
    "/resetMetrics",
    "/swagger-ui/",
    "/swaggerapi/",
    "/ui/",
    "/version"
  ]
}
```

We don't want the previous message to be exposed to everyone, do we? Same as the etcd host, Kubernetes master can apply basic authentication as a security mechanism. In this section, we are going to limit the permission of the API server. However, different from etcd, the information of authentication is defined in a file:

```
// Create a file for basic authentication with content:
PASSWORD,USERNAME,UID
# cat /root/k8s-bafile
<APISERVER_BA_PASSWORD>,<APISERVER_BA_USERACCOUNT>,1
```

Then, we have to specify this file in the configuration file. According to your daemon management tool, the configuration file could be located at `/etc/init.d/kubernetes-master` or `/etc/kubernetes/apiserver`. A new flag named `--basic-auth-file` should be added to the configuration file:

- For the file `kubernetes-master`, append flag `--basic-auth-file` after the `kube-apiserver` command or the `hyperkube apiserver` command. The value for this tag should be the full path of the basic authentication file. For instance, `--basic-auth-file=/root/k8s-bafile`.
- For the file `apiserver`, add the tag to the variable `KUBE_API_ARGS`. For example, `KUBE_API_ARGS=--basic-auth-file=/root/k8s-bafile`.

Most important of all, ensure the user who starts the services of Kubernetes, either root or kubelet, has the permission to access the file you attached to the tag. After you add the new tag, it is necessary to restart the service for making the authentication effective. Next, it is good for you to try the `curl` command at the beginning of this section. It will return `Unauthorized` without providing the username and password.

Our nodes communicate with API server through the insecure port `8080`. Although we didn't have to specify any role for authorizing permission, be aware of configuring the firewall of master, which only allows nodes to go through port `8080`. On AWS, a security group can help for this part.

There are still some methods for the Kubernetes master's authentication. Please check the official website for other ideas (`http://kubernetes.io/docs/admin/authentication/`).

Making use of user authorization

We can also add different user permissions for the Kubernetes master's API server daemon. There are three flags required to set user authorization. They are as follows:

- `--authorization-mode=ABAC`: The value, ABAC, is the abbreviation of Attribute-Based Access Control. By enabling this mode, we can set up customized user permissions.

- `--token-auth-file=<FULL_PATH_OF_YOUR_TOKEN_FILE>`: This is the file we used to announce the qualified users for API access. It is possible to provide more accounts and token pairs.
- `--authorization-policy-file=<FULL_PATH_OF_YOUR_POLICY_FILE>`: We would need this policy file to generate separated rules for different users.

These special tags are going to be appended after the command `kube-apiserver` or `hyperkube apiserver`. You can refer to the following example:

```
// The daemon configuration file of Kubernetes master for init.d service
# cat /etc/init.d/kubernetes-master
(above lines are ignored)
:
# Start daemon.
    echo $"Starting apiserver: "
    daemon $apiserver_prog \
    --service-cluster-ip-range=${CLUSTER_IP_RANGE} \
    --insecure-port=8080 \
    --secure-port=6443 \
    --authorization-mode=ABAC \
    --token-auth-file=/root/k8s-tokenfile \
    --authorization-policy-file=/root/k8s-policyfile \
    --address=0.0.0.0 \
    --etcd-servers=${ETCD_SERVERS} \
    --cluster-name=${CLUSTER_NAME} \
    > ${logfile}-apiserver.log 2>&1 &
:
(below lines are ignored)
```

or

```
// The kubernetes apiserver's configuration file for systemd service in
RHEL
# cat /etc/kubernetes/apiserver
(above lines are ignored)
:
KUBE_API_ARGS="--authorization-mode=ABAC --token-auth-file=/root/k8s-
tokenfile --authorization-policy-file=/root/k8s-policyfile"
```

You still need to configure the file for account and the file for policy. To demonstrate the usage of customizing user permission, the following content of files show you how to create an admin account with full access and a read-only account:

```
# cat /root/k8s-tokenfile
k8s2016,admin,1
happy123,amy,2
```

The format of user definition is similar to the basic authentication file we mentioned before. Each line has these items in sequence: token, username, and UID. Other than admin, we create another user account called `amy`, which will only have read-only permission:

```
# cat /root/k8s-policyfile
{"apiVersion": "abac.authorization.kubernetes.io/v1beta1", "kind":
"Policy", "spec": {"user": "admin", "namespace": "*", "resource": "*",
"apiGroup": "*"}}
{"apiVersion": "abac.authorization.kubernetes.io/v1beta1", "kind":
"Policy", "spec": {"user": "amy", "namespace": "*", "resource": "*",
"readonly": true}}
```

For the policy file, each line will be a policy in the JSON format. Every policy should indicate the user who obeys its rule. The first policy for admin allows control permission on every namespace, resource, and API group. The key `apiGroup` specifies different API categories. For instance, the resource `job` is defined in the `extensions` API group. In order to access `job`, the `extensions` type should be included in `apiGroup`. The second policy is defined with read-only permission, which means the role `amy` can only view resources, but not create, remove, and edit actions.

Later, restart all the daemons of the Kubernetes master after you make both configuration files and service files ready:

```
// For init.d service management
# service kubernetes-master restart
// Or, you can restart the individually with dependency
# systemctl stop kube-scheduler
# systemctl stop kube-controller-manager
# systemctl stop kube-apiserver
# systemctl start kube-apiserver
# systemctl start kube-controller-manager
# systemctl start kube-scheduler
```

See also

It is recommended to read some of the previous recipes on installing the Kubernetes cluster:

- The *Building datastore*, *Configuring master* and *Configuring nodes* recipe in *Chapter 1*, *Building Your Own Kubernetes*
- The *Auto-deploying Kubernetes through Chef recipes* recipe in *Chapter 6*, *Building Kubernetes on AWS*

8 Logging and Monitoring

In this chapter, we will cover the following topics:

- Collecting application logs
- Working with Kubernetes logs
- Working with etcd log
- Monitoring master and node

Introduction

As DevOps, logging and monitoring are what we always keep in mind. These tell us the stability and the status of our systems. For taking care of logs, you will learn how to collect the application logs inside Kubernetes. You will also learn how to collect and inspect the logs for Kubernetes. Finally, we will go through setting up monitoring systems for Kubernetes.

Collecting application logs

When you start managing the application, the log collection and analysis are two of the important routines to keep tracking the application's status.

However, there are some difficulties when the application is managed by Docker/Kubernetes; because the log files are inside the container, it is not easy to access them from outside the container. In addition, if the application has many pods by the replication controller, it will also be difficult to trace or identify in which pod the issue that has happened.

One way to overcome this difficulty is to prepare a centralized log collection platform that accumulates and preserves the application log. This recipe describes one of the popular log collection platforms **ELK** (**Elasticsearch**, **Logstash**, and **Kibana**).

Getting ready

First, we need to prepare the Elasticsearch server at the beginning. Then, the application will send a log to Elasticsearch using Logstash. We will visualize the analysis result using Kibana.

Elasticsearch

Elasticsearch (`https://www.elastic.co/products/elasticsearch`) is one of the popular text indexes and analytic engines. There are some examples YAML files that are provided by the Kubernetes source file; let's download it using the `curl` command to set up Elasticsearch:

[An example YAML file is located on GitHub at `https://github.com/kubernetes/kubernetes/tree/master/examples/elasticsearch`.]

```
# curl -L -O https://github.com/kubernetes/kubernetes/releases/download/
v1.1.4/kubernetes.tar.gz
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed
100   593    0   593    0     0   1798      0 --:--:-- --:--:-- --:--:-
-  1802
100  181M  100  181M    0     0  64.4M      0  0:00:02  0:00:02 --:--:--
75.5M
# tar zxf kubernetes.tar.gz
# cd kubernetes/examples/elasticsearch/
# ls
es-rc.yaml  es-svc.yaml  production_cluster  README.md  service-account.
yaml
```

Create ServiceAccount (`service-account.yaml`) and then create the Elasticsearch replication controller (`es-rc.yaml`) and service (`es-svc.yaml`) as follows:

```
# kubectl create -f service-account.yaml
serviceaccount "elasticsearch" created

//As of Kubernetes 1.1.4, it causes validation error
//therefore append --validate=false option
```

```
# kubectl create -f es-rc.yaml --validate=false
replicationcontroller "es" created

# kubectl create -f es-svc.yaml
service "elasticsearch" created
```

Then, you can access the Elasticsearch interface via the Kubernetes service as follows:

```
//Elasticsearch is open by 192.168.45.152 in this example
# kubectl get service
NAME            CLUSTER_IP       EXTERNAL_IP   PORT(S)
SELECTOR                  AGE
elasticsearch   192.168.45.152                 9200/TCP,9300/TCP
component=elasticsearch   9s
kubernetes      192.168.0.1      <none>        443/TCP             <none>
110d

//access to TCP port 9200
# curl http://192.168.45.152:9200/
{
  "status" : 200,
  "name" : "Wallflower",
  "cluster_name" : "myesdb",
  "version" : {
    "number" : "1.7.1",
    "build_hash" : "b88f43fc40b0bcd7f173a1f9ee2e97816de80b19",
    "build_timestamp" : "2015-07-29T09:54:16Z",
    "build_snapshot" : false,
    "lucene_version" : "4.10.4"
  },
  "tagline" : "You Know, for Search"
}
```

Now, get ready to send an application log to Elasticsearch.

How to do it...

Let's use a sample application, which was introduced in the *Moving monolithic to microservices* recipe in *Chapter 5*, *Building a Continuous Delivery Pipeline*. Prepare a Python Flask program as follows:

```
# cat entry.py

from flask import Flask, request
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"

@app.route("/addition/<int:x>/<int:y>")
def add(x, y):
    return "%d" % (x+y)

if __name__ == "__main__":
    app.run(host='0.0.0.0')
```

Use this application to send a log to Elasticsearch.

Logstash

Send an application log to Elasticsearch; using Logstash (`https://www.elastic.co/products/logstash`) is the easiest way, because it converts from a plain text format to the Elasticsearch (JSON) format.

Logstash needs a configuration file that specifies the Elasticsearch IP address and port number. In this recipe, Elasticsearch is managed by Kubernetes service; therefore, the IP address and port number can be found using the environment variable as follows:

Item	**Environment Variable**	**Example**
Elasticsearch IP address	`ELASTICSEARCH_SERVICE_HOST`	`192.168.45.152`
Elasticsearch port number	`ELASTICSEARCH_SERVICE_PORT`	`9200`

However, the Logstash configuration file doesn't support an environment variable directly. Therefore, the Logstash configuration file uses the placeholder as `_ES_IP_` and `_ES_PORT_` as follows:

```
# cat logstash.conf.temp

input {
    stdin {}
}

filter {
  grok {
    match => {
        "message" => "%{IPORHOST:clientip} %{HTTPDUSER:ident}
%{USER:auth} \[%{DATA:timestamp}\] \"(?:%{WORD:verb} %{NOTSPACE:request}
(?: HTTP/%{NUMBER:httpversion})?|%{DATA:rawrequest})\" %{NUMBER:response}
(?:%{NUMBER:bytes}|-)"
    }
  }
}

output {
  elasticsearch {
    hosts => ["_ES_IP_:_ES_PORT_"]
    index => "mycalc-access"
}

stdout { codec => rubydebug }
}
```

Startup script

The startup script will read an environment variable, and then replace the placeholder to set the real IP and port number, as follows:

```
#!/bin/sh

TEMPLATE="logstash.conf.temp"
LOGSTASH="logstash-2.2.2/bin/logstash"
```

```
cat $TEMPLATE | sed "s/_ES_IP_/$ELASTICSEARCH_SERVICE_HOST/g" | sed
"s/_ES_PORT_/$ELASTICSEARCH_SERVICE_PORT/g" > logstash.conf

python entry.py 2>&1 | $LOGSTASH -f logstash.conf
```

Dockerfile

Finally, prepare Dockerfile as follows to build a sample application:

```
FROM ubuntu:14.04

# Update packages
RUN apt-get update -y

# Install Python Setuptools
RUN apt-get install -y python-setuptools git telnet curl openjdk-7-jre

# Install pip
RUN easy_install pip

# Bundle app source
ADD . /src
WORKDIR /src

# Download LogStash
RUN curl -L -O https://download.elastic.co/logstash/logstash/logstash-
2.2.2.tar.gz

RUN tar -zxf logstash-2.2.2.tar.gz

# Add and install Python modules
RUN pip install Flask

# Expose
EXPOSE  5000

# Run
CMD ["./startup.sh"]
```

Docker build

Let's build a sample application using the `docker build` command:

```
# ls
Dockerfile  entry.py  logstash.conf.temp  startup.sh

# docker build -t hidetosaito/my-calc-elk .
Sending build context to Docker daemon  5.12 kB
Step 1 : FROM ubuntu:14.04
 ---> 1a094f2972de
Step 2 : RUN apt-get update -y
 ---> Using cache
 ---> 40ff7cc39c20
Step 3 : RUN apt-get install -y python-setuptools git telnet curl
openjdk-7-jre
 ---> Running in 72df97dcbb9a

(skip…)

Step 11 : CMD ./startup.sh
 ---> Running in 642de424ee7b
 ---> 09f693436005
Removing intermediate container 642de424ee7b
Successfully built 09f693436005

//upload to Docker Hub using your Docker account
# docker login
Username: hidetosaito
Password:
Email: hideto.saito@yahoo.com
WARNING: login credentials saved in /root/.docker/config.json
Login Succeeded

//push to Docker Hub
```

```
# docker push hidetosaito/my-calc-elk
The push refers to a repository [docker.io/hidetosaito/my-calc-elk] (len: 1)
09f693436005: Pushed
b4ea761f068a: Pushed

(skip…)

c3eb196f68a8: Image already exists
latest: digest: sha256:45c203d6c40398a988d250357f85f1b5ba7b14ae73d449b3ca64b562544cf1d2 size: 22268
```

Kubernetes replication controller and service

Now, use this application by Kubernetes to send a log to Elasticsearch. First, prepare the YAML file to load this application using the replication controller and service as follows:

```
# cat my-calc-elk.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-calc-elk-rc
spec:
  replicas: 2
  selector:
        app: my-calc-elk
  template:
    metadata:
      labels:
        app: my-calc-elk
    spec:
      containers:
      - name: my-calc-elk
        image: hidetosaito/my-calc-elk
---
apiVersion: v1
```

```
kind: Service
metadata:
  name: my-calc-elk-service

spec:
  ports:
    - protocol: TCP
      port: 5000
  type: ClusterIP
  selector:
     app: my-calc-elk
```

Use the `kubectl` command to create the replication controller and service as follows:

```
# kubectl create -f my-calc-elk.yaml
replicationcontroller "my-calc-elk-rc" created
service "my-calc-elk-service" created
```

Check the Kubernetes service to find an IP address for this application as follows. It indicates `192.168.121.63`:

```
# kubectl get service
NAME                   CLUSTER_IP        EXTERNAL_IP    PORT(S)
SELECTOR                   AGE
elasticsearch          192.168.101.143                  9200/TCP,9300/TCP
component=elasticsearch    15h
kubernetes             192.168.0.1       <none>         443/TCP
<none>                     19h
my-calc-elk-service    192.168.121.63    <none>         5000/TCP
app=my-calc-elk            39s
```

Let's access this application using the `curl` command as follows:

```
# curl http://192.168.121.63:5000/
Hello World!

# curl http://192.168.121.63:5000/addition/3/5
8
```

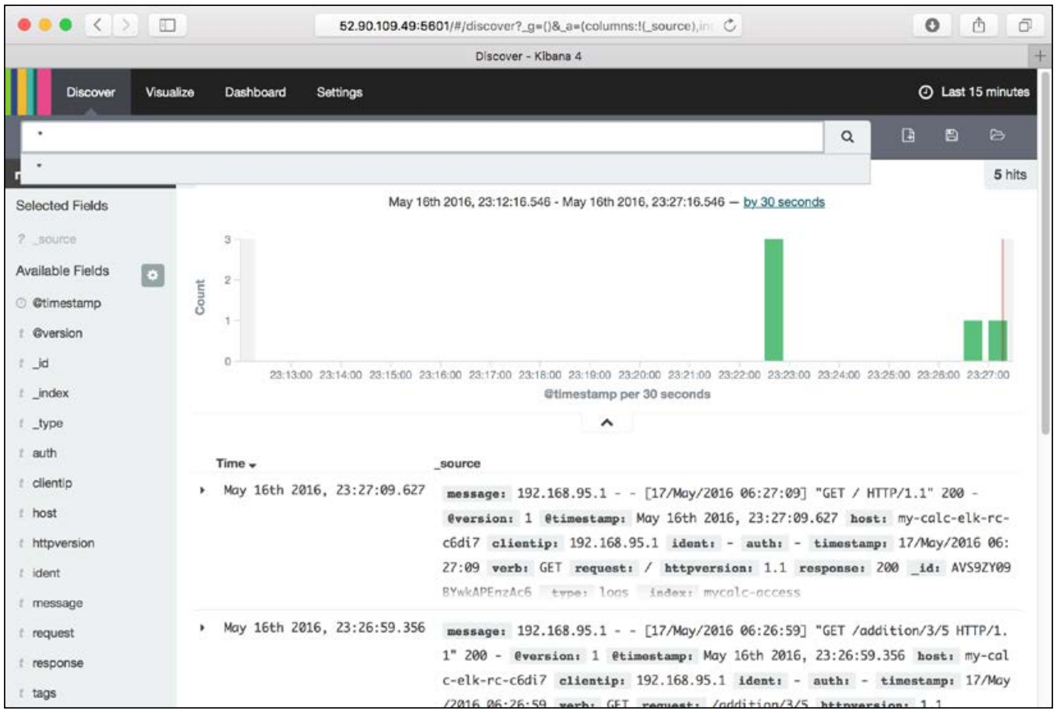
Kibana

Kibana (`https://www.elastic.co/products/kibana`) is a visualization tool for Elasticsearch. Download Kibana, and specify the Elasticsearch IP address, and port number, to launch Kibana:

```
//Download Kibana 4.1.6
# curl -O https://download.elastic.co/kibana/kibana/kibana-4.1.6-
linux-x64.tar.gz
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed
100 17.7M  100 17.7M    0     0  21.1M      0 --:--:-- --:--:-- --:--:--
21.1M


//unarchive
# tar -zxf kibana-4.1.6-linux-x64.tar.gz


//Find Elasticsearch IP address
# kubectl get services
NAME                   CLUSTER_IP        EXTERNAL_IP    PORT(S)
SELECTOR                   AGE
elasticsearch          192.168.101.143                  9200/TCP,9300/TCP
component=elasticsearch    19h
kubernetes             192.168.0.1       <none>         443/TCP
<none>                     23h


//specify Elasticsearch IP address
# sed -i -e "s/localhost/192.168.101.143/g" kibana-4.1.6-linux-x64/
config/kibana.yml


//launch Kibana
# kibana-4.1.6-linux-x64/bin/kibana
```

Then, you will see the application log. Create a chart as follows:

[This cookbook doesn't cover how to configure Kibana; please visit the official page to learn about the Kibana configuration via `https://www.elastic.co/products/kibana`.]



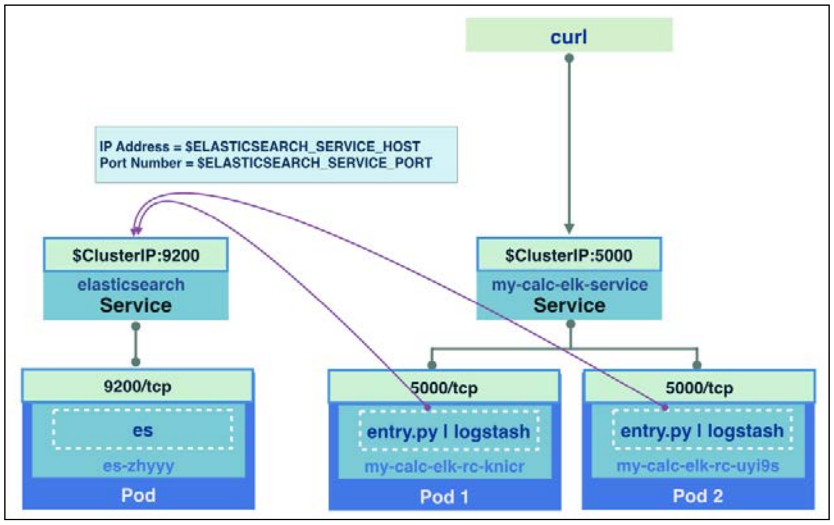
```
Time           _source
May 16th 2016, 23:27:09.627   message: 192.168.95.1 - - [17/May/2016 06:27:09] "GET / HTTP/1.1" 200 -
  @version: 1 @timestamp: May 16th 2016, 23:27:09.627 host: my-calc-elk-rc-
  c6di7 clientip: 192.168.95.1 ident: - auth: - timestamp: 17/May/2016 06:
  27:09 verb: GET request: / httpversion: 1.1 response: 200 _id: AVS9ZY09
  BYwkAPEnzAc6 _type: logs _index: mycalc-access
May 16th 2016, 23:26:59.356   message: 192.168.95.1 - - [17/May/2016 06:26:59] "GET /addition/3/5 HTTP/1.
  1" 200 - @version: 1 @timestamp: May 16th 2016, 23:26:59.356 host: my-cal
  c-elk-rc-c6di7 clientip: 192.168.95.1 ident: - auth: - timestamp: 17/May
  /2016 06:26:59 verb: GET request: /addition/3/5 httpversion: 1.1
```

How it works...

Now, the application log is captured by Logstash; it is converted into the JSON format and then sent to Elasticsearch.

Since Logstash is bundled with an application container, there are no problems when the replication controller increases the number of replicas (pods). It captures all the application logs with no configuration changes.

All the logs will be stored in Elasticsearch as follows:



See also

This recipe covers how to integrate to the ELK stack. The centralized log collection platform is important for the Kubernetes environment. The container is easy to launch, and is destroyed by the scheduler, but it is not easy to know which node runs which pod. Check out the following recipes:

- *Working with Kubernetes logs*
- *Working with etcd log*
- The *Moving monolithic to microservices* recipe in *Chapter 5*, *Building a Continuous Delivery Pipeline*

Working with Kubernetes logs

Kubernetes comes with three daemon processes on master: API server, scheduler, and controller manager. Under the `/var/log` folder, there are three corresponding log files recording the logs of these processes:

Daemon on master	**Log file**	**Description**
API server	`apiserver.log`	Logs for API calls.
Scheduler	`k8s-scheduler.log`	Logs of scheduler data for any containers scheduling events

Daemon on master	Log file	Description
Controller manager	`controller-manager.log`	Logs for showing any events or issues relate to controller manager

On nodes, we have a `kubelet` process to handle container operations and report to the master:

Daemon on node	Log file	Description
`kubelet`	`kubelet.log`	Logs for any issues happening in container

On both masters and nodes, there is another log file named `kube-proxy.log` to record any network connection issues.

Getting ready

We will use the log collection platform ELK, which was introduced in the previous section, to collect Kubernetes logs as a centralized log platform. For the setting of ELK, we'd suggest you to review the collecting application logs section again. Before we start collecting the Kubernetes logs, knowing the data structure in the logs is important. The preceding logs are this format:

```
<log level><date> <timestamp> <indicator> <source file>:<line
number>] <logs>
```

The following is an example:

```
E0328 00:46:50.870875    3189 reflector.go:227] pkg/proxy/config/api.
go:60: Failed to watch *api.Endpoints: too old resource version: 45128
(45135)
```

By the heading character of the lines in the log file, we are able to know the log severity of this line:

- D: DEBUG
- I: INFO
- W: WARN
- E: ERROR
- F: FATAL

How to do it...

We will still use the `grok` filter in the `logstash` setting, as discussed in the previous section, but we might need to write our custom pattern for the `<log level><date>` pattern, which is listed at the beginning of the log line. We will create a pattern file under the current directory:

```
// list custom patterns
# cat ./patterns/k8s
LOGLEVEL     [DEFIW]
DATE         [0-9]{4}
K8SLOGLEVEL %{LOGLEVEL:level}%{DATE}
```

The preceding setting is used to split the E0328 pattern into level=E and DATE=0328. The following is an example of how to send `k8s-apiserver.log` into the ElasticSearch cluster:

```
// list config file for k8s-apiserver.log in logstash
# cat apiserver.conf
input {
  file {
    path => "/var/log/k8s-apiserver.log"
  }
}

filter {
  grok {
    patterns_dir => ["./patterns"]
    match => { "message" => "%{K8SLOGLEVEL} %{TIME}    %{NUMBER}
%{PROG:program}:%{POSINT:line}] %{GREEDYDATA:message}" }
  }
}

output {
  elasticsearch {
    hosts => ["_ES_IP_:_ES_PORT_"]
    index => "k8s-apiserver"
  }

  stdout { codec => rubydebug }
}
```

For the input, we will use the file plugin (`https://www.elastic.co/guide/en/logstash/current/plugins-inputs-file.html`), which adds the path of the `k8s-apiserver.log`. We will use `patterns_dir` in `grok` to specify the definition of our custom patterns `K8SLOGLEVEL`. The hosts' configuration in the output `elasticsearch` section should be specified to your Elasticsearch IP and port number. The following is a sample output:

```
// start logstash with config apiserver.conf
# bin/logstash -f apiserver.conf
Settings: Default pipeline workers: 1
Pipeline main started
{
       "message" => [
        [0] "E0403 15:55:24.706498    2979 errors.go:62] apiserver
received an error that is not an unversioned.Status: too old resource
version: 47419 (47437)",
        [1] "apiserver received an error that is not an unversioned.
Status: too old resource version: 47419 (47437)"
    ],
 "@timestamp" => 2016-04-03T15:55:25.709Z,
         "level" => "E",
          "host" => "kube-master1",
       "program" => "errors.go",
          "path" => "/var/log/k8s-apiserver.log",
          "line" => "62",
      "@version" => "1"
}
{
       "message" => [
        [0] "E0403 15:55:24.706784    2979 errors.go:62] apiserver
received an error that is not an unversioned.Status: too old resource
version: 47419 (47437)",
        [1] "apiserver received an error that is not an unversioned.
Status: too old resource version: 47419 (47437)"
    ],
    "@timestamp" => 2016-04-03T15:55:25.711Z,
         "level" => "E",
          "host" => "kube-master1",
       "program" => "errors.go",
          "path" => "/var/log/k8s-apiserver.log",
          "line" => "62",
      "@version" => "1"
}
```

It shows the current host, the log path, log level, the triggered program, and the total message. The other logs are all in the same format, so it is easy to replicate the settings. Just specify different indexes from `k8s-apiserver` to the others. Then, you are free to search the logs via Kibana, or get the other tools integrated with Elasticsearch to get notifications or so on.

See also

Check out the following recipes:

- The *Configuring master* and *Configuring nodes* recipes in *Chapter 1*, *Building Your Own Kubernetes*
- *Collecting application logs*
- *Monitoring master and node*

Working with etcd log

The datastore, etcd, works for saving the information of Kubernetes resources. The Kubernetes system won't be stable without robust etcd servers. If the information of a pod is lost, we will not be able to recognize it in the system, mention how to access it through the Kubernetes service, or manage it through the replication controller. In this recipe, you will learn what kind of message you may get from the etcd log and how to collect them with ELK.

Getting ready

Before we start collecting the log of etcd, we should prepare the servers of ELK. Please go back to the *Collecting application logs* recipe in *Chapter 8*, *Logging and Monitoring* to review how to set up ELK and study its basic usage.

On the other hand, please expose your Kubernetes service of Elasticsearch if your etcd servers are individual machines beyond the Kubernetes cluster. You can modify the service template for Elasticsearch as follows:

```
# cat es-svc.yaml
apiVersion: v1
kind: Service
metadata:
  name: elasticsearch
  labels:
    component: elasticsearch
spec:
  type: LoadBalancer
  selector:
    component: elasticsearch
  ports:
  - name: http
    port: 9200
    nodePort: 30000
    protocol: TCP
  - name: transport
    port: 9300
    protocol: TCP
```

Then, your Logstash process on the etcd server can access it using the URL `<NODE_ENDPOINT>:30000`. Port `30000` is exposed on every node, which means that it is possible to be contacted by every nodes' endpoints.

How to do it...

On the etcd server, we have recorded the log from the daemon `etcd` at `/var/log/etcd.log`. The message is line by line in the following format:

```
<date> <time> <subpackage>: <logs>
```

It is quite straightforward to show the timestamp and the information. We can also see where the logs come from, which means we know which kind of subpackages handle this issue. Here is an example of an etcd log:

```
2016/04/4 08:43:51 etcdserver: starting server... [version: 2.1.1,
cluster version: to_be_decided]
```

After you understand the style of the message, it is time to create the Logstash configuration file:

```
# cat etcd.conf
input {
  file {
    path => "/var/log/etcd.log"
  }
}

filter {
  grok {
    match => {
        "message" => "%{DATA:date} %{TIME:time} %{PROG:subpackage}:
%{GREEDYDATA:message}"
    }
  }
}

output {
  elasticsearch {
    hosts => ["<ELASTIC_SERVICE_IP>:<EXPOSE_PORT>"]
    index => "etcd-log"
  }

  stdout { codec => rubydebug }
}
```

In the file, we will assign the location of the etcd logfile as input data. The pattern defined in the `grok` filter simply separates the log into three parts: timestamp, the subpackage, and the message. Of course, we not only show the output on screen, but also send the data to the Elasticsearch server for further analysis:

```
// Under the directory of $LOGSTASH_HOME
# ./bin/logstash -f etcd.conf
Settings: Default pipeline workers: 1
Pipeline main started
{
    "subpackage" => "raft",
       "message" => [
        [0] "2016/04/4 08:43:53 raft: raft.node: ce2a822cea30bfca elected
leader ce2a822cea30bfca at term 2",
        [1] "raft.node: ce2a822cea30bfca elected leader ce2a822cea30bfca
at term 2"
    ],
    "@timestamp" => 2016-04-04T11:23:
27.571Z,
          "time" => "08:43:53",
          "host" => "etcd1",
          "path" => "/var/log/etcd.log",
          "date" => "2016/04/4",
      "@version" => "1"
}
{
    "subpackage" => "etcdserver",
       "message" => [
        [0] "2016/04/4 08:43:53 etcdserver: setting up the initial
cluster version to 2.1.0",
        [1] "setting up the initial cluster version to 2.1.0"
    ],
    "@timestamp" => 2016-04-04T11:24:09.603Z,
          "time" => "08:43:53",
          "host" => "etcd1",
          "path" => "/var/log/etcd.log",
          "date" => "2016/04/4",
      "@version" => "1"
}
```

As you can see, through Logstash, we will parse the log in different subpackage issues and at a particular time. It is a good time for you now to access the Kibana dashboard, and work with the etcd logs.

See also

Before you trace the log file of etcd, you must have your own etcd system. Take a look at the previous chapters; you will understand how to build a single node or a cluster-like etcd system. Check out the following recipes:

- The *Building datastore* recipe in *Chapter 1*, *Building Your Own Kubernetes*
- The *Clustering etcd* recipe in *Chapter 4*, *Building a High Availability Cluster*
- *Collecting application logs*
- *Working with Kubernetes logs*

Monitoring master and node

During the journey of the previous recipes, you learned how to build your own cluster, run various resources, enjoy different scenarios of usages, and even enhance cluster administration. Now comes a new level of view for your Kubernetes cluster. In this recipe, we are going to talk about monitoring. Through the monitoring tool, users not only learn about the resource consumption of workers and nodes, but also the pods. It will help us have a better efficiency in resource utilization.

Getting ready

Before we set up our monitoring cluster in the Kubernetes system, there are two main prerequisites:

- One is to update the last version of binary files, which makes sure your cluster has stable and capable functionality
- The other one is to set up the DNS server

A Kubernetes DNS server can reduce some steps and dependency for installing cluster-like pods. In here, it is easier to deploy a monitoring system in Kubernetes with a DNS server.

In Kubernetes, how does the DNS server gives assistance in large-system deployment?

The DNS server can support resolving the name of the Kubernetes service for every container. Therefore, while running a pod, we don't have to set a specific service IP for connecting to other pods. Containers in a pod just need to know the service name.

The daemon of the node `kubelet` assigns containers to the DNS server by modifying the file `/etc/resolv.conf`. Try to check the file or use the command `nslookup` for verification after you have installed the DNS server:

```
# kubectl exec <POD_NAME> [-c <CONTAINER_NAME>] -- cat
/etc/resolv.conf
// Check where the service "kubernetes" served
# kubectl exec <POD_NAME> [-c <CONTAINER_NAME>] --
nslookup kubernetes
```

Updating Kubernetes to the latest version: 1.2.1

Updating the version of a running Kubernetes system is not troublesome. You can simply follow these steps. The procedure is similar for both master and node:

- Since we are going to upgrade every Kubernetes' binary file, stop all of the Kubernetes services before you upgrade. For instance, `service <KUBERNETES_DAEMON> stop`
- Download the latest tarball file version `1.2.1`:

  ```
  # cd /tmp && wget https://storage.googleapis.com/kubernetes-
  release/release/v1.2.1/kubernetes.tar.gz
  ```

- Decompress the file in a permanent directory. We are going to use the add-on templates provided in the official source files. These templates can help to create both the DNS server and the monitoring system:

  ```
  // Open the tarball under /opt
  # tar -xvf /tmp/kubernetes.tar.gz -C /opt/
  // Go further decompression for binary files
  # cd /opt && tar -xvf /opt/kubernetes/server/kubernetes-server-
  linux-amd64.tar.gz
  ```

- Copy the new files and overwrite the old ones:

  ```
  # cd /opt/kubernetes/server/bin/
  // For master, you should copy following files and confirm to
  overwrite
  # cp kubectl hypercube kube-apiserver kube-controller-manager
  kube-scheduler kube-proxy /usr/local/bin
  // For nodes, copy the below files
  # cp kubelet kube-proxy /usr/local/bin
  ```

- Finally, you can start the system services. It is good to verify the version through the command line:

  ```
  # kubectl version
  Client Version: version.Info{Major:"1", Minor:"2",
  GitVersion:"v1.2.1", GitCommit:"50809107cd47a1f62da362bccefdd9e
  6f7076145", GitTreeState:"clean"}
  Server Version: version.Info{Major:"1", Minor:"2",
  GitVersion:"v1.2.1", GitCommit:"50809107cd47a1f62da362bccefdd9e
  6f7076145", GitTreeState:"clean"}
  ```

As a reminder, you should update both the master and node at the same time.

Setting up the DNS server

As mentioned, we will use the official template to build up the DNS server in our Kubernetes system. There are just two steps. First, modify the templates and create the resources. Then, we need to restart the `kubelet` daemon with DNS information.

Start the server using templates

The add-on files of Kubernetes are located at `<KUBERNETES_HOME>/cluster/addons/`. According to the last step, we can access the add-on files for DNS at `/opt/kubernetes/cluster/addons/dns`. Two template files are going to be modified and executed. Feel free to depend on the following steps:

- Copy the file from the format `.yaml.in` to the YAML file and we will edit the copied ones later:

```
# cp skydns-rc.yaml.in skydns-rc.yaml
```

Input variable	Substitute value	Example
`{{ pillar['dns_domain'] }}`	The domain of this cluster	`k8s.local`
`{{ pillar['dns_replicas'] }}`	The number of replica for this replication controller	`1`
`{{ pillar['dns_server'] }}`	The private IP of DNS server. Must also be in the CIDR of cluster	`192.168.0.2`

```
# cp skydns-svc.yaml.in skydns-svc.yaml
```

- In these two templates, replace the `pillar` variable, which is covered by double big parentheses, with the items in this table. As you know, the default service `kubernetes` will occupy the first IP in CIDR. That's why we will use IP `192.168.0.2` for our DNS server:

```
# kubectl get svc
NAME         CLUSTER-IP    EXTERNAL-IP   PORT(S)   AGE
kubernetes   192.168.0.1   <none>        443/TCP   4d
```

- In the template for the replication controller, the file named `skydns-rc.yaml` specifies the master URL in the container `kube2sky`:

```
# cat skydns-rc.yaml
(Ignore above lines)
:
- name: kube2sky
```

```
  image: gcr.io/google_containers/kube2sky:1.14
  resources:
    limits:
      cpu: 100m
      memory: 200Mi
    requests:
      cpu: 100m
      memory: 50Mi
  livenessProbe:
    httpGet:
      path: /healthz
      port: 8080
      scheme: HTTP
    initialDelaySeconds: 60
    timeoutSeconds: 5
    successThreshold: 1
    failureThreshold: 5
  readinessProbe:
    httpGet:
      path: /readiness
      port: 8081
      scheme: HTTP
    initialDelaySeconds: 30
    timeoutSeconds: 5
  args:
  # command = "/kube2sky"
  - --domain=k8s.local
  - --kube-master-url=<MASTER_ENDPOINT_URL>:<EXPOSED_PORT>
:
(Ignore below lines)
```

After you finish the preceding steps for modification, you can just start them using the subcommand `create`:

```
# kubectl create -f skydns-svc.yaml
service "kube-dns" created
# kubectl create -f skydns-rc.yaml
replicationcontroller "kube-dns-v11" created
```

Enable Kubernetes DNS in kubelet

Next, we have access to each node and add DNS information to the daemon `kubelet`. The tags we used for the cluster DNS are `--cluster-dns`, for assign the IP of DNS server, and `--cluster-domain`, which define the domain of Kubernetes services:

```
// For init service daemon
# cat /etc/init.d/kubernetes-node
(Ignore above lines)
:
# Start daemon.
echo $"Starting kubelet: "
        daemon $kubelet_prog \
                --api_servers=<MASTER_ENDPOINT_URL>:<EXPOSED_PORT> \
                --v=2 \
                --cluster-dns=192.168.0.2 \
                --cluster-domain=k8s.local \
                --address=0.0.0.0 \
                --enable_server \
                --hostname_override=${hostname} \
                > ${logfile}-kubelet.log 2>&1 &
:
(Ignore below lines)
// Or, for systemd service
# cat /etc/kubernetes/kubelet
(Ignore above lines)
:
# Add your own!
KUBELET_ARGS="--cluster-dns=192.168.0.2 --cluster-domain=k8s.local"
```

Now, it is good for you to restart either the service `kubernetes-node` or just `kubelet`! And you can enjoy the cluster with a DNS server.

How to do it...

In this section, we will work on installing a monitoring system and introducing its dashboard. This monitoring system is based on **Heapster** (`https://github.com/kubernetes/heapster`), a resource usage collecting and analyzing tool. Heapster communicates with kubelet to get the resource usage of both machine and container. Along with Heapster, we have **influxDB** (`https://influxdata.com`) for storage and **Grafana** (`http://grafana.org`) as the frontend dashboard, which visualizes the status of resources in several user-friendly plots.

Installing a monitoring cluster

If you have gone through the preceding section about the prerequisite DNS server, you must be very familiar with deploying the system with official add-on templates.

1. Let's check the directory `cluster-monitoring` under `<KUBERNETES_HOME>/cluster/addons`. There are different environments provided for deploying the monitoring cluster. We choose `influxdb` in this recipe for demonstration:

   ```
   # cd /opt/kubernetes/cluster/addons/cluster-monitoring/influxdb &&
   ls

   grafana-service.yaml      heapster-service.yaml
   influxdb-service.yaml

   heapster-controller.yaml  influxdb-grafana-controller.yaml
   ```

   Under this directory, you can see three templates for services and two for replication controllers.

2. We will retain most of the service templates as the original ones. Because these templates define the network configurations, it is fine to use the default settings but expose Grafana service:

   ```
   # cat heapster-service.yaml
   apiVersion: v1
   kind: Service
   metadata:
     name: monitoring-grafana
     namespace: kube-system
     labels:
       kubernetes.io/cluster-service: "true"
       kubernetes.io/name: "Grafana"
   spec:
     type: NodePort
     ports:
   ```

```
    - port: 80
      nodePort: 30000
      targetPort: 3000
  selector:
    k8s-app: influxGrafana
```

As you can see, we expose Grafana service with port `30000`. This revision will allow us to access the dashboard of monitoring from browser.

3. On the other hand, the replication controller of Heapster and the one combining influxDB and Grafana require more additional editing to meet our Kubernetes system:

```
# cat influxdb-grafana-controller.yaml
(Ignored above lines)
:
- image: gcr.io/google_containers/heapster_grafana:v2.6.0-2
          name: grafana
          env:
          resources:
            # keep request = limit to keep this container in
guaranteed class
            limits:
              cpu: 100m
              memory: 100Mi
            requests:
              cpu: 100m
              memory: 100Mi
          env:
            # This variable is required to setup templates in
Grafana.
            - name: INFLUXDB_SERVICE_URL
              value: http://monitoring-influxdb.kube-system:8086
            - name: GF_AUTH_BASIC_ENABLED
              value: "false"
            - name: GF_AUTH_ANONYMOUS_ENABLED
              value: "true"
            - name: GF_AUTH_ANONYMOUS_ORG_ROLE
              value: Admin
            - name: GF_SERVER_ROOT_URL
```

```
              value: /
:
(Ignored below lines)
```

For the container of Grafana, please change some environment variables. The first one is the URL of influxDB service. Since we set up the DNS server, we don't have to specify the particular IP address. But an extra-postfix domain should be added. It is because the service is created in the namespace `kube-system`. Without adding this postfix domain, DNS server cannot resolve `monitoring-influxdb` in the default namespace. Furthermore, the Grafana root URL should be changed to a single slash. Instead of the default URL, the root (`/`) makes Grafana transfer the correct webpage in the current system.

4. In the template of Heapster, we run two Heapster containers in a pod. These two containers use the same image and have similar settings, but actually, they take to different roles. We just take a look at one of them as an example of modification:

```
# cat heapster-controller.yaml
(Ignore above lines)
:
      containers:
        - image: gcr.io/google_containers/heapster:v1.0.2
          name: heapster
          resources:
            limits:
              cpu: 100m
              memory: 200Mi
            requests:
              cpu: 100m
              memory: 200Mi
          command:
            - /heapster
            - --source=kubernetes:<MASTER_ENDPOINT_URL>:<EXPOSED_
PORT>?inClusterConfig=false
            - --sink=influxdb:http://monitoring-influxdb.kube-
system:8086
            - --metric_resolution=60s
:
(Ignore below lines)
```

At the beginning, remove all double-big-parentheses lines. These lines will cause creation error, since they cannot be parsed or considered in the YAML format. Still, there are two input variables that need to be replaced to possible values. Replace `{{ metrics_memory }}` and `{{ eventer_memory }}` to `200Mi`. The value 200MiB is a guaranteed amount of memory that the container could have. And please change the usage for Kubernetes source. We specify the full access URL and port, and disable `ClusterConfig` for refraining authentication. Remember to adjust on both the `heapster` and `eventer` containers.

5. Now you can create these items with simple commands:

```
# kubectl create -f influxdb-service.yaml
service "monitoring-influxdb" created
# kubectl create -f grafana-service.yaml
You have exposed your service on an external port on all nodes in
your
cluster.  If you want to expose this service to the external
internet, you may
need to set up firewall rules for the service port(s) (tcp:30000)
to serve traffic.

See http://releases.k8s.io/release-1.2/docs/user-guide/services-
firewalls.md for more details.
service "monitoring-grafana" created
# kubectl create -f heapster-service.yaml
service "heapster" created
# kubectl create -f influxdb-grafana-controller.yaml
replicationcontroller "monitoring-influxdb-grafana-v3" created
// Because heapster requires the DB server and service to be
ready, schedule it as the last one to be created.
# kubectl create -f heapster-controller.yaml
replicationcontroller "heapster-v1.0.2" created
```

6. Check your Kubernetes resources at namespace `kube-system`:

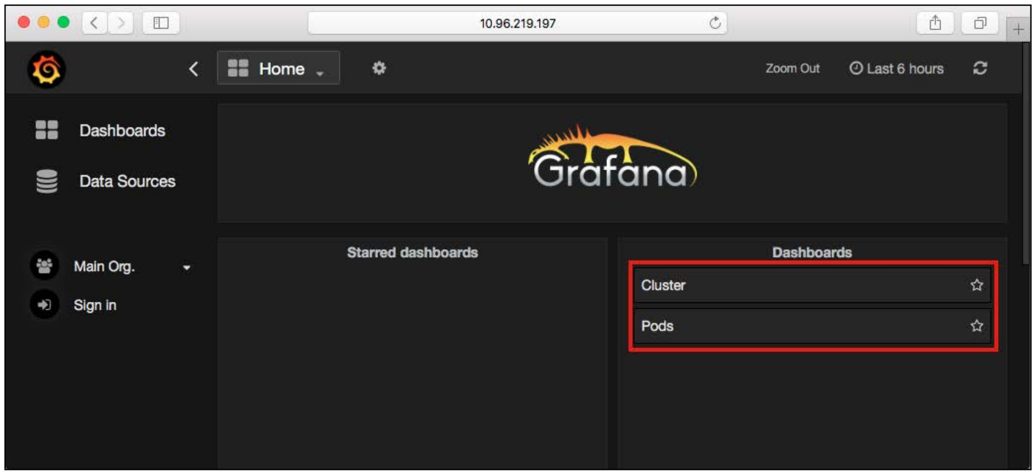
```
# kubectl get svc --namespace=kube-system
NAME                    CLUSTER-IP        EXTERNAL-IP   PORT(S)
AGE
heapster                192.168.135.85    <none>        80/TCP
12m
```

```
kube-dns              192.168.0.2      <none>        53/UDP,53/
TCP       15h
monitoring-grafana    192.168.84.223   nodes         80/TCP
12m
monitoring-influxdb   192.168.116.162  <none>        8083/
TCP,8086/TCP   13m
# kubectl get pod --namespace=kube-system
NAME                                   READY     STATUS
RESTARTS   AGE
heapster-v1.0.2-r6oc8                  2/2       Running   0
4m
kube-dns-v11-k81cm                     4/4       Running   0
15h
monitoring-influxdb-grafana-v3-d6pcb   2/2       Running   0
12m
```

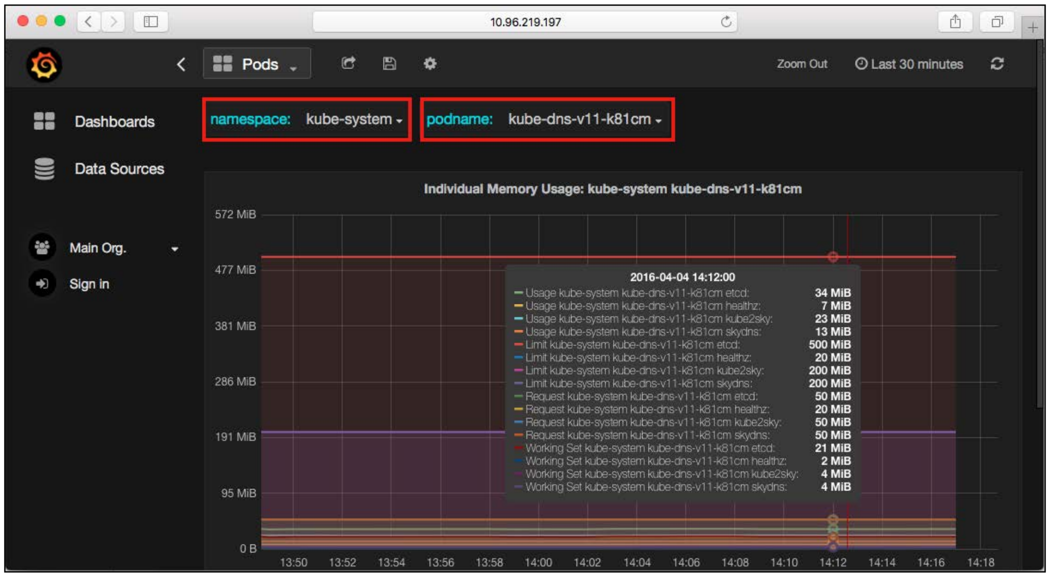
Congratulations! Once you have all the pods in a ready state, let's check the monitoring dashboard.

Introducing the Grafana dashboard

At this moment, the Grafana dashboard is available through nodes' endpoints. Please make sure the node's firewall or security group on AWS has opened port `30000` to your local subnet. Take a look at the dashboard using a browser. Type `<NODE_ENDPOINT>:30000` in your URL search bar:



In the default settings, we have two dashboards **Cluster** and **Pods**. The **Cluster** board covers nodes' resource utilization, such as CPU, memory, network transaction, and storage. The **Pods** dashboard has similar plots for each pod and you can watch each container in a pod.



Individual Memory Usage: kube-system kube-dns-v11-k81cm

2016-04-04 14:12:00

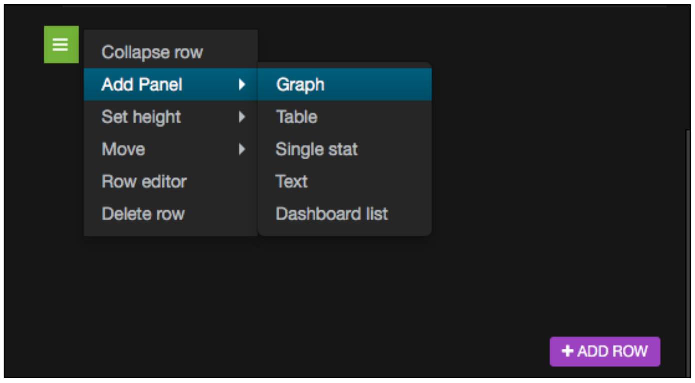
Usage kube-system kube-dns-v11-k81cm etcd:	34 MiB
Usage kube-system kube-dns-v11-k81cm healthz:	7 MiB
Usage kube-system kube-dns-v11-k81cm kube2sky:	23 MiB
Usage kube-system kube-dns-v11-k81cm skydns:	13 MiB
Limit kube-system kube-dns-v11-k81cm etcd:	500 MiB
Limit kube-system kube-dns-v11-k81cm healthz:	20 MiB
Limit kube-system kube-dns-v11-k81cm kube2sky:	200 MiB
Limit kube-system kube-dns-v11-k81cm skydns:	200 MiB
Request kube-system kube-dns-v11-k81cm etcd:	50 MiB
Request kube-system kube-dns-v11-k81cm healthz:	20 MiB
Request kube-system kube-dns-v11-k81cm kube2sky:	50 MiB
Request kube-system kube-dns-v11-k81cm skydns:	50 MiB
Working Set kube-system kube-dns-v11-k81cm etcd:	21 MiB
Working Set kube-system kube-dns-v11-k81cm healthz:	2 MiB
Working Set kube-system kube-dns-v11-k81cm kube2sky:	4 MiB
Working Set kube-system kube-dns-v11-k81cm skydns:	4 MiB

As the preceding images show, for example, we can observe the memory utilization of individual containers in the pod `kube-dns-v11`, which is the cluster of the DNS server. The purple lines in the middle just indicate the limitation we set to the containers `skydns` and `kube2sky`.

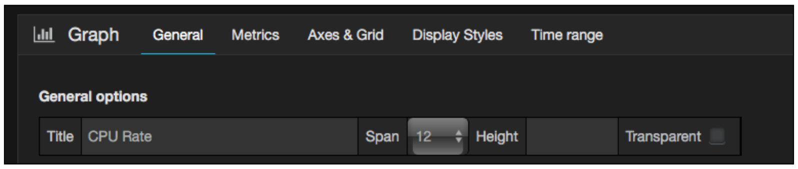
Creating a new metric to monitor pods

There are several metrics for monitoring offered by Heapster (`https://github.com/kubernetes/heapster/blob/master/docs/storage-schema.md`). We are going to show you how to create a customized panel by yourself. Please take the following steps as a reference:

1. Go to the **Pods** dashboard and click on **ADD ROW** at the bottom of the webpage. A green button will show up on the left-hand side. Choose to add a graph panel.



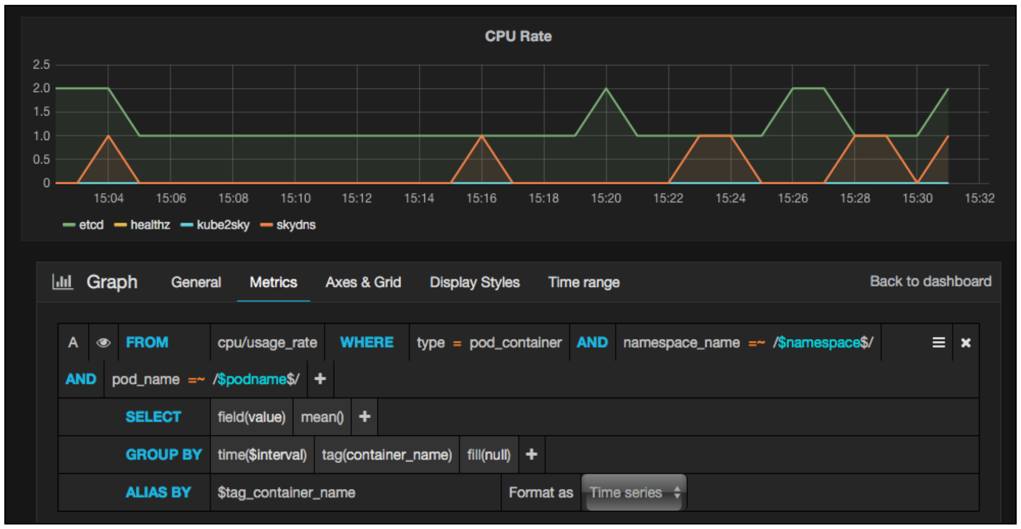
2. First, give your panel a name. For example, CPU Rate. We would like to create one showing the rate of CPU's utility:



General options

Title CPU Rate Span 12 Height Transparent

3. Set up the parameters in the query as shown in the following screenshot:

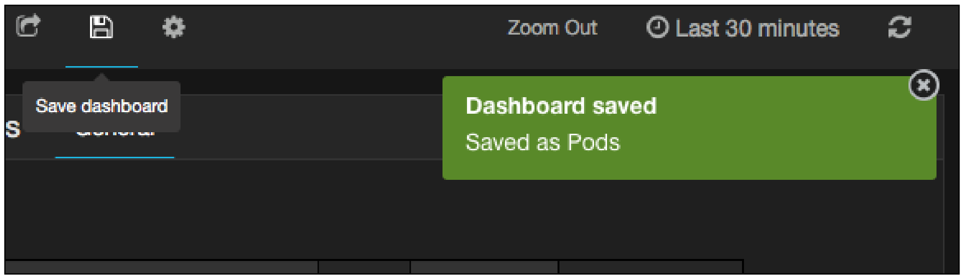


A FROM cpu/usage_rate WHERE type = pod_container AND namespace_name =~ /$namespace$/
AND pod_name =~ /$podname$/ +
SELECT field(value) mean() +
GROUP BY time($interval) tag(container_name) fill(null) +
ALIAS BY $tag_container_name Format as Time series

- `FROM`: For this parameter input `cpu/usage_rate`
- `WHERE`: For this parameter set `type = pod_container`

- `AND`: Set this parameter with the `namespace_name=$namespace`, `pod_name= $podname` value
- `GROUP BY`: Enter `tag(container_name)` for this parameter
- `ALIAS BY`: For this parameter input `$tag_container_name`

4. Good job! You can now save the pod by clicking on the icon at the top:



Dashboard saved
Saved as Pods

Just try to discover more functionality of the Grafana dashboard and the Heapster monitoring tool. You will get more details about your system, services, and containers through the information from the monitoring system.

See also

This recipe informs you how to monitor your master node and nodes in the Kubernetes system. However, it is wise to study the recipes about the main components and daemons. You can get more of an idea about the view of working processes and resource usage. Moreover, since we have worked with several services to build our monitoring system, reviewing the recipe about the Kubernetes services again will give you a clear idea about how you can build up this monitoring system.

- The *Creating an overlay network*, *Configuring master*, and *Configuring nodes* recipes in *Chapter 1*, *Building Your Own Kubernetes*
- The *Working with services* recipe in *Chapter 2*, *Walking through Kubernetes Concepts*

Kubernetes is a project which keeps moving forward and upgrading at a fast speed. The recommended way for catching up is to check out new features on its official website: `http://kubernetes.io`. Also, you can always get new Kubernetes on GitHub: `https://github.com/kubernetes/kubernetes/releases`. Keeping your Kubernetes system up to date, and learning new features practically, is the best method to access the Kubernetes technology continuously.

Bibliography

This course is a blend of different projects and texts all packaged up keeping your journey in mind. It includes the content from the following Packt products:

- *Puppet Cookbook - Third Edition – Thomas Uphill, John Arundel*
- *Docker Cookbook – Neependra Khare*
- *Kubernetes Cookbook – Hideto Saito, Hui-Chuan Chloe Lee, Ke-Jou Carol Hsu*

Index