O'REILLY®

Docker Cookbook

SOLUTIONS AND EXAMPLES FOR
BUILDING DISTRIBUTED APPLICATIONS

Early Release

RAW & UNEDITED

Sébastien Goasguen

Docker Cookbook

Sebastien Goasguen

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Table of Contents

Preface

This book is not finished, it has not gone through technical review nor has it been edited for grammar, punctuation, typos etc. You are reading this book through the advanced release program of O'Reilly, please consider it a preview of a draft and be kind in your reviews. Feel free to send me any suggestions or comments to *how2dock@gmail.com*. You can also file a review in the O'Reilly portal. Happy reading.

Why I Wrote This Book

I have been working on Clouds at the IaaS layer for over ten years. With Amazon AWS, Google GCE and Microsoft Azure now providing large scale Cloud services for several years, it is fair to say that getting access to a server has never been that easy and that quick. The real value to me has been the availability of an API to access these services. We can now program to create an infrastructure and program to deploy an application. These programmable layers help us reach a higher level of automation, which for a business translates in faster time to market, more innovation and better user service.

However, application packaging, configuration, composition of services in a distributed environment has not progressed much despite a lot of work in configuration management and orchestration. Deploying and running a distributed application at scale and in a fault tolerant manner is still hard.

I was not crazy about Docker until I tried it and understood what it brings to the table. Docker primarily brings a new user experience to linux containers. It is not about full virtualization versus containers, it is about the ease of packaging and running an application. Once you start using Docker and enjoy this new experience, the side effect is that you will also start thinking automatically about composition and clustering.

Containers help us think more in terms of *functional isolation* which in turn forces us to decompose our applications before stitching it back together for a distributed world. Yes, I have drunk the Kool-Aid but hopefully this book will show you why.

How This Book Is Organized

This cookbook is currently made of ten chapters. Each chapter is composed of *recipes* written in the standard O'Reilly recipes format (i.e Problem, Solution, Discussion). You can read it front to back or pick up a specific chapter/recipe. Each recipe is made to be independent of the others but when concepts needed in a recipe are needed, appropriate references are provided.

- The Chapter 1 chapter goes through several Docker installation scenarios including *Docker machine*. It then presents the basic Docker commands to manage containers, mount data volumes, link containers and so on. At the end of this chapter you should have a working Docker host and you should have started multiple containers as well as understood the lifecycle of containers.
- In Chapter 2 we introduce the *Dockerfile*, *Docker Hub* and show how to build/tag/commit an image. We also show how to run your own Docker registry and setup automated builds. At the end of this chapter you will know how to create Docker images and share them privately or publicly and have some basic foundation to build continuous delivery pipelines.
- Currently the Chapter 3 chapter only contains stubs of planned recipes. You will find information such as linking containers, using an ambassador container to expose services from different hosts, configuring a custom bridge for use in your Docker host. You will also learn about more advanced networking setups and tools, such as Weave, Flannel and Socketplane VXLAN overlays. Empty chapter right now.
- The Chapter 4 chapter goes through some configuration of the Docker daemon, especially security settings and access to the Docker API remotely. It also covers a few basic problems, like compiling Docker from source, running its test suite and using a new Docker binary. A few recipes are meant to gain a better insight on linux namespaces and its use in containers.
- Chapter 5 introduces the new container management platform from Google. Kubernetes provides a way to deploy multi container applications on a distributed cluster. In addition it provides an automated way to expose services and create replicas of containers. We show how to deploy Kubernetes on your own infrastructure, starting with a local Vagrant cluster and subsequently on a set of machines started in the Cloud. We then present the key aspects of Kubernetes: *pods*, *services* and *replication controllers*. Empty chapter right now.

- In the Chapter 6 chapter we cover three new linux distributions that are customized to run containers: CoreOS, Project Atomic and Ubuntu core. These new distributions provide just enough operating system to run and orchestrate docker containers. Recipes cover installation and access to machines that use these distributions. We also introduce tools that are used with these distributions to ease container orchestration (e.g `etcd`, `fleet`, `systemd`)
- One of Docker's strength is its booming ecosystem. In Chapter 7 we introduce several tools that have been created over the last 18 months and that leverage Docker to ease application deployment, continuous integration, service discovery and orchestration. As an example, you will find recipes about Fig, Docker Swarm, Flynn and Apache Clocker.
- The Docker daemon can be installed on a developer local machine, however, with Cloud computing providing easy access to on-demand servers it is fair to say that a lot of container based applications will be deployed in the Cloud. In Chapter 8 we present a few recipes to show how to access a Docker host on Amazon AWS, Google GCE and Microsoft Azure. We also introduce two new cloud services that use Docker: The AWS Elastic Container Service (ECS) and the Google Container Engine. This chapter is currently stubbed out. Empty chapter right now.
- The Chapter 9 chapters aims to address some concerns about application monitoring when using containers. Monitoring and visibility of the infrastructure and the application has been a huge focus in the DevOps community. As Docker becomes more pervasive as a development and operational mechanism, lessons learned need to be applied to container based applications. This chapter is currently stubbed out. Empty chapter right now.
- Finally, in the ??? chapter we present end to end application deployment scenarios on both single host and clusters. While some basic application deployments are presented in earlier chapters. The recipes presented here aim to be closer to a production deployment setup. This is a more in depth chapter that puts the reader on the path towards designing more complex microservices. This chapter is currently stubbed out. Empty chapter right now.

The book structure may still change and the order of the chapters may be modified.

Finally, ??? summarizes the book and provides some tips for further reading and investigation.

Technology You Need to Understand

This book is of an intermediate level and requires a minimum understanding of a few development and system administration concepts. Before diving into the book, you might want to review:

Bash (Unix shell)
: This is the default Unix shell on Linux and OS X. Familiarity with the Unix shell, such as editing files, setting file permissions, moving files around the filesystems, user privileges, and some basic shell programming will be very beneficial. If you don't know the Linux shell in general, consult books such as Cameron Newham's Learning the Bash Shell or Carl Albing, JP Vossen, and Cameron Newham's Bash Cookbook, both from O'Reilly.

Package management
: The tools we will present in this book often have multiple dependencies that need to be met by installing some packages. Knowledge of the package management on your machine is therefore required. It could be `apt` on Ubuntu/Debian systems, `yum` on CentOS/RHEL systems, `port` or `brew` on OS X. Whatever it is, make sure that you know how to install, upgrade, and remove packages.

Git
: Git has established itself as the standard for distributed version control. If you are already familiar with CVS and SVN, but have not yet used Git, you should. Version Control with Git by Jon Loeliger and Matthew McCullough (O'Reilly) is a good start. Together with Git, the GitHub website is a great resource to get started with a hosted repository of your own. To learn GitHub, try *http://training.github.com* and the associated interactive tutorial.

Python
: In addition to programming with C/C++ or Java, I always encourage students to pick up a scripting language of their choice. Perl used to rule the world, while these days, Ruby and Go seem to be prevalent. I personally use Python. Most examples in this book use Python but there are a few examples with Ruby, one even uses Clojure. O'Reilly offers an extensive collection of books on Python, including Introducing Python by Bill Lubanovic, Programming Python by Mark Lutz, and Python Cookbook by David Beazley and Brian K. Jones.

Vagrant
: Vagrant has become one of the great tools for DevOps engineer to build and manage their virtual environments. It is best suited for testing and quickly provisioning virtual machines locally, but also has several plugins to connect to public cloud providers. In this book we use Vagrant to quickly deploy a virtual machine

instance that acts as a `docker` host. You might want to read Vagrant, Up and Running from the author of Vagrant itself, Mitchell Hashimoto.

Go
: Docker is written in Go. Over the last couple years `go` has established itself has the new programming language of choice in many startups. Docker is written in go, and while this cookbook is not about `go` programming, we will show how to compile a few `go` projects. If you want to know more, Go Up and Running is a good start.

Online Content

Code examples, Vagrantfile and other scripts used in this book are available at GitHub. You can clone this repository, go to the relevant chapter and recipe and use the code as is. For example to start an Ubuntu 14.04 virtual machine using Vagrant and install Docker do:

```
$ git clone https://github.com/how2dock/docbook.git
$ cd dockbook/ch01/ubuntu14.04/
$ vagrant up
```

The examples in this repo are not made to represent optimized setups. There are the basic minimum required to run the examples in the recipes. Until the book is complete, expect frequent changes to this repo.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip, suggestion, or general note.

This element indicates a warning or caution.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc. 1005 Gravenstein Highway North Sebastopol, CA 95472 800-998-9938 (in the United States or Canada) 707-829-0515 (international or local) 707-829-0104 (fax)
We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://shop.oreilly.com/product/0636920034377.do*.

To comment or ask technical questions about this book, send email to bookquestions@oreilly.com.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgements

CHAPTER 1

Getting Started with Docker

This chapter consists of introductory recipes. Readers should be able to go through the recipes to install a Docker Host, then progressively discover the Docker CLI to manage containers and start a two container application. You can send me suggestions at *how2dock@gmail.com*

There will be an introduction here.

1.1 Installing Docker on Ubuntu 14.04 and CentOS 6.5

Problem

You want to use Docker on Ubuntu 14.04 or CentOS 6.5

Solution

Install the `docker.io` package using your package management system and start the docker service.

On Ubuntu 14.04 this is achieved in two lines of bash commands. The actual package to install is `docker.io` since there is a pre-existing docker package for Ubuntu which is not related to Docker.

```
$ sudo apt-get update
$ sudo apt-get install -y docker.io
```

You can test that the installation worked fine by checking the version of docker:

```
$ sudo docker --version
Docker version 1.0.1, build 990021a
```

You can stop, start, restart the service. For example, to restart it do:

```
$ sudo service docker.io restart
```

On CentOS 6.5, getting `docker` is achieved by grabing the `docker-io` package from the EPEL repository.

```
$ sudo yum -y update
$ sudo yum -y install epel-release
$ sudo yum -y install docker-io
$ sudo service docker start
$ chkconfig docker on
```

While on Ubuntu 14.04 the setup installed version 1.0.1, on CentOS 6.5 it installed version 1.3.1

```
# docker --version
Docker version 1.3.1, build c78088f/1.3.1
```

If you want to use `docker` from a non root user, add the user account to the `docker` group

```
$ sudo gpasswd -a <user> docker
```

Discussion

To install the latest version of Docker on fedora, Ubuntu, Debian, Linux Mint and Gentoo, there exists a simple bootstrap script that you can run instead.

```
$ sudo curl -sSL https://get.docker.com/ | sudo sh
$ sudo docker --version
Docker version 1.4.1, build 5bc2ff8
```

See Also

For installation of Docker on other operating systems see the official installation documentation

By Publishing time, the default version number and latest version number of Docker might change.

1.2 Setting Up a Local Docker Host Using Vagrant

Problem

The operating system of your local machine is different than the operating system you want to use Docker on. For example you are running OSX and want to try Docker on Ubuntu.

Solution

Use Vagrant to start a virtual machine (VM) locally and bootstrap the VM using a shell provisioner in the Vagrantfile.

With a working Virtual Box and Vagrant installation, create a *Vagrantfile*:

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "ubuntu/trusty64"

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

  config.vm.provider "virtualbox" do |vb|
     vb.customize ["modifyvm", :id, "--memory", "1024"]
  end

  config.vm.provision :shell, :path => "docker-bootstrap.sh"

end
```

And create a *docker_bootstrap.sh* script:

```
#!/bin/bash

sudo apt-get update
sudo apt-get -y install docker.io
sudo gpasswd -a vagrant docker
sudo service docker.io restart
```

You can then bring up the virtual machine. Vagrant will download the ubuntu/trusty64 box from the Vagrant cloud, start an instance of it using virtual box and run the bootstrap script in the instance. You will then be able to *ssh* to the instance and use `docker`

```
$ vagrant up
$ vagrant ssh
vagrant@vagrant-ubuntu-trusty-64:~$ docker ps
CONTAINER ID    IMAGE       COMMAND     CREATED     STATUS      PORTS       NAMES
```

In this Vagrant setup, the vagrant user was added to the docker group. Hence docker commands can be issued even if you are not root. You can get these scripts from the how2dock repository in the `ch01` directory.

Discussion

If you have never used Vagrant before, you will need to install it. The download page on the Vagrant website lists all major packages families. For example on Debian based systems grab the .deb package and install it like so:

```
$ wget https://dl.bintray.com/mitchellh/vagrant/vagrant_1.6.5_x86_64.deb
$ sudo dpkg -i vagrant_1.6.5_x86_64.deb
$ sudo vagrant --version
Vagrant 1.6.5
```

1.3 Using `boot2docker` to Get a Docker Host on OSX

Problem

The Docker daemon is not supported on OSX, but you want to use the Docker client seamlessly on your OSX host.

Solution

Use the boot2docker lightweight linux distribution. Boot2docker is based on Tiny Core Linux and configured specifically to act as a docker host. After installation, a `boot2docker` command will be available to you. You will use it to interact with a virtual machine started through virtual box that will act as a docker host. The docker client -which runs on OSX, unlike the daemon- will be setup on your local OSX machine.

Let's start by downloading and installing `boot2docker`. Go to the site where you will find several download links. From the release page, grab the latest release. Once the download is finished, launch the installer.

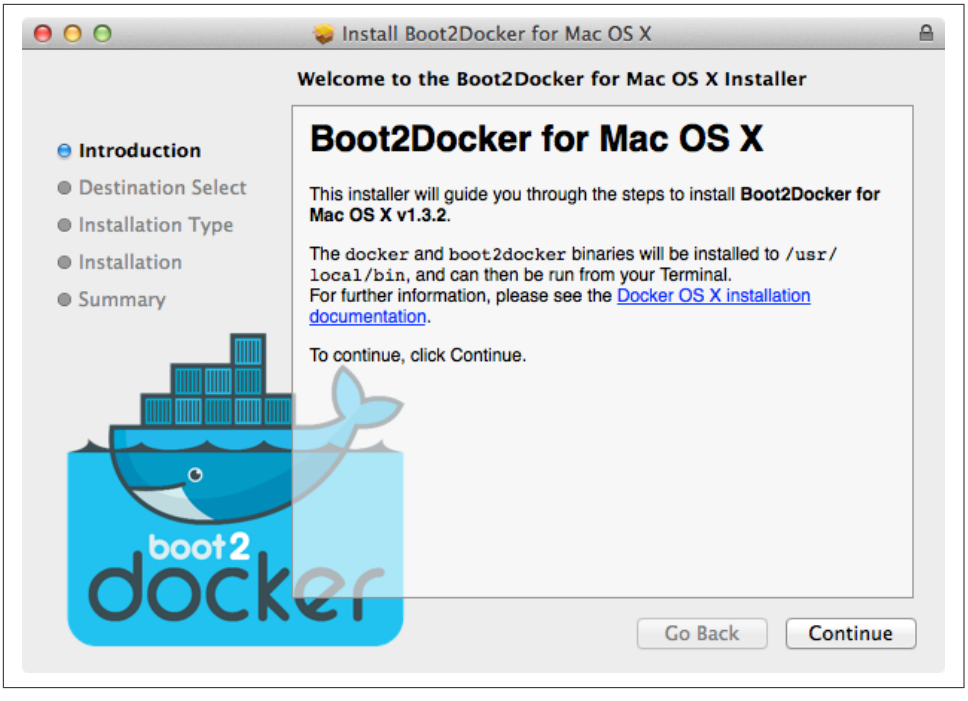


Figure 1-1. Boot2docker installer wizard

Once the installation is finished you are ready to use boot2docker.

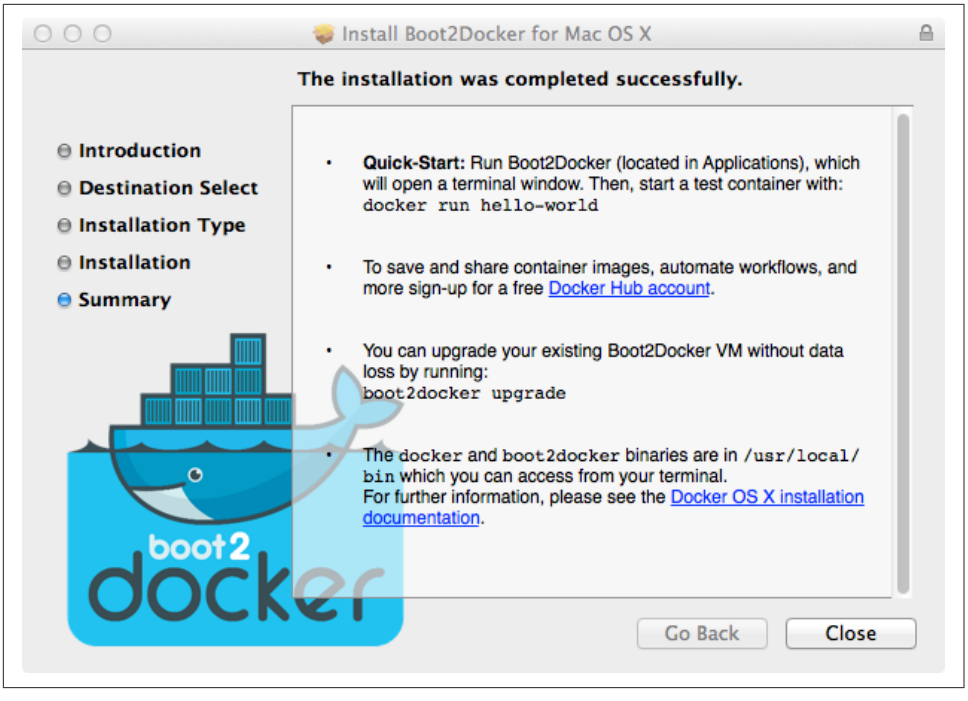


Figure 1-2. Boot2docker installer completion

In a terminal window, type `boot2docker` at the prompt, you should see the usage options. You can also check the version number that you installed.

```
$ boot2docker
Usage: boot2docker [<options>] {help|init|up|ssh|save|down|poweroff|reset|
                               restart|config|status|info|ip|shellinit|delete|
                               download|upgrade|version} [<args>]
$ boot2docker version
Boot2Docker-cli version: v1.3.2
Git commit: e41a9ae
```

With `boot2docker` installed, the first step is to initialize it. If you have not downloaded the boot2docker ISO, this step will do so and create the virtual machine in virtualbox.

```
$ boot2docker init
Latest release for boot2docker/boot2docker is v1.3.2
Downloading boot2docker ISO image...
Success:
    downloaded https://github.com/boot2docker/boot2docker/releases/download/\
        v1.3.2/boot2docker.iso
    to /Users/sebgoa/.boot2docker/boot2docker.iso
```

As you can see the ISO will be located in your home directory under `.boot2docker/boot2docker.iso` and when you open the VirtualBox UI you will see the boot2docker VM in a powered off state.

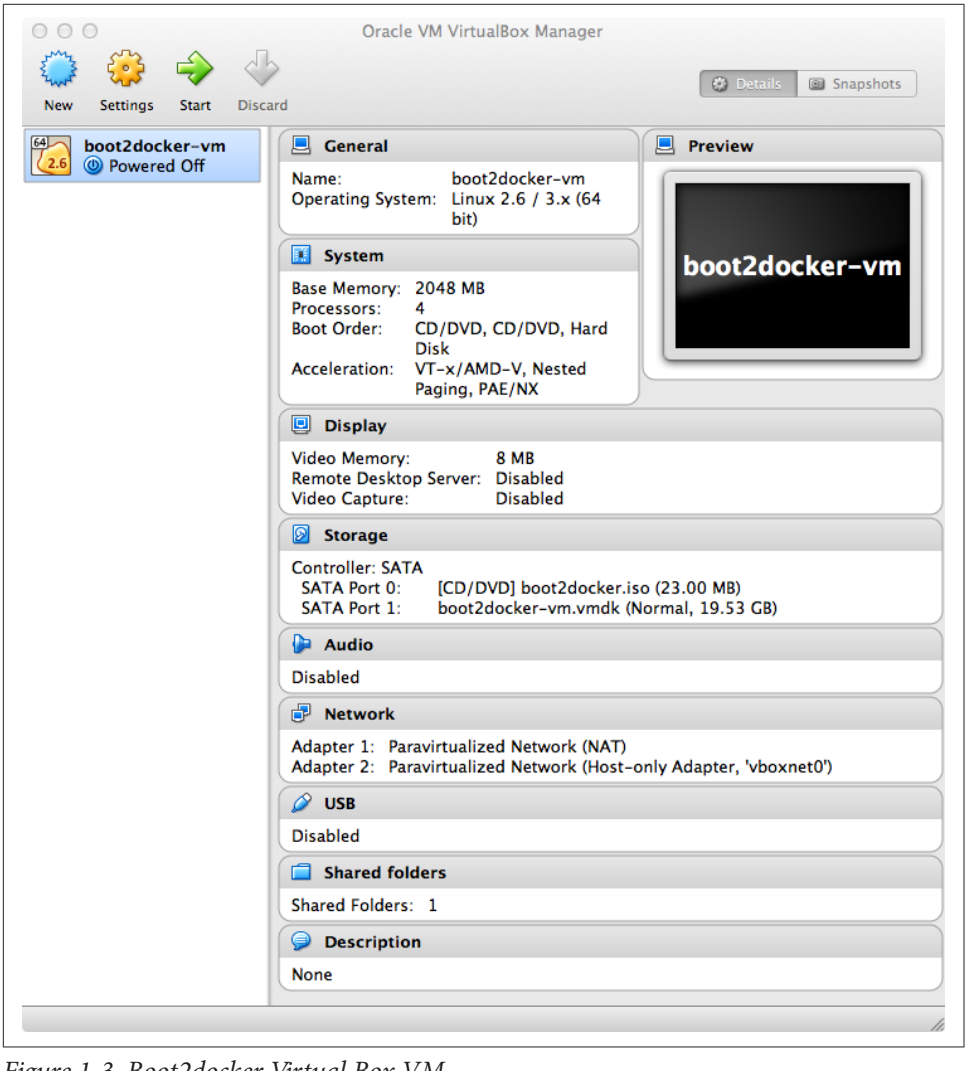


Figure 1-3. Boot2docker Virtual Box VM

You do not need to have the VirtualBox UI open, the snapshots are only here for illustration. Boot2docker uses the VBoxManage commands to manage the boot2docker VM in the background.

You are now ready to start `boot2docker`. This will run the VM and return some instructions to set environment variables for properly connecting to the docker daemon running in the VM.

```
$ boot2docker start
Waiting for VM and Docker daemon to start...
.....................oooooooooooooooooooo
Started.
Writing /Users/sebgoa/.boot2docker/certs/boot2docker-vm/ca.pem
Writing /Users/sebgoa/.boot2docker/certs/boot2docker-vm/cert.pem
Writing /Users/sebgoa/.boot2docker/certs/boot2docker-vm/key.pem

To connect the Docker client to the Docker daemon, please set:
    export DOCKER_CERT_PATH=/Users/sebgoa/.boot2docker/certs/boot2docker-vm
    export DOCKER_TLS_VERIFY=1
    export DOCKER_HOST=tcp://192.168.59.103:2376
```

While you can set the environment variables by hand, boot2docker provides a handy command `shellinit`. Use it to configure the TLS connection to the docker daemon and you will have access to the docker host from your local OSX machine.

```
$ $(boot2docker shellinit)
Writing /Users/sebgoa/.boot2docker/certs/boot2docker-vm/ca.pem
Writing /Users/sebgoa/.boot2docker/certs/boot2docker-vm/cert.pem
Writing /Users/sebgoa/.boot2docker/certs/boot2docker-vm/key.pem
$ docker ps
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS    PORTS    NAMES
```

Discussion

When a new version of `boot2docker` is available, you can upgrade easily by downloading the new boot2docker installer and downloading a new ISO image with the `download` command.

Make sure that you stop the current boot2docker vm `$ boot2docker stop` before running the installer script

```
$ boot2docker stop
$ boot2docker download
$ boot2docker start
```

1.4 Running Boot2docker on Windows 8.1 Desktop

Problem

You have a Windows 8.1 Desktop and would like to use Boot2docker to test Docker.

Solution

Use the Boot2docker windows installer.

After downloading the latest version of the windows installer (an `.exe` binary), run it through the command prompt or through your file explorer (see <<boot2dockeri>>). It will automatically install Virtualbox, MSYSGit and the boot2docker ISO. MSYSGit is necessary to get the `ssk-keygen` binary on your windows machine. Going through the installer wizard, you will need to accept couple VirtualBox licenses from Oracle. The installer can create shortcuts in your Desktop for VirtualBox and to start `boot2docker`.

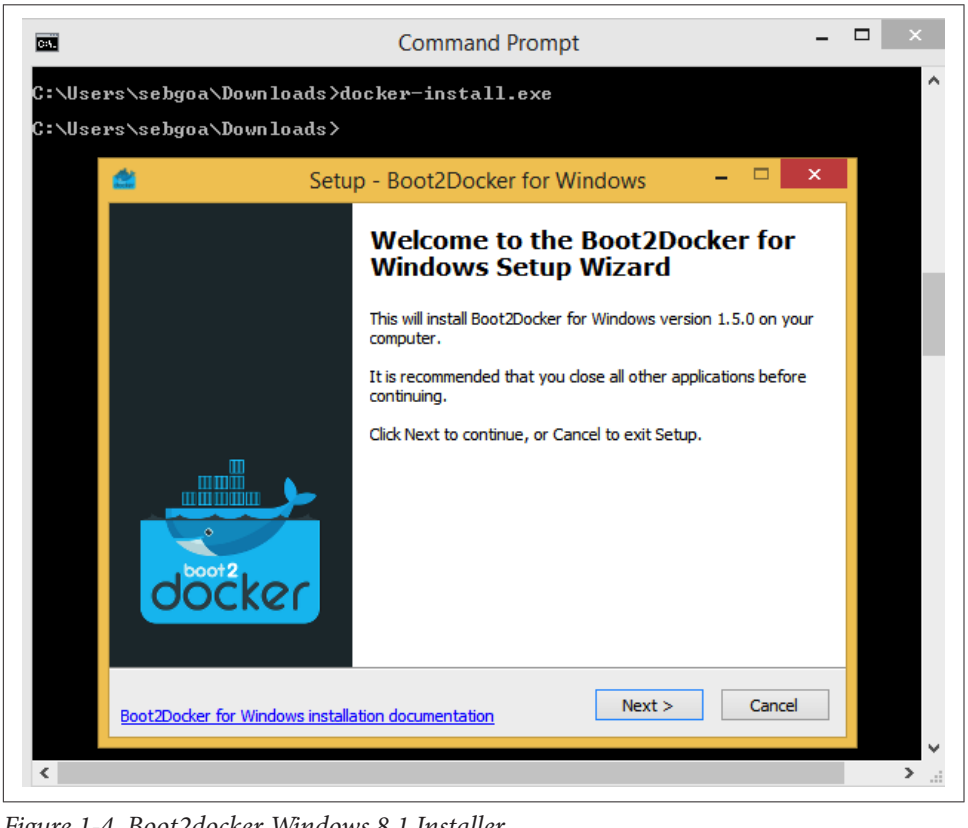


Figure 1-4. Boot2docker Windows 8.1 Installer

Once the installation is finished, double click on the shortcut for `boot2docker`, this will launch the VM in VirtualBox and you will get a command prompt inside it (see Figure 1-5). You can now use Docker on your Windows desktop.

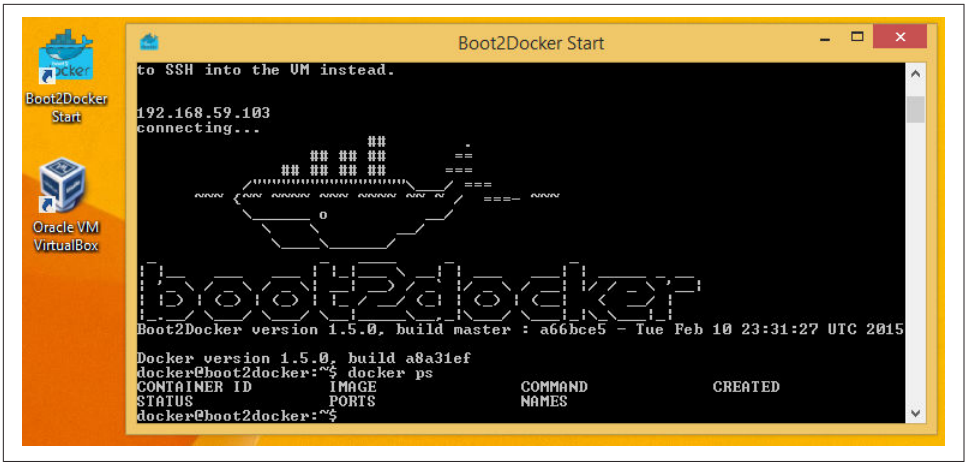


Figure 1-5. Boot2docker Windows 8.1 Command

Discussion

Docker machine also comes with an Hyper-V driver. If you setup Hyper-V on your desktop you could start a boot2docker instance with docker machine.

Recipe on Docker machine with Hyper-V will come.

See Also

- Boot2docker for Windows official Docker documentation.

1.5 Starting a Docker Host in the Cloud Using Docker Machine

Problem

You do not want to install the Docker daemon locally, use Vagrant (Recipe 1.2) or use Boot2docker (Recipe 1.3). Instead you would like to use a Docker host in the Cloud (e.g AWS, DigitalOcean, Azure, GCE etc) and connect to it seamlessly using the local Docker client.

Solution

Use *Docker machine* to start a cloud instance in your public cloud of choice. *Machine* will automatically install Docker and setup TLS for secure communication. You will then be able to use the cloud instance as your Docker host and use it from a local Docker client. See Chapter 8 for more recipes dedicated to using Docker in the Cloud.

Docker machine beta was announced on February 26th 2015. Official documentation is now available on the Docker website. The source code is available on GitHub.

Let's get started. *Machine* currently supports VirtualBox, DigitalOcean, Amazon Web Services, Azure, GCE and a few other providers. There are several drivers under development or review, so we should definitely expect much more soon. In this recipe, we will use DigitalOcean, therefore if you want to follow along step by step you will need an account on DigitalOcean.

Once you have an account, generate an access token for using Docker Machine. This token will need to be both a *read* and a *write* token so that *Machine* can upload a public SSH key (Figure 1-6). Set an environment variable `DIGITALOCEAN_ACCESS_TOKEN` that defines the token you created.

Machine will upload an SSH key to your cloud account, make sure that your access tokens or API keys give you the privileges necessary to create a key.

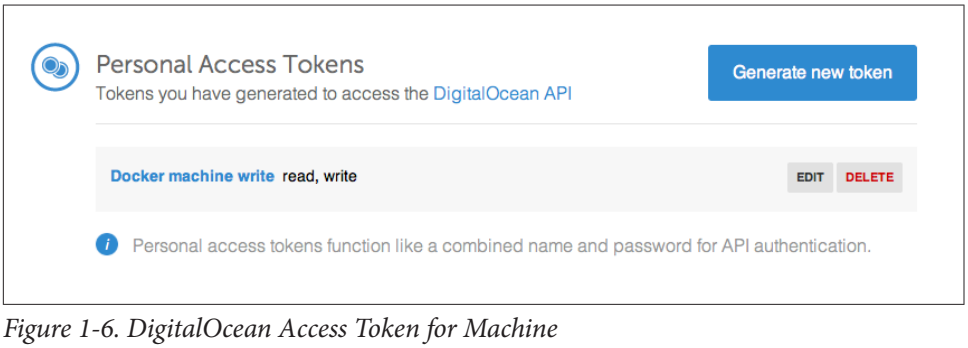


Figure 1-6. DigitalOcean Access Token for Machine

You are almost set. You just need to download the *docker-machine* binary. Go to the documentation site and choose the correct binary for your local computer architecture. For example on OSX:

```
$ wget https://github.com/docker/machine/releases/download/v0.1.0/docker-machine_darwin-amd64
$ mv docker-machine_darwin-amd64 docker-machine
$ chmod +x docker-machine
$ ./docker-machine --version
docker-machine version 0.1.0
```

With the environment variable `DIGITALOCEAN_ACCESS_TOKEN` set, you can create your remote Docker host with:

```
$ ./docker-machine create -d digitalocean foobar
INFO[0000] Creating SSH key...
INFO[0001] Creating Digital Ocean droplet...
INFO[0005] Waiting for SSH...
INFO[0072] Configuring Machine...
INFO[0117] "foobar" has been created and is now the active machine.
INFO[0117] To point your Docker client at it, run this in your shell: $(docker-machine env foobar)
```

If you go back to your DigitalOcean dashboard, you will see that a SSH Key has been created, as well as a new droplet.

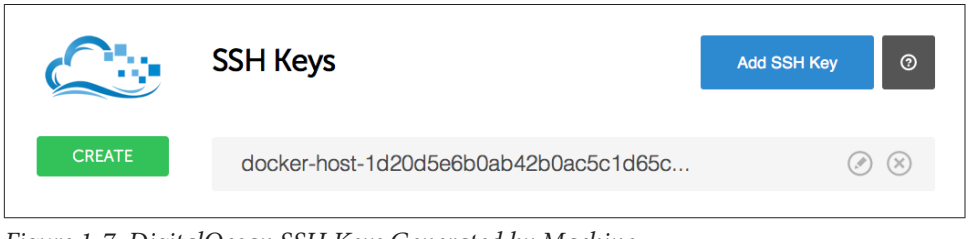


Figure 1-7. DigitalOcean SSH Keys Generated by Machine

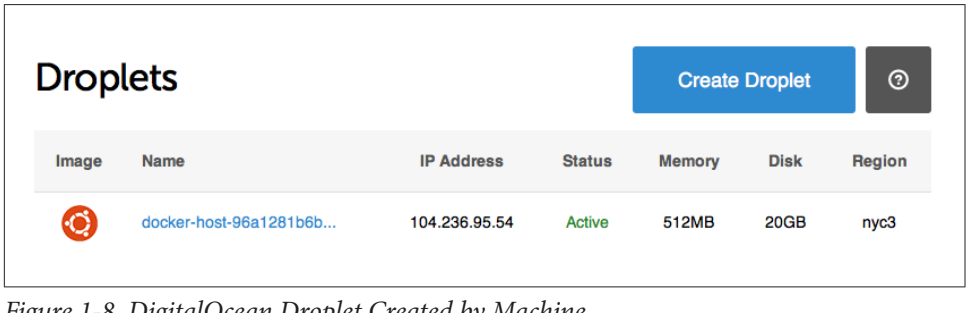


Figure 1-8. DigitalOcean Droplet Created by Machine

To configure your local Docker client to use this remote Docker host, you execute the sub-shell command that was listed in the output of creating the machine.

```
$ $(docker-machine env foobar)
$ docker ps
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS    PORTS    NAMES
```

Enjoy Docker running remotely on a DigitalOcean droplet created with *Docker Machine*.

Discussion

If not specified at the command line, *Machine* will look for a `DIGITALOCEAN_IMAGE`, `DIGITALOCEAN_REGION` and `DIGITALOCEAN_SIZE` environment variables. By default they are set to *docker*, *nyc3* and *512mb* respectively.

The `docker-machine` binary lets you create multiple machines, on multiple providers. You also have the basic management capabilities: start, stop, rm etc.

```
$ ./docker-machine
...
COMMANDS:
   active       Get or set the active machine
   create       Create a machine
   config       Print the connection config for machine
   inspect      Inspect information about a machine
   ip           Get the IP address of a machine
   kill         Kill a machine
   ls           List machines
   restart      Restart a machine
   rm           Remove a machine
   env          Display the commands to set up the environment for the Docker client
   ssh          Log into or run a command on a machine with SSH
   start        Start a machine
   stop         Stop a machine
   upgrade      Upgrade a machine to the latest version of Docker
   url          Get the URL of a machine
   help, h      Shows a list of commands or help for one command
```

For instance you can list the machine you created previously, obtain its IP address and even connect to it via SSH.

```
$ ./docker-machine ls
NAME     ACTIVE   DRIVER         STATE     URL                         SWARM
foobar   *        digitalocean   Running   tcp://45.55.161.171:2376
$ ./docker-machine ip foobar
45.55.161.171
$ ./docker-machine ssh foobar
Welcome to Ubuntu 14.04.1 LTS (GNU/Linux 3.13.0-43-generic x86_64)
...
```

```
Last login: Mon Mar 16 09:02:13 2015 from ...
root@foobar:~#
```

Before you are done with this recipe, do not forget to delete the machine you created:

```
$ ./docker-machine rm foobar
```

See Also

- Official documentation

1.6 Running `Hello World` in Docker

Problem

You have access to a Docker host and want to run your first container, executing `Hello World` in it.

Solution

Typing `docker` at the prompt will return the usage of the docker command:

```
$ docker
Usage: docker [OPTIONS] COMMAND [arg...]

A self-sufficient runtime for linux containers.

...

Commands:
    attach    Attach to a running container
    build     Build an image from a Dockerfile
    commit    Create a new image from a container's changes
...
    rm        Remove one or more containers
    rmi       Remove one or more images
    run       Run a command in a new container
    save      Save an image to a tar archive
    search    Search for an image on the Docker Hub
    start     Start a stopped container
    stop      Stop a running container
    tag       Tag an image into a repository
    top       Lookup the running processes of a container
    unpause   Unpause a paused container
    version   Show the Docker version information
    wait      Block until a container stops, then print its exit code
```

We have already seen the `docker ps` command which list all running containers. They are many more commands that we will explore in the recipes of this book. To get started we want to run a container. Let's get straight to it and use `docker run`.

```
$ docker run busybox echo hello world
Unable to find image 'busybox' locally
busybox:latest: The image you are pulling has been verified
511136ea3c5a: Pull complete
df7546f9f060: Pull complete
e433a6c5b276: Pull complete
e72ac664f4f0: Pull complete
Status: Downloaded newer image for busybox:latest
hello world
```

Containers are based on images. An image needs to be passed to the `docker run` command. In the example above we specified an image called `busybox`. Docker did not have this image locally and *pulled* it from a public registry. Once the image was pulled, it started a container based on it and executed the `echo hello world` command. Congratulations your ran your first container.

Discussion

If we list the running containers, we will see that they are none running. That's because as soon as the container did its job (i.e echoing hello world) it stopped. However it is not totally gone and we can see it with the `docker ps -a` command:

```
$ docker ps
CONTAINER ID    IMAGE    COMMAND      CREATED   STATUS     PORTS        NAMES
$ docker ps -a
CONTAINER ID    IMAGE           COMMAND            ...    PORTS        NAMES
8f7089b187e8    busybox:latest  "echo hello world" ...                 thirsty_morse
```

We see that the container has an ID (8f7089b187e8) and an image (busybox:latest) as well as a name, and we see the command that it ran. You can remove permanently this container with `docker rm 8f7089b187e8`. The image that we used was downloaded locally and `docker images` returns it.

```
$ docker images
REPOSITORY       TAG             IMAGE ID        CREATED         VIRTUAL SIZE
busybox          latest          e72ac664f4f0    9 weeks ago     2.433 MB
```

If no running or stopped containers are using this image, you can remove it with `docker rmi busybox`.

Running `echo` is fun but getting a terminal session within a container is even better. Try to run a container that executes `/bin/bash`. You will need to use the `-t` and `-i` options to get a proper interactive session and while we are at it, let's use an Ubuntu image.

```
$ docker run -t -i ubuntu:14.04 /bin/bash
Unable to find image 'ubuntu:14.04' locally
ubuntu:14.04: The image you are pulling has been verified
01bf15a18638: Pull complete
30541f8f3062: Pull complete
e1cdf371fbde: Pull complete
9bd07e480c5b: Pull complete
511136ea3c5a: Already exists
Status: Downloaded newer image for ubuntu:14.04
root@6f1050d21b41:/#
```

We see that docker pulled the Ubuntu:14.04 image composed of several layers, and we got a session as root within a container. The prompt gives us the ID of the container. As soon as we exit this terminal, the container will stop running just like our first *hello world* example.

If you skipped the first few recipes on installing Docker. You should try the web emulator. It will give you a 10 minute tour of Docker and you will get your first practice with it.

1.7 Running a Docker Container in Detached Mode

Problem

You know how to run a container interactively but would like to run a service in the background.

Solution

Use the `-d` option of `docker run` and expose a port with the `-p` option.

To try this we are going to run a simple HTTP server with Python in a python:2.7 Docker image pulled from Docker Hub (see also Recipe 2.8).

```
$ docker run -d -p 1234:1234 python:2.7 python -m SimpleHTTPServer 1234
```

You will see that the container keeps running and that it is exposing its port 1234 to the Docker host 1234 port

```
$ docker ps
CONTAINER ID  IMAGE       COMMAND                ...  PORTS                    NAMES
0fae2d2e8674  python:2.7  "python -m SimpleHTT   ...  0.0.0.0:1234->1234/tcp suspicious_pike
```

If you open your browser at the IP of your Docker host on port 1234, you will see the listing of the root directory inside your container.

Discussion

The `-d` option made the container run in the background. You can connect to the container by using the `exec` command and running a bash shell.

```
$ docker exec -ti 9d7cebd75dcf /bin/bash
root@9d7cebd75dcf:/# ps -ef | grep python
root         1     0  0 15:42 ?        00:00:00 python -m SimpleHTTPServer 1234
```

`docker exec` was introduced in Docker 1.3. Upgrade to Docker 1.3+ if you want to use it.

While you specified a container port to host port mapping with the `-p` option, you could have let Docker decide of this port forwarding rule using the `P` option and exposing port 1234 with the `--expose` option. The mapping would be visible from `docker ps` or directly from `docker port`:

```
$ docker run -d -P --expose=1234 python:2.7 python -m SimpleHTTPServer 1234
$ docker port 317451b6eab3
1234/tcp -> 0.0.0.0:49153
```

Lots of other options are available for `docker run`. Try to experiment by specifying a name for the container, changing the working directory of the container, setting an environment variables and so on.

See Also

- Docker run reference

1.8 Creating, Starting, Stopping, Removing Containers.

Problem

You know how to start containers, you can also run them in detached mode, you would like to learn the basic commands to manage the entire lifecycle of a container.

Solution

Use the `create`, `start`, `stop`, `kill` and `rm` commands of the Docker cli. Find the appropriate usage of each command with the `-h` or the `--h` option or simply by typing the command with no arguments (e.g `docker create`)

Discussion

In Recipe 1.7 we started a container with `docker run`, the container started automatically. You can also stage a container with the `docker create` command. Using the same example of running a simple HTTP server, the only difference will be that we will not specify the `-d` option. Once staged, the container will need to be started with `docker start`

```
$ docker create -P --expose=1234 python:2.7 python -m SimpleHTTPServer 1234
a842945e2414132011ae704b0c4a4184acc4016d199dfd4e7181c9b89092de13
$ docker ps -a
CONTAINER ID  IMAGE       COMMAND              CREATED        ...  NAMES
a842945e2414  python:2.7  "python -m SimpleHTT 8 seconds ago  ...  fervent_hodgkin
$ docker start a842945e2414
a842945e2414
$ docker ps
CONTAINER ID  IMAGE       COMMAND              ....  PORTS                    NAMES
a842945e2414  python:2.7  "python -m SimpleHTT ....  0.0.0.0:49154->1234/tcp fervent_hodgkin
```

To stop a running container, you have the choice between `docker kill` which will send a SIGKILL signal to the container or `docker stop` which will send a SIGTERM and after a grace period will send a SIGKILL. The end result will be that the container is stopped and is not listed in the list of running containers `docker ps`. However the container has not yet disappeared (i.e the filesystem of the container is still there), you could restart it `docker restart` or remove it forever with `docker rm`.

```
$ docker restart a842945e2414
a842945e2414
$ docker ps
CONTAINER ID    IMAGE       COMMAND              ... PORTS                    NAMES
a842945e2414    python:2.7  "python -m SimpleHTT ... 0.0.0.0:49155->1234/tcp  fervent_hodgkin
$ docker kill a842945e2414
a842945e2414
$ docker rm a842945e2414
a842945e2414
$ docker ps -a
CONTAINER ID    IMAGE       COMMAND     CREATED     STATUS      PORTS       NAMES
```

If you have a lot of stopped containers that you would like to remove, use a subshell to do it in one command. The `-q` option of `docker ps` will return only the containers IDs.

```
$ docker rm $(docker ps -a -q)
```

1.9 Sharing Host Data With Containers

Problem

You have some data on your host that you would like to make available in a container.

Solution

Use the `-v` option of `docker run` to mount a host volume into a container.

For example, to share the working directory of your host into a `/cookbook` directory in a container do:

```
$ ls
data
$ docker run -ti -v $PWD:/cookbook ubuntu:14.04 /bin/bash
root@11769701f6f7:/# ls /cookbook
data
```

If you create files or directories within the container, the changes will be written directly to the mounted host directory.

```
$ docker run -ti -v $PWD:/cookbook ubuntu:14.04 /bin/bash
root@44d71a605b5b:/# touch /cookbook/foobar
root@44d71a605b5b:/# exit
exit
$ ls -l foobar
-rw-r--r-- 1 root root 0 Mar 11 11:42 foobar
```

You can inspect your mount mapping with the `docker inspect` command. See Recipe 9.1 for more information about `inspect`.

```
$ docker inspect -f  44d71a605b5b
map[/cookbook:/home/vagrant]
```

Discussion

- Understanding Volumes
- Data container
- Docker volumes
- Official Docker documentation

1.10 Sharing Data Between Containers

Problem

You know how to mount a host volume into a running container, but you would like to share a volume defined in a container with other containers. This would have the benefit of letting Docker manage the volumes and keep with the principle of single responsibility.

Solution

Use Data containers. In Recipe 1.9 we saw how to mount a host volume into a container. The `-v` option of `docker run` was used, specifying a host volume and a path within a container to mount that volume to. If the host path is omitted we create a so-called data container. The volume specified is created inside the container as a read-write filesystem not layered on top of the read-only layers used to create the container image. Docker manages that filesystem but you can read and write to it from the host. Let's illustrate this.

```
$ docker run -ti -v /cookbook ubuntu:14.04 /bin/bash
root@b5835d2b951e:/# touch /cookbook/foobar
root@b5835d2b951e:/# ls cookbook/
foobar
root@b5835d2b951e:/# exit
exit
$ docker inspect -f  b5835d2b951e
map[/cookbook:/var/lib/docker/vfs/dir/0262f322bd19f61a1fd56c3183c1f7b9358fa4b3a8226556b8487a86b523
$ sudo ls /var/lib/docker/vfs/dir/0262f322bd19f61a1fd56c3183c1f7b9358fa4b3a8226556b8487a86b523ec38
foobar
```

When the container was started, Docker created the `/cookbook` directory, from within the container you can read and write to this volume. Once you exit the container you can use `inspect` (see Recipe 9.1) to know where the volume has been created on the host. Docker created it under `/var/lib/docker/vfs/dir`. From the host you can read and write to it. Changes will persist and be available if you restart the container:

```
$ sudo touch /var/lib/docker/vfs/dir/0262f322bd19f61a1fd56c3183c1f7b9358fa4b3a8226556b8487a86b523e
$ docker start b5835d2b951e
$ docker exec -ti b5835d2b951e /bin/bash
root@b5835d2b951e:/# ls /cookbook
foobar  foobar2
```

To share this data volume with other containers, use the `--volumes-from` option. Create a data container, then start another container that will mount the volume from the data container.

```
$ docker run -v /data --name data ubuntu:14.04
$ docker ps
CONTAINER ID        IMAGE       COMMAND     CREATED       STATUS       PORTS        NAMES
$ docker inspect -f  data
map[/data:/var/lib/docker/vfs/dir/4ee1d9e3d453e843819c6ffca7b05d7f7431dcbf90195f392288214a1ddb3ecc
```

Note that the *Data* container is not running. Still, the volume mapping exists and the volume has persisted in `/var/lib/docker/vfs/dir`. You can only remove the container and the volume with `docker rm -v data`. If you do not use the `rm -v` option to delete containers and their volumes, you will end up with lots of orphaned volumes.

Even though the data container is not running you can mount the volume from it with `--volumes-from` option.

```
$ docker run -ti --volumes-from data ubuntu:14.04 /bin/bash
root@b94a006377c1:/# touch /data/foobar
root@b94a006377c1:/# exit
exit
$ sudo ls /var/lib/docker/vfs/dir/4ee1d9e3d453e843819c6ffca7b05d7f7431dcbf90195f392288214a1ddb3ecc
foobar
```

See Also

- Understanding Volumes
- Data container
- Docker volumes
- Official Docker documentation
- The Ah Ah moment of Docker volumes.

1.11 Copying Data To And From Containers

Problem

You have a running container started without any volumes configuration, but you would like to copy files in and out of the container.

Solution

Use the `docker cp` command to copy files from a running container to the Docker host.

Let's first start a container that will just sleep. We can enter the container and create a file manually.

```
$ docker run -d --name testcopy ubuntu:14.04 sleep 360
$ docker exec -ti testcopy /bin/bash
root@b81793e9eb3e:/# cd /root
root@b81793e9eb3e:~# echo I am in the container > file.txt
root@b81793e9eb3e:~# exit
```

Now to get the file that we just created in the container back in the host, `docker cp` does the work.

```
$ docker cp testcopy:/root/file.txt .
$ cat file.txt
I am in the container
```

To copy from the host to the container, we can use a combination of `docker exec` and some shell redirection.

```
$ echo I am in the host > host.txt
$ docker exec -i testcopy sh -c 'cat > /root/host.txt' < host.txt
$ docker exec -i testcopy sh -c 'cat /root/host.txt'
I am in the host
```

To copy from one container to another container, it is a matter of combining the two methods by temporarily saving the files on the host. For example if we want to transfer `/root/file.txt` from two running containers *c1* and *c2*:

```
$ docker cp c1:/root/file.txt .
$ docker exec -i c2 sh -c 'cat > /root/file.txt' < file.txt
```

See Also

- Original idea for this recipe from Grigoriy Chudnov

1.12 Managing and Configuring the Docker Daemon

Problem

You would like to stop, start, restart the Docker Daemon. Additionally, you would like to configure it in specific ways, potentially changing things like the path to the docker binary or using a different network bridge.

Solution

Use the `docker` init script to manage the Docker daemon. On most systems it is located at `/etc/init.d/docker.io` (for `docker.io` package on Ubuntu) or `/etc/init.d/`

`docker` for the latest versions of Docker. Like most other init services it can be managed via the `service` command. The Docker daemon runs as `root`.

```
# service docker status
docker start/running, process 2851
# service docker stop
docker stop/waiting
# service docker start
docker start/running, process 3119
```

The configuration file is located in `/etc/default/docker`. On Ubuntu systems, all configuration variables are commented out. The `/etc/default/docker` file looks like this:

```
# Docker Upstart and SysVinit configuration file

# Customize location of Docker binary (especially for development testing).
#DOCKER="/usr/local/bin/docker"

# Use DOCKER_OPTS to modify the daemon startup options.
#DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4"

# If you need Docker to use an HTTP proxy, it can also be specified here.
#export http_proxy="http://127.0.0.1:3128/"

# This is also a handy place to tweak where Docker's temporary files go.
#export TMPDIR="/mnt/bigdrive/docker-tmp"
```

For example, if you wanted to configure the daemon to listen on a TCP socket to enable remote API access, you would edit this file as explained in Recipe 4.8.

Check for CentOS … Recipes not finished and probably misplaced in the book.

Discussion

1.13 Running a Wordpress Blog Using Two Linked Containers

Problem

You want to run a Wordpress site with containers, but you do not want to run the Mysql database in the same container as Wordpress. Keeping the concept of separation of concerns in mind and decoupling the various components of an application as much as possible.

Solution

You start two containers. One running Wordpress using the official image from the Docker hub, and one running the Mysql database. The two containers are linked using the `--link` option of the Docker cli.

Start by pulling the latest images for *https://registry.hub.docker.com//wordpress/[Wordpress] and https://registry.hub.docker.com//*mysql/[Mysql]:

```
$ docker pull wordpress:latest
$ docker pull mysql:latest
$ docker images
REPOSITORY      TAG             IMAGE ID        CREATED         VIRTUAL SIZE
mysql           latest          9def920de0a2    4 days ago      282.9 MB
wordpress       latest          93acfaf85c71    8 days ago      472.8 MB
```

Start a Mysql container, give it a name via the `--name` cli option, and set the `MYSQL_ROOT_PASSWORD` via an environment variable:

```
$ docker run --name mysqlwp -e MYSQL_ROOT_PASSWORD=wordpressdocker -d mysql
```

By not specifying the tag for the `mysql` image, it automatically chose the `latest` tag, which is the one we downloaded specifically. The container was daemonized with the `-d` option.

You can now run a wordpress container based on the `wordpress:latest` image. It will be *linked* to the Mysql container using the `--link` option, which means that Docker will automatically set up the networking so that the ports exposed by the Mysql container are reachable inside the Wordpress container.

```
$ docker run --name wordpress --link mysqlwp:mysql -p 80:80 -d wordpress
```

Both containers should be running in the background, with port 80 of the wordpress container mapped to port 80 of the host.

```
$ docker ps
CONTAINER ID    IMAGE             COMMAND                CREATED              ...
e1593e7a20df    wordpress:latest  "/entrypoint.sh apac   About a minute ago   ...
d4be18e33153    mysql:latest      "/entrypoint.sh mysq   5 minutes ago        ...

...             STATUS              PORTS                NAMES
...                 Up About a minute   0.0.0.0:80->80/tcp   wordpress
...                 Up 5 minutes        3306/tcp             mysqlwp
```

Open a browser at *http://<ip _of _host>* and it should show the Wordpress installation screen with the language selection window. If you go through the Wordpress setup, you will then have a fully functional Wordpress site running with two linked containers.

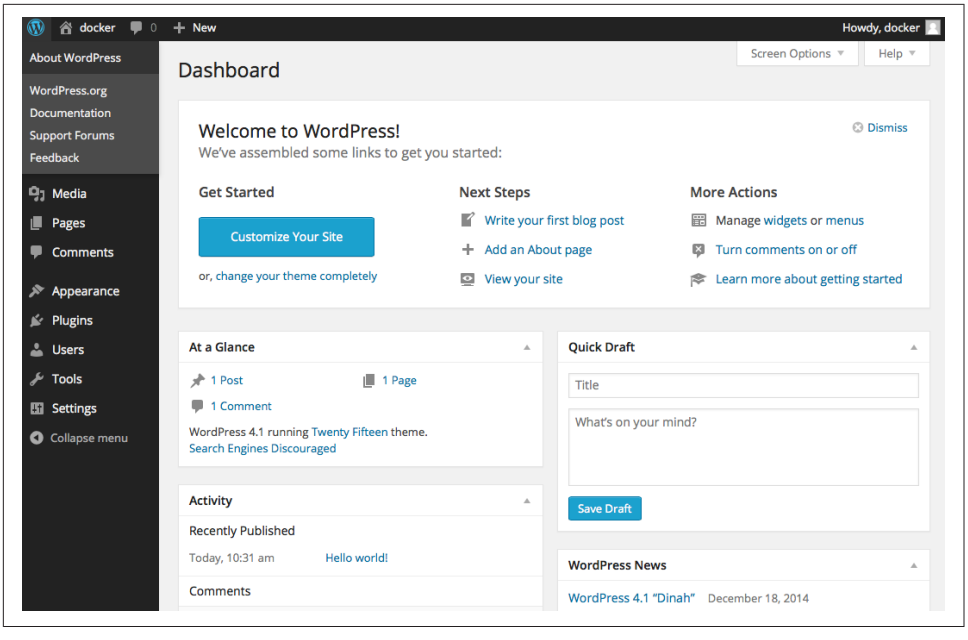


Figure 1-9. Working Wordpress Site Within Containers

Discussion

The two images for Wordpress and Mysql are official images maintained by the Wordpress and Mysql communities. Each page on the Docker Hub provides additional documentation for configuration of containers started with those images.

Do not forget to read: - The Wordpress image *https://registry.hub.docker.com//wordpress/[documentation]*. - *The Mysql image https://registry.hub.docker.com/*/mysql/[documentation].

Of interest is that we can create a database and a user with appropriate privileges to manipulate that database using a few environment variables `MYSQL_DATABASE`, `MYSQL_USER` and `MYSQL_PASSWORD`. In the example above, Wordpress was run as the *root* mysql user and this is far from best practice. It would be better to create a *wordpress* database and a user for it, like so:

```
$ docker run --name mysqlwp -e MYSQL_ROOT_PASSWORD=wordpressdocker
                            -e MYSQL_DATABASE=wordpress
                            -e MYSQL_USER=wordpress
                            -e MYSQL_PASSWORD=wordpresspwd
                            -d mysql
```

If you need to remove existing containers:

```
$ docker stop $(docker ps -q)
$ docker rm $(docker ps -aq)
```

Once the database container is running, you run the wordpress container and specify the database tables you defined:

```
$ docker run --name wordpress --link mysqlwp:mysql -p 80:80
                              -e WORDPRESS_DB_NAME=wordpress
                              -e WORDPRESS_DB_USER=wordpress
                              -e WORDPRESS_DB_PASSWORD=wordpresspwd
                              -d wordpress
```

1.14 Backing up a Database Running in a Container

Problem

You are using a Mysql image to provide a database service. You need to backup this database for data persistency.

Solution

There are several backup strategies possible alone or in combination. The two main concepts with containers are that you can execute a command inside a container running in the background and that you can also mount a host volume into the container. In this recipe we will see how to:

- Mount a volume from the docker host into the mysql container.
- Use the `docker exec` command to call `mysqldump`.

Starting from the recipe Recipe 1.13, where we setup a Wordpress site using two linked containers, we are going to modify the way we start the *mysql* container. Once the containers are started and that we have a fully functional Wordpress site, you can stop the containers, which stops your application. At that point the containers have actually not been removed entirely yet and the data in the database is still accessible. However as soon as you remove the containers (e.g `docker rm $(docker ps -aq)`) all data will be lost.

A way to keep the data, even when containers are removed is to mount a volume from your Docker host inside a container. If you look at the Dockerfile used to build the Mysql image, you sill see a reference to `VOLUME /var/lib/mysql`. This means that when we start a container based on this image we can bind mount a host directory to this mount point inside the container. Let's do it:

```
$ docker run --name mysqlwp -e MYSQL_ROOT_PASSWORD=wordpressdocker
                            -e MYSQL_DATABASE=wordpress
                            -e MYSQL_USER=wordpress
                            -e MYSQL_PASSWORD=wordpresspwd
                            -v /home/docker/mysql:/var/lib/mysql
                            -d mysql
```

Note the `-v /home/docker/mysql:/var/lib/mysql` line which does this mount. After doing the wordpress configuration, `/home/docker/mysql` directory on the host is populated:

```
$ ls mysql/
auto.cnf  ibdata1  ib_logfile0  ib_logfile1  mysql  performance_schema  wordpress
```

To get a dump of the entire Mysql database, simply use the `docker exec` command to run `mysqldump` inside the container

```
$ docker exec mysqlwp mysqldump --all-databases
                              --password=wordpressdocker > wordpress.backup
```

You can then use the traditional techniques for backup and recovery of the database. For instance in the cloud, you might want to use an Elastic Block Store (e.g AWS EBS) mounted on an instance and then mounted inside a container. You can also keep your mysql dumps inside an Elastic Storage (e.g AWS S3)

Discussion

While this recipe uses Mysql, same techniques are valid for Postgres and other databases. If you use the Postgres image from Docker hub, you can also see in the Dockerfile that a volume is created (i.e `VOLUME /var/lib/postgresql/data`)

1.15 Using `Supervisor` to Run Wordpress in a Single Container

Problem

While you know how to link containers together (See Recipe 1.13), you would like to run all services needed for your application in a single container. Specifically for running Wordpress, you would like to run `Mysql` and `httpd` at the same time in a container. However Docker executes foreground processes, therefore you need to figure out a way to run multiple "foreground" processes simultaneously.

Solution

Use Supervisor to monitor and run both Mysql and httpd. Supervisor is not an init system, but is meant to control multiple processes and is ran like any other program.

This recipe is an example of using `Supervisor` to run multiple processes in a container. It can be used as the basis to run any number of services via a single Docker image (e.g SSH, Nginx). The Wordpress setup detailed below is a minimum viable setup, not meant for production use.

The example files can be found at `https://github.com/how2dock/docbook/tree/master/ch01/supervisor`. It consists of a Vagrantfile to start a virtual machine that runs Docker, a Dockerfile that defines the image being created, a supervisor configuration file `supervisord.conf` and a Wordpress configuration file `wp-config.php`.

If you do not want to use Vagrant, you can simply take the Dockerfile, `supervisord` and Wordpress configuration files and set things up on your own Docker host.

To run Wordpress, you will need to install Mysql, Apache2 (i.e `httpd`), Php and grab the latest Wordpress release. You will need to create a database for Wordpress. In the configuration file used in this recipes, the Wordpress database user is `root`, its password is `root` and the database is `wordpress`. Change these settings to your liking in the `wp-config.php` file and edit the Dockerfile accordingly.

A Dockerfile is a manifest that describes how a Docker image is built, it will be described in details in follow-on chapters. If this is your first use of a Dockerfile you can use it as is and come back to it later on (see Recipe 2.3 for an introduction to Dockerfile).

```
FROM ubuntu:14.04

RUN apt-get update && apt-get -y install \
    apache2 \
    php5 \
    php5-mysql \
    supervisor \
    wget

RUN echo 'mysql-server mysql-server/root_password password root' | debconf-set-selections && \
    echo 'mysql-server mysql-server/root_password_again password root' | debconf-set-selections

RUN apt-get install -qqy mysql-server

RUN wget http://wordpress.org/latest.tar.gz && \
    tar xzvf latest.tar.gz && \
    cp -R ./wordpress/* /var/www/html && \
    rm /var/www/html/index.html

RUN (/usr/bin/mysqld_safe &); sleep 5; mysqladmin -u root -proot create wordpress
```

```
COPY wp-config.php /var/www/html/wp-config.php
COPY supervisord.conf /etc/supervisor/conf.d/supervisord.conf

EXPOSE 80

CMD ["/usr/bin/supervisord"]
```

Supervisor is configured via the `supservisord.conf` file like so:

```
[supervisord]
nodaemon=true

[program:mysqld]
command=/usr/bin/mysqld_safe
autostart=true
autorestart=true
user=root

[program:httpd]
command=/bin/bash -c "rm -rf /run/httpd/* && /usr/sbin/apachectl -D FOREGROUND"
```

Two programs are defined to be run and monitored: `mysqld` and `httpd`. Each program can use a number of options like `autorestart` and `autostart`. The most important directive is `command` which defines how to run each program. With this configuration, a Docker container only needs to run a single foreground process `supervisord`. Hence the line in the Dockerfile `CMD ["/usr/bin/supervisord"]`.

On your Docker host, build the image and start a background container off of it. If you are using the Vagrant virtual machine started via the example files do:

```
$ cd /vagrant
$ docker build -t wordpress .
$ docker run -d -p 80:80 wordpress
```

Port forwarding will be setup between your host and the docker container for port 80. You will just need to open your browser `http://<IP_OF_DOCKER_HOST>` and configure Wordpress.

Discussion

While using Supervisor to run multiple application services in a single container works perfectly. It is better to use multiple containers. It promotes the isolation of concerns using containers and helps create a microservices based design for your application. Ultimately this will help with scale and resilience.

See Also

- Supervisor documentation.

- Docker supervisor article.

CHAPTER 2

Image Creation and Sharing

This chapter consists of recipes focused on creating and sharing Docker images. You can send me suggestions at *how2dock@gmail.com*

2.1 Keeping Changes Made to a Container by Committing to an Image.

Problem

After making some changes inside a container, you decide that you would like to keep those changes. You do not want to loose those changes once you exit or stop the container and you would like to re-use this type of container as a basis for others.

Solution

Commit the changes that you made using the `docker commit` command and define a new image.

Let's start a container with an interactive bash shell and update the packages in it:

```
$ docker run -t -i ubuntu:14.04 /bin/bash
root@69079aaaaab1:/# apt-get update
```

When we exit the container it stops running, but it is still available to you until you remove it entirely with `docker rm`. So before we do this, we can commit the changes made to the container and create a new image `ubuntu:update`. The name of the image is `ubuntu` and we added a tag `update` (see Recipe 2.5) to mark the difference from the `ubuntu:latest` image.

```
$ docker commit 69079aaaaab1 ubuntu:update
13132d42da3cc40e8d8b4601a7e2f4dbf198e9d72e37e19ee1986c280ffcb97c
$ docker images
REPOSITORY          TAG             IMAGE ID        CREATED         VIRTUAL SIZE
ubuntu              update          13132d42da3c    5 days ago      213 MB
...
```

You can now safely remove the stopped container and you will be able to start new ones based on the `ubuntu:update` image.

Discussion

We can inspect the changes that have been made inside this container with the `docker diff` command

```
$ docker diff 69079aaaaab1
C /root
A /root/.bash_history
C /tmp
C /var
C /var/cache
C /var/cache/apt
D /var/cache/apt/pkgcache.bin
D /var/cache/apt/srcpkgcache.bin
C /var/lib
C /var/lib/apt
C /var/lib/apt/lists
...
```

Where A means that the file or directory listed was added, C means that there was a change made and D means that it was deleted.

See Also

- `docker commit` reference
- `docker diff` reference

2.2 Saving Images and Containers as Tar Files for Sharing.

Problem

You have created some images or have some containers that you would like to keep and share with your collaborators.

Solution

Use the Docker CLI `save` and `load` command to create a tar ball from a previously created image or use the Docker CLI `import` and `export` command for containers.

Let's start with a stop container and export it to a new tar ball:

```
$ docker ps -a
CONTAINER ID    IMAGE            COMMAND     CREATED         STATUS                     PORTS   NA
77d9619a7a71    ubuntu:14.04    "/bin/bash"  10 seconds ago  Exited (0) 2 seconds ago           hi
$ docker export 77d9619a7a71 > update.tar
$ ls
update.tar
```

You could off course commit this container as a new image (see Recipe 2.1) locally but you could also use the Docker `import` command:

```
$ docker import - update < update.tar
157bcbb5fdfce0e7c10ef67ebdba737a491214708a5f266a3c74aa6b0cfde078
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
update              latest              157bcbb5fdfc        8 seconds ago       188.1 MB
```

If you wanted to share this image with one of your collaborators you could upload a zip version of this tar ball on a webserver and let your collaborator download it and use the `import` command on his Docker host.

If you would rather deal with images that you have already committed, you can use the `load` and `save` commands.

```
$ docker save -o update1.tar update
$ ls -l
total 385168
-rw-rw-r-- 1 vagrant vagrant 197206528 Jan 13 14:13 update1.tar
-rw-rw-r-- 1 vagrant vagrant 197200896 Jan 13 14:05 update.tar
$ docker rmi update
Untagged: update:latest
Deleted: 157bcbb5fdfce0e7c10ef67ebdba737a491214708a5f266a3c74aa6b0cfde078
$ docker load < update1.tar
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
update              latest              157bcbb5fdfc        5 minutes ago       188.1 MB
ubuntu              14.04               8eaa4ff06b53        12 days ago         192.7 MB
```

Discussion

The two methods are very similar, the difference is that saving an image will keep its history, exporting a container will squash its history.

2.3 Writing your First Dockerfile

Problem

Running a container in interactive mode, making changes to it and then committing these changes to create a new image works well (see Recipe 2.1). However, you want to automate building your image and share your build steps with others.

Solution

To automate building a Docker image, you decribe the building steps in a Docker manifesto called the `Dockerfile`. This text file uses a set of instructions used to describe which base image the new container is based on, what steps need to be taken to install various dependencies and applications, what files need to be present in the image, how they are made available to a container, what ports should be exposed, what command should run when a container starts as well as a few other things.

To illustrate this, let's write our first Dockerfile. The resulting image will allow us to create a container that executes the `/bin/echo` command. Create a text file called `Dockerfile` in your working directory and write the following content in it:

```
FROM ubuntu:14.04

ENTRYPOINT ["/bin/echo"]
```

The FROM instruction tells us which image to base the new image off of. Here we choose the ubuntu:14.04 image from the Official Ubuntu repository in Docker hub. The ENTRYPOINT instruction tells us which command to run when a container based on this image is started. To build the image, issue a `docker build .` at the prompt like so:

```
$ docker build .
Sending build context to Docker daemon  2.56 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu:14.04
 ---> 9bd07e480c5b
Step 1 : ENTRYPOINT /bin/echo
 ---> Running in da3fa01c973a
 ---> e778362ca7cf
Removing intermediate container da3fa01c973a
Successfully built e778362ca7cf
$ docker images
REPOSITORY          TAG         IMAGE ID            CREATED             VIRTUAL SIZE
<none>              <none>      e778362ca7cf        4 days ago          192.7 MB
ubuntu              14.04       9bd07e480c5b        10 days ago         192.7 MB
```

We are now ready to run this container, specifying the image ID of the freshly built image and passing an argument to it `Hi Docker !`.

```
$ docker run e778362ca7cf Hi Docker !
Hi Docker !
```

Amazing, we ran `echo` in a container ! A container was started using the image that we built from this two line Dockerfile. The container ran and executed the command defined by the ENTRYPOINT instruction. Once this command was finished the container job was done and it exited. If you run it again without passing an argument, nothing is echoed.

```
$ docker run e778362ca7cf
```

This image is not very useful, since the ENTRYPOINT value cannot be overwritten this image will only be able to run `echo`. If you wanted to do something else with this container, you could instead use the CMD instruction in a Dockerfile like so:

```
FROM ubuntu:14.04

CMD ["/bin/echo" , "Hi Docker !"]
```

Let's build it and run it:

```
$ docker build .
...
$ docker run eff764828551
Hi Docker !
```

It looks the same but if we pass a new executable as argument to the docker run command, this command will be executed instead of the `/bin/echo` defined in the Dockerfile.

```
$ docker run eff764828551 /bin/date
Thu Dec 11 02:49:06 UTC 2014
```

Discussion

A Dockerfile is a text file that represents the way a Docker image is built and what happens when a container is started with this image. Starting with three simple instructions you can build a fully functioning container: FROM, ENTRYPOINT, CMD. Of course this is quite limited in this recipe. Read the Docker file reference to learn about all the other instructions, or go to Recipe 2.4 for a more detailed example.

The CentOS project maintains a large set of Dockerfile examples. You should check out this repository and run a few of their examples to get more familiar with Dockerfile files.

Remember that CMD can be overwritten by an argument while ENTRYPOINT cannot. Also we saw that once a command is finished the container exits. A process that

we want to run in a container needs to run in the foreground, otherwise the container will stop.

Once our first build was done, a new image was created with in our case, the ID e778362ca7cf. Note that there was no repository or tag defined because we did not specify any. We can rebuild the image with the repository cookbook as name and the tag `hello` using the `-t` option of `docker build`. As long as you are doing this locally the choice of repository and tag is up to you, but once you start publishing this image into a registry, you will need to follow a naming convention.

```
$ docker build -t cookbook:echo .
$ docker images
REPOSITORY          TAG         IMAGE ID            CREATED             VIRTUAL SIZE
cookbook            echo         e778362ca7cf        4 days ago          192.7 MB
ubuntu              14.04       9bd07e480c5b        10 days ago         192.7 MB
```

The docker `build` command has couple options to deal with intermediate containers

```
$ docker build -h

Usage: docker build [OPTIONS] PATH | URL | -

Build a new image from the source code at PATH

  --force-rm=false     Always remove intermediate containers...
  --no-cache=false     Do not use cache when building the ...
  -q, --quiet=false    Suppress the verbose output generated...
  --rm=true            Remove intermediate containers after ...
  -t, --tag=""         Repository name (and optionally a tag)...
```

See Also

- Dockerfile reference
- Best practices for writing a Dockerfile
- The CentOS project maintains a large set of Dockerfiles.

2.4 Packaging a `Flask` Application inside a container

Problem

You have a web application built with the Python framework Flask, running in Ubuntu 14.04. You want to run this application in a container.

Solution

As an example we are going to use the simple Hello World application defined by the following Python script

```
#!/usr/bin/env python

from flask import Flask
app = Flask(__name__)

@app.route('/hi')
def hello_world():
    return 'Hello World!'

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

To get this application running inside a Docker container, we need to write a Dockerfile that installs the pre-requisites needed to run the application using the RUN key and exposes the port that the application runs on using the EXPOSE key. We also need to move the application inside the container filesystem using the ADD key.

Our Dockerfile will be:

```
FROM ubuntu:14.04

RUN apt-get update
RUN apt-get install -y python
RUN apt-get install -y python-pip
RUN apt-get clean all

RUN pip install flask

ADD hello.py /tmp/hello.py

EXPOSE 5000

CMD ["python","/tmp/hello.py"]
```

The RUN commands allow us to executed specific shell commands during the container image build time. Here we update the repository cache, we install Python as well as Pip and we install the Flask micro-framework.

To copy the application inside the container image we use the ADD command. It copies the file `hello.py` in the `/tmp/` directory.

The application uses port 5000, and we expose this port to the Docker host docker bridge.

Finally the CMD command, specifies that the container will run `python /tmp/hello.py` at run time.

What is left to do is to build the image with `docker build -t flask .`. This will create a `flask` docker image:

```
$ docker images
REPOSITORY          TAG         IMAGE ID            CREATED             VIRTUAL SIZE
flask               latest      d381310506ed        4 days ago          354.6 MB
cookbook            echo        e778362ca7cf        4 days ago          192.7 MB
ubuntu              14.04       9bd07e480c5b        10 days ago         192.7 MB
```

To run the application, we use the `-d` option of `docker run` which daemonizes the container, and we also use the `-P` option of `docker run` to let Docker choose a port on the Docker host that will be mapped to the exposed port specified in the Dockerfile (e.g 5000).

```
$ docker run -d -P flask
5ac72ed12a72f0e2bec0001b3e78f11660905d20f40e670d42aee292263cb890
sebimac:bottle sebgoa$ docker ps
CONTAINER ID        IMAGE               COMMAND                CREATED             STATUS
5ac72ed12a72        flask:latest        "python /tmp/hello.p   4 days ago          Up 2 seconds
```

We see that the container returned, it has been daemonized and we are not logged into an interactive shell. The PORTS shows a mapping between port 5000 of the container and port 49153 of the Docker host. A simple curl to `http://localhost:49153/hi` will return `Hello World`, or you can also open your browser to the same url.

If you are using `boot2docker` you will have to use the IP address of the bridge network, instead of `localhost`. If you do want to use `localhost` then add port forwarding rules in VirtualBox

Discussion

Since our Dockerfile specified a command to run via CMD, we do not need to specify a command after the name of the image to use. However since we used CMD and not ENTRYPOINT, we could override it and start the container in interactive mode, to explore it

```
$ docker run -t -i -P flask /bin/bash
root@fc1514ced93e:/# ls -l /tmp
total 4
-rw-r--r-- 1 root root 194 Dec  8 13:41 hello.py
root@fc1514ced93e:/#
```

2.5 Versioning an Image with Tags

Problem

You are creating multiple images and multiple versions of the same image. You would like to keep track of each image and its versions easily, instead of using an image ID.

Solution

Tag the image with the `docker tag` command. This allows you to rename an existing image, or create a new tag for the same name.

When we committed an image (see Recipe 2.1) we already used tags. The naming convention for images is that everything after a colon is a *TAG*.

A tag is actually optional. If a tag is not used, `docker` will implicitly try to use a tag called *latest*. If such a tag for the image being referenced, does not exist, then `docker` will throw an error.

For example, let's rename the *ubuntu:14.04* image to *foobar*. We will not specify a tag, just change the name, hence docker will use the *latest* tag automatically.

```
$ docker images
REPOSITORY          TAG             IMAGE ID        CREATED         VIRTUAL SIZE
ubuntu              14.04           9bd07e480c5b    12 days ago     192.7 MB

$ docker tag ubuntu foobar
2014/12/17 09:57:48 Error response from daemon: No such id: ubuntu

$ docker tag ubuntu:14.04 foobar

$ docker images
REPOSITORY          TAG             IMAGE ID        CREATED         VIRTUAL SIZE
foobar              latest          9bd07e480c5b    12 days ago     192.7 MB
ubuntu              14.04           9bd07e480c5b    12 days ago     192.7 MB
```

The first thing that we see in the example above, is that when we tried to tag the *ubuntu* image, Docker threw an error. That is because the *ubuntu* image only has a *14.04* tag and no *latest* tag. In our second attempt we specified the existing tag using a colon and the tagging was successull. Docker created a new *foobar* image and automatically added the *latest* tag. If we specify a tag by using a colon after the new name for the image, we get:

```
$ docker tag ubuntu:14.04 foobar:cookbook

$ docker images
REPOSITORY          TAG             IMAGE ID        CREATED         VIRTUAL SIZE
```

```
foobar              cookbook        9bd07e480c5b    12 days ago     192.7 MB
foobar              latest          9bd07e480c5b    12 days ago     192.7 MB
ubuntu              14.04           9bd07e480c5b    12 days ago     192.7 MB
```

All the images we used so far are local to the Docker host we used. But when we want to share these images through registries, we need to name them appropriately. Specifically we will need to follow the *USERNAME/NAME* convention when preparing an image for Docker Hub. When using a private registry we will need to specify the registry host, and optional username and the name of the image (i.e *REGISTRYHOST/USERNAME/NAME*). And of course we can still use a tag (i.e *:TAG*)

Discussion

Properly Tagging the image is an important part of sharing an image on Docker Hub (see Recipe 2.8) or using a private registry (see Recipe 2.9) The `docker tag` help information is pretty succint but shows the proper naming convention which references the proper namespace, be it local, on docker Hub or on a private registry.

```
$ docker tag -h

Usage: docker tag [OPTIONS] IMAGE[:TAG] [REGISTRYHOST/][USERNAME/]NAME[:TAG]

Tag an image into a repository

  -f, --force=false    Force
```

2.6 Migrating From Vagrant to Docker With the Docker Provider

Problem

You have been using Vagrant for your testing and development work and would like to re-use some of your Vagrantfiles to work with Docker.

Solution

Use the Vagrant Docker provider. You can keep writing Vagrant files to bring up new containers and develop your Dockerfile files.

An example Vagrantfile that uses the Docker provider is:

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

VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
```

```
  config.vm.provider "docker" do |d|
     d.build_dir = "."
  end

  config.vm.network "forwarded_port", guest: 5000, host: 5000

end
```

The `build_dir` option will simply look for a Dockerfile in the same directory as the Vagrantfile. Vagrant will then issue a `docker build` in turn, and start the container.

```
$ vagrant up --provider=docker
Bringing machine 'default' up with 'docker' provider...
==> default: Building the container from a Dockerfile...
    default: Sending build context to Docker daemon 8.704 kB
    default: Step 0 : FROM ubuntu:14.04
...
==> default: Creating the container...
    default:   Name: provider_default_1421147689
    default:  Image: 324f2babf057
    default: Volume: /vagrant/provider:/vagrant
    default:   Port: 5000:5000
    default:
    default: Container created: efe111afb8b9d3ff
==> default: Starting container...
==> default: Provisioners will not be run since container doesn't support SSH.
```

Once the `vagrant up` is over, the container will be running and an image will have been created. You can use the regular Docker commands to interact with the container or use the new `vagrant docker-logs` and `vagrant docker-run` commands. Standard commands like `vagrant status`, `vagrant destroy` will also work with your containers.

It is very likely that you will not install `ssh` in your container. Therefore the Vagrant provisioners will not be able to run. Any software installation within your container will need to happen through the Dockerfile.

Discussion

I created a simple environment to help you test this recipe. Similarly to other recipes you can clone the git repository that accompanies this book and head over to the recipe examples. An Ubuntu 14.04 virtual machine will be started, Docker 1.01 will be installed as well as Vagrant. In the `/vagrant/provider` directory you will find yet another `Vagrantfile` (shown above), and a Dockerfile. This Dockerfile builds a simple Flask application in the container.

```
$ git clone
$ cd ch02/vagrantprovider/
```

```
$ vagrant up
$ vagrant ssh
$ cd /vagrant/provider
$ vagrant up --provider=docker
```

The possible configurations of the Vagrantfile are almost a one to one match with directives in a Dockerfile. You can define what software to install in a container, what environment variables to pass, which ports to expose, which containers to link to and which volumes to mount. The interesting thing is that Vagrant will attempt to translate the regular Vagrant configuration into Docker run options. For instance, forwarding ports from a Docker container to the host can be done with the regular Vagrant command:

```
config.vm.network "forwarded_port", guest: 5000, host: 5000
```

Overall, it is a personal feeling that the Docker support in Vagrant should be seen as a transitioning step for developers who might have invested a lot of work with Vagrant and would like to slowly adopt Docker.

Vagrant also features a Docker provisioner. It can be used in a case where you are starting virtual machines, provisioning them with configuration management solutions (e.g Puppet, Chef) but would also like to start containers within those virtual machines.

See Also

- Vagrant Docker provider configuration
- Vagrant Docker provider documentation

2.7 Using Packer to Create a Docker Image

Problem

You have developed several configuration management recipes using Chef, Puppet, Ansible or SaltStack. You would like to re-use those recipes to build Docker images.

Solution

Use Packer from Hashicorp. Packer as mentioned on its home page is a tool to create identical machine images for multiple platforms from a single template definition. For example from a template it can automatically create images for Amazone EC2 (i.e AMI), VMware, VirtualBox, DigitalOcean etc. One of those target platforms is Docker.

This means that if you define a Packer template, you can automatically generate a Docker image. You can also post-process it to tag the image and push it to Docker Hub (see Recipe 2.8).

The following template shows three main steps. First it specifies a *builder*, here we use Docker and specify to use the base image *ubuntu:14.04*, then it defines the provisioning step. Here we use a simple shell provisioning. Finally, it lists post-processing steps. Here we only tag the resulting image.

```
{
"builders": [
  {
    "type": "docker",
    "image": "ubuntu:14.04",
    "commit": "true"
  }
],
"provisioners": [
  {
    "type": "shell",
    "script": "bootstrap.sh"
  }
],
"post-processors": [
   {
     "type": "docker-tag",
     "repository": "how2dock/packer",
     "tag": "latest"
   }
]
}
```

You can validate the template and launch a build of the image with two commands:

```
$ packer validate template.json
$ packer build template.json
```

Setup several builders in your template and output different images for your application (e.g Docker and AMI)

There is currently a bug with Packer and the latest version of Docker 1.4.1.

To help you test Packer, I created a Vagrantfile which starts an Ubuntu 14.04 virtual machine, installs Docker 1.0.1 on it and downloads Packer. Test it like this:

```
$ git clone https://github.com/how2dock/docbook.git
$ cd ch02/packer
$ vagrant up
$ vagrant ssh
$ cd /vagrant
$ /home/vagrant/packer validate template.json
Template validated successfully.
$ /home/vagrant/packer build template.json
...
==> docker: Creating a temporary directory for sharing data...
==> docker: Pulling Docker image: ubuntu:14.04
...
==> Builds finished. The artifacts of successful builds are:
--> docker: Imported Docker image: 3ebae8e2f2a8af8f2c5f366c603091c5e9c8e234bff8
--> docker: Imported Docker image: how2dock/packer:latest
$ docker images
REPOSITORY          TAG         IMAGE ID        CREATED           VIRTUAL SIZE
how2dock/packer     latest      3ebae8e2f2a8    20 seconds ago    210.8 MB
ubuntu              14.04       8eaa4ff06b53    11 days ago       192.7 MB
```

In this particular example you can now run `nginx` (which has been installed via the `boostrap.sh` script) with:

```
$ docker run -d -p 80:80 how2dock/packer /usr/sbin/nginx -g "daemon off;"
```

But note that since a Dockerfile was not used to create this image, there was no CMD or ENTRYPOINT defined. Nginx will not be started when the container is launched, hence a container started from the image generated without specifying how to run `nginx` will exit right away.

Discussion

Packer is a great tool which can help you migrate some of your work from previous *DevOps* workflows into a Docker based workflow. However, Docker containers run applications in the foreground and encourage running single application processes per container. Hence, creating a Docker image with Packer that would have for example, Mysql, Nginx and Wordpress in the same image would go contrary to the Docker philosophy and might prove difficult to run without some additional manual post-processing with something like Supervisor (see Recipe 1.15).

The solution above featured a basic shell provisioning. If you have existing configuration management recipes, you can also use them to create a Docker image. Packer features shell, Ansible, Chef, Puppet and Salt provisioners. As an example, the `template-ansible.json` file in the repository used above makes use of the Ansible local provisioner. The packer template gets modified like so:

```
{
"builders": [
  {
    "type": "docker",
    "image": "ansible/ubuntu14.04-ansible:stable",
    "commit": "true"
  }
],
"provisioners": [
  {
    "type": "ansible-local",
    "playbook_file": "local.yml"
  }
],
"post-processors": [
   {
     "type": "docker-tag",
     "repository": "how2dock/packer",
     "tag": "latest"
   }
]
}
```

It uses a special Docker image, pulled from Docker hub that has Ansible installed in it. Packer will use the local Ansible CLI to run the ansible playbook `local.yml`. The playbook installs `nginx` locally:

```
---
- hosts: localhost
  connection: local
  tasks:
   - name: install nginx
     apt: pkg=nginx state=installed update_cache=true
```

The result of building with Packer will be a working Docker template:

```
$ /home/vagrant/packer build template-ansible.json
...
==> docker: Creating a temporary directory for sharing data...
==> docker: Pulling Docker image: ansible/ubuntu14.04-ansible:stable
    docker: Pulling repository ansible/ubuntu14.04-ansible
...
==> docker: Provisioning with Ansible...
    docker: Creating Ansible staging directory...
    docker: Creating directory: /tmp/packer-provisioner-ansible-local
    docker: Uploading main Playbook file...
...
    docker:
    docker: PLAY [localhost] *******************************************
    docker:
    docker: GATHERING FACTS ********************************************
    docker: ok: [localhost]
    docker:
```

```
    docker: TASK: [install nginx] *********************************
    docker: changed: [localhost]
    docker:
    docker: PLAY RECAP ********************************************
    docker: localhost  : ok=2    changed=1    unreachable=0    failed=0
    docker:
==> docker: Committing the container
..
    docker (docker-tag): Repository: how2dock/packer:latest
Build 'docker' finished.
```

2.8 Publishing your image to Docker hub

Problem

You have written a Dockerfile and build an image for a useful container. You want to share this image with everyone.

Solution

Share this image on the Docker hub. Docker hub is to Docker what GitHub is to source code. It allows anyone to host its image on-line and share it publicly or keep it private. To share an image on Docker hub you will need to:

- Create an account on Docker hub
- Login to the hub on your Docker host
- Push your image

Let's get started. Registering only takes a valid email address. Head over to the signup page and create create an account

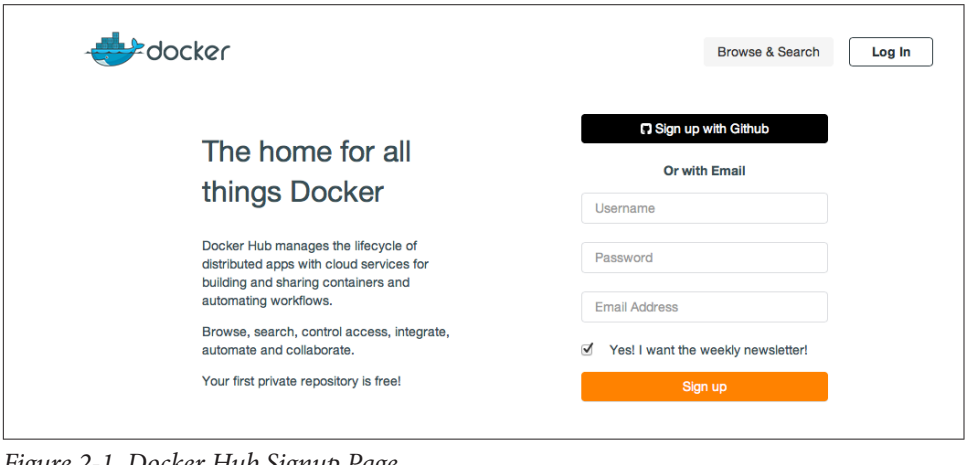


Figure 2-1. Docker Hub Signup Page

After verifying the email address that you used to create the account, your registration will be complete. This will be a free account that allows you to publish public images as well as have one private repository. If you want to have more than one private repositories, you will need to pay a subscription.

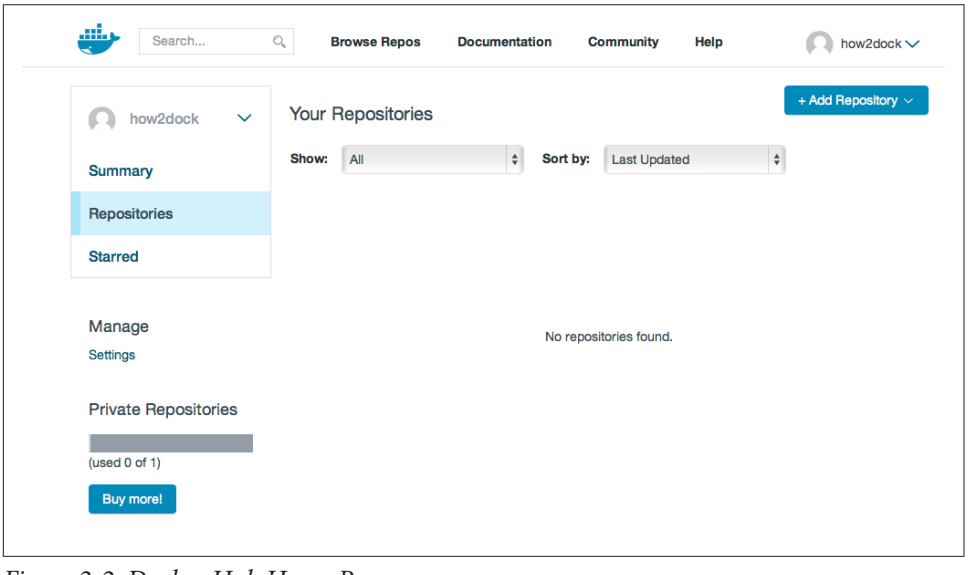


Figure 2-2. Docker Hub Home Page

Now that we have an account created, we can head back to our Docker host, select one of our images and use the docker CLI to publish this image on our public repository. This will be a three step process:

- Login with `docker login`. This will ask for our Docker Hub credentials
- Tag an existing image with our username from Docker Hub
- Push the newly tagged image

The login step will store your Docker Hub credentials in a `~/.dockercfg` file

```
$ docker login
Username: how2dock
Password:
Email: how2dock@gmail.com
Login Succeeded
$ cat ~/.dockercfg
{"https://index.docker.io/v1/":{"auth":"aG93MmerTertWertqyaVpoMzUh",
                                "email":"how2dock@gmail.com"}}
```

If we check the list of images that we currently have, we see that our flask image from Recipe 2.4 is using a local repository and has a tag called `latest`.

```
$ docker images
REPOSITORY      TAG             IMAGE ID        CREATED         VIRTUAL SIZE
flask           latest          88d6464d1f42    5 days ago      354.6 MB
...
```

To push this image to our Docker Hub account we need to tag this image with our own Docker Hub repository with the `docker tag` command (see Recipe 2.5).

```
$ docker tag flask how2dock/flask
sebimac:flask sebgoa$ docker images
REPOSITORY          TAG          IMAGE ID          CREATED       VIRTUAL SIZE
flask               latest       88d6464d1f42      5 days ago    354.6 MB
how2dock/flask      latest       88d6464d1f42      5 days ago    354.6 MB
```

We now have our Flask image with the repository of `how2dock/flask` which follows the proper naming convention for repositories. We are ready to push the image. Docker will attempt to push the various layers that make the image, if the layer is pre-existing on the Docker Hub it will skip it. Once the push is finished the `how2dock/flask` image will be visible in your Docker Hub page and anyone will be able to `docker pull how2dock/flask`.

```
$ docker push how2dock/flask
The push refers to a repository [how2dock/flask] (len: 1)
Sending image list
Pushing repository how2dock/flask (1 tags)
511136ea3c5a: Image already pushed, skipping
01bf15a18638: Image already pushed, skipping
30541f8f3062: Image already pushed, skipping
e1cdf371fbde: Image already pushed, skipping
9bd07e480c5b: Image already pushed, skipping
e659c9e9ba21: Image successfully pushed
22ebd8b6f3e6: Image successfully pushed
54280d866a09: Image successfully pushed
6667589085ed: Image successfully pushed
dc4a9a43bb7f: Image successfully pushed
e394b9fbe3fa: Image successfully pushed
3f7abcdc10d4: Image successfully pushed
88d6464d1f42: Image successfully pushed
Pushing tag for rev [88d6464d1f42] on
{https://cdn-registry-1.docker.io/v1/repositories/how2dock/flask/tags/latest}
```

See Also

- Docker Hub reference

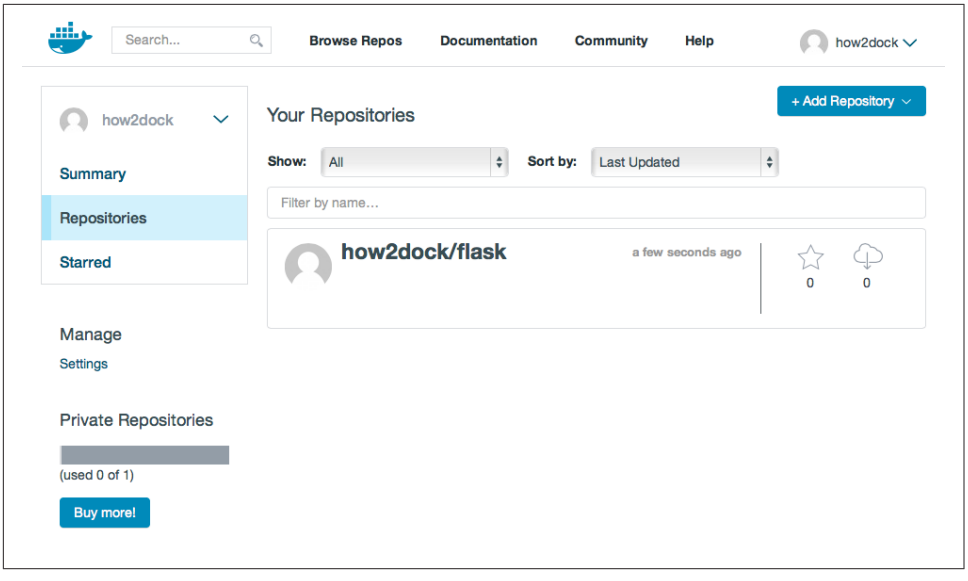


Figure 2-3. Docker Hub Flask image

Discussion

The `docker tag` command allows you to change the repository and tag of an image. In our example we did not specify a tag, hence Docker assigned it the `latest` tag. If we wanted we could specifiy tags and push these to Docker hub, maintaining several versions of an image in the same repository.

This recipes allowed us to be introduced to two new docker CLI commands `docker tag` and `docker push`. There is one more that is worth noting in terms of image management and that is `docker search`. It allows you to search for images in Docker Hub. For example, if we are looking for an image that would give us `postgres`:

```
$ docker search postgres
NAME                      DESCRIPTION             ...   STARS     OFFICIAL   AUTOMATED
postgres                  The PostgreSQL          ...   402       [OK]
paintedfox/postgresql     A docker image          ...   50                   [OK]
helmi03/docker-postgis    PostGIS 2.1 in          ...   20                   [OK]
atlassianfan/jira         Atlassian Jira          ...   17                   [OK]
orchardup/postgresql      https://github          ...   16                   [OK]
abevoelker/ruby           Ruby 2.1.2, Post        ...   13                   [OK]
slafs/sentry              my approach for         ...   12                   [OK]
...
```

The command actually returns over 600 images. The first one is the official postgres image maintained by the postgres team. The other ones are images created by users of Docker Hub. Some of the images are built and pushed automatically and we will learn about Automated builds in Recipe 2.10.

2.9 Running a Private Registry

Problem

Using the public Docker Hub is very easy, however you might have data governance concerns with your images being hosted outside of your own infrastructure. You would like to run your own Docker registry, hosting it on your own infrastructure.

Solution

Use the Docker registry image and start a container from it. You will have your private registry.

Pull the official `registry` image and run it as a detached container. You should then be able to curl `http://localhost:5000` for a quick test that the registry is running.

```
$ docker pull registry:0.9.1
$ docker run -d -p 5000:5000 registry:0.9.1
$ curl http://localhost:5000
"\"docker-registry server\""
```

You can now prepare a local image that you have created previously (e.g a `flask` image, Recipe 2.4) and tag it with the proper naming convention for use with a private registry. In our case, the registry is running at `http://localhost:5000`, so we will prefix our tag with `localhost:5000` and then push this image to the private registry.

```
$ docker tag flask localhost:5000/flask
$ docker push localhost:5000/flask
The push refers to a repository [localhost:5000/flask] (len: 1)
Sending image list
Pushing repository localhost:5000/flask (1 tags)
511136ea3c5a: Image successfully pushed
...
88d6464d1f42: Image successfully pushed
Pushing tag for rev [88d6464d1f42] on
{http://localhost:5000/v1/repositories/flask/tags/latest}
```

If you try to access this private registry from another machine, you will get an error message telling you that your Docker client does not allow you to use an insecure registry. For testing purposes only, edit your Docker configuration file to use the `insecure-registry` option. For instance on Ubuntu 14.04, edit `/etc/default/docker` and add the line:

```
DOCKER_OPTS="--insecure-registry <IP_OF_REGISTRY>:5000"
```

Then restart Docker on your machine `sudo service docker restart` and try to access the remote private registry again. (Remember that this is done on a different machine than where you are running the registry).

Discussion

In this short example, we used the default setup of the registry. It assumed no authentication, an insecure registry, local storage and a sqlalchemy search backend. All of these can be set via environmental variables or by editing a configuration file. This is well documented.

The registry that is running via the `registry` Docker image is a Flask application, running via Gunicorn. It exposes an API, that you can access with your own registry client or even `curl`.

For example to list all images stored in the private registry, you can use the `search` API with no search term:

```
$ curl -s http://localhost:5000/v1/search | python -m json.tool
{
    "num_results": 1,
    "query": "",
    "results": [
        {
            "description": "",
            "name": "library/flask"
        }
    ]
}
```

You will notice that the name of the image is prefixed with the default registry namespace `library`. If we push a *busybox* image to the private registry and use curl again to search across all images in the repository we get:

```
$ docker pull buysbox
$ docker tag busybox localhost:5000/busybox
$ docker push localhost:5000/busybox
$ curl -s http://localhost:5000/v1/search | python -m json.tool
{
    "num_results": 2,
    "query": "",
    "results": [
        {
            "description": "",
            "name": "library/flask"
        },
        {
            "description": "",
            "name": "library/busybox"
        }
    ]
}
```

You can delete a repository by issuing a HTTP DELETE request to `/v1/repositories/library/flask`:

```
$ curl -s -X DELETE http://localhost:5000/v1/repositories/library/flask/ | python -m json.tool
true
```

To list all tags for a specific image use the `v1/repositories/library/<image name>/tags` endpoint:

```
$ curl -s http://localhost:5000/v1/repositories/library/busybox/tags | python -m json.tool
{
    "latest": "4986bf8c15363d1c5d15512d5266f8777bfba4974ac56e3270e7760f6f0a8125"
}
```

And if you want to add a tag manuall, send a HTTP PUT request with some json payload like so:

```
$ curl -s -X PUT
          -H 'Content-Type: application/json;'
          -d '"4986bf8c15363d1c5d15512d5266f8777bfba4974ac56e3270e7760f6f0a8125"'
                  http://localhost:5000/v1/repositories/library/busybox/tags/foobar
$ curl -s http://localhost:5000/v1/repositories/library/busybox/tags | python -m json.tool
{
    "foobar": "4986bf8c15363d1c5d15512d5266f8777bfba4974ac56e3270e7760f6f0a8125",
    "latest": "4986bf8c15363d1c5d15512d5266f8777bfba4974ac56e3270e7760f6f0a8125"
}
```

These couple examples using `curl` are meant to give you a sense of the registry API. Complete API documentation is availabe on the Docker website.

See Also

- The Docker registry page on Docker Hub.
- The more extensive documentation on Github

2.10 Setting Up an Automated Build on DockerHub for Continous Integration/Deployment

Problem

You have access to Docker Hub (see Recipe 2.8) and already pushed an image to it. However this is a manual process. You want to automate the build of this image everytime you commit a change to it.

Solution

Instead of setting up a standard repository, create an *Automated Build* repository and point to your application on GitHub or Bitbucket. Then

On your docker hub page, click on the *Add Repository* button and select *Automated Build* (Figure 2-4). You will then have the choice between GitHub and Bitbucket (Figure 2-5).

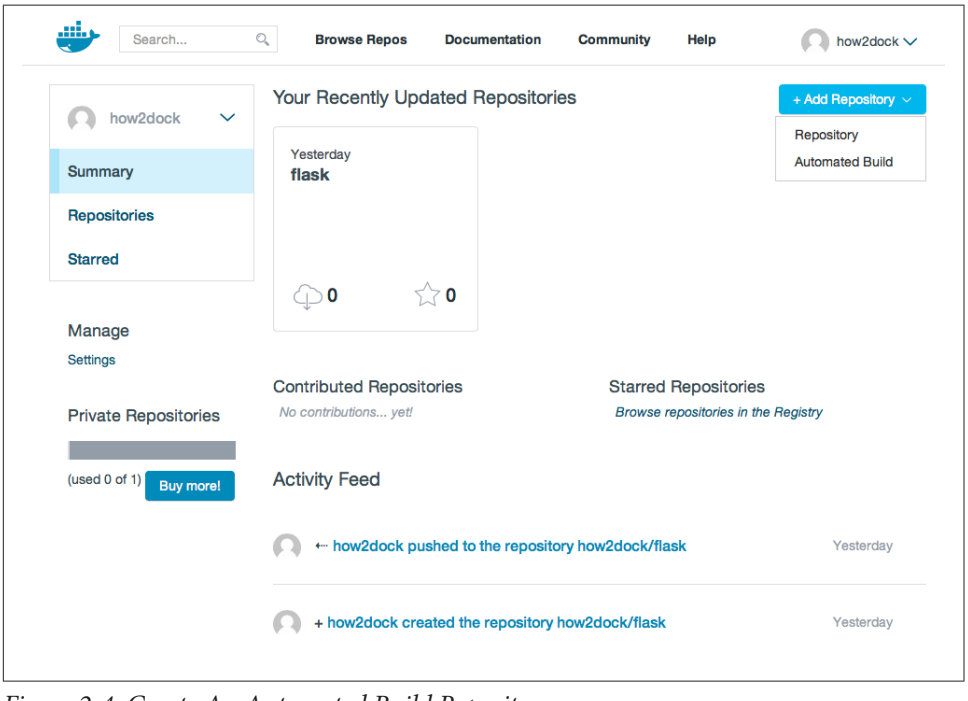


Figure 2-4. Create An Automated Build Repository

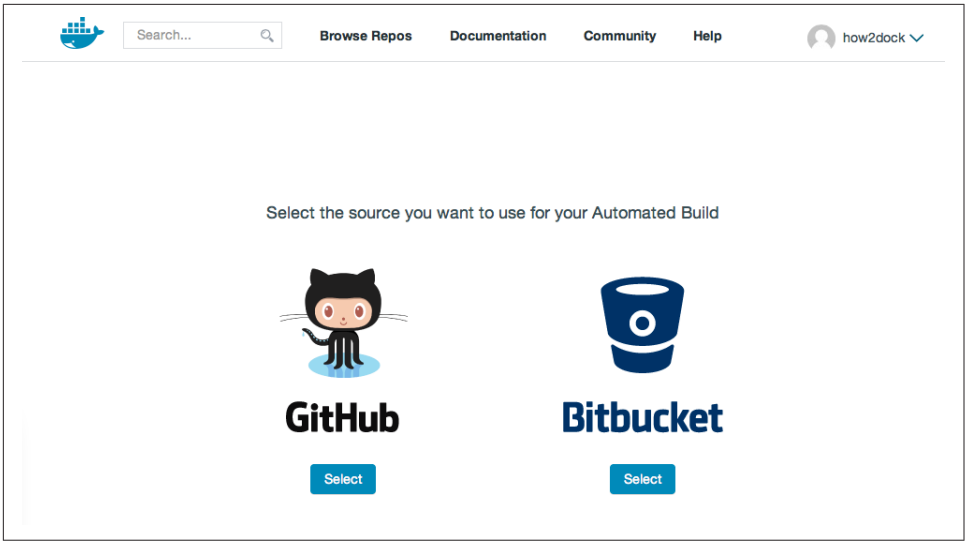


Figure 2-5. Choose Between GitHub and Bitbucket

Docker Hub allows you to setup an automated build as a public or private repository pointing to a public or private code repository. If you are setting up a private automated build Docker Hub will need read and write access to your GitHub account.

After selecting the type of on-line version control system you want to use, you will be allowed to select the project you want to build from (Figure 2-6). This should be a project on GitHub or Bitbucket that contains the Dockerfile you want to build. Once selected, you will be able to give a name to the Docker Hub repository your are creating, you will be able to select the branch and specify the location of the Dockerfile. This is handy as it allows you to maintain several Dockerfile inside a single GitHub/Bitbucket repository.

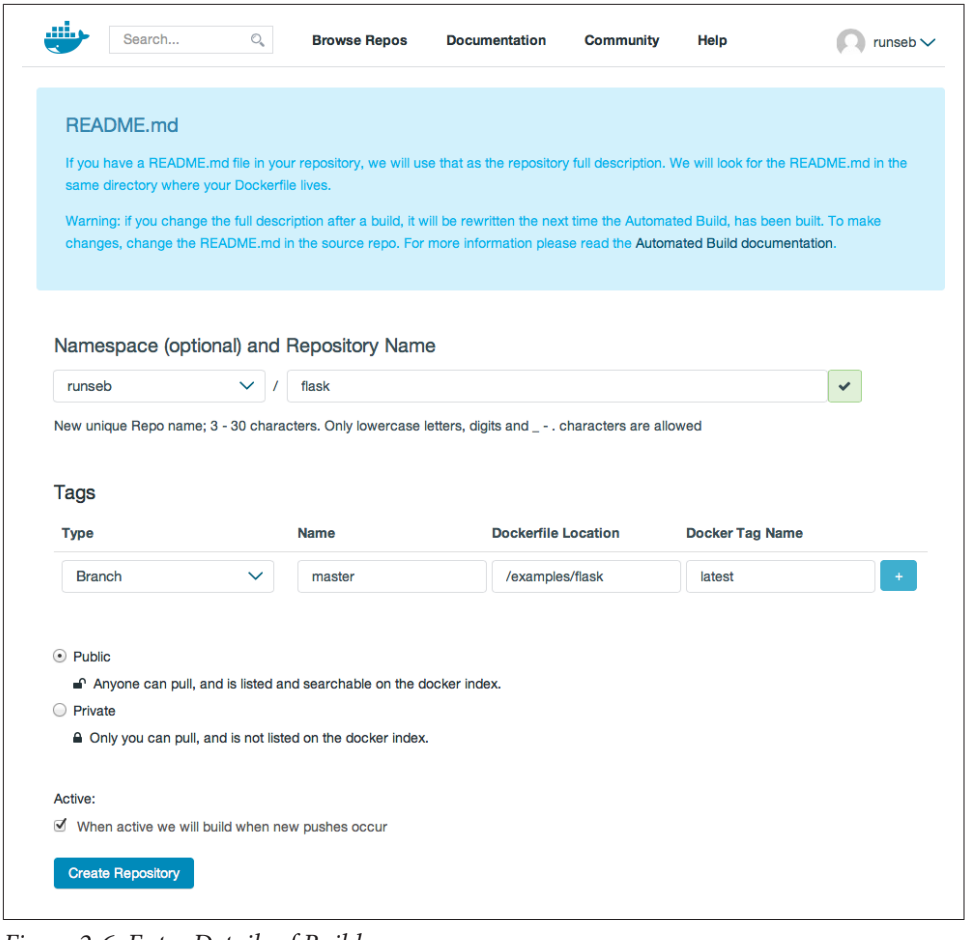


Figure 2-6. Enter Details of Build

The name of the Docker hub repository you are creating does not have to be the same as the GitHub/ bitbucket repository you select.

Once you have setup the build, you will have access to the build details. The status will go from *pending* to *building* to *pushing* and finally to *finished*. Once the build is finished you will be able to pull the new image:

```
$ docker pull runseb/flask
```

The *Dockerfile* tab will automatically be populated with the content of your Dockerfile in your GitHub repository and the *Information* tab will automatically be populated with the content of the *README.md* file if it exists.

As soon as you push a new commit to the GitHub repo used for the build, a new build will be triggered and once finished and new image will be available.

You can edit the build settings, to trigger builds from different branches and specify a different tag. For instance you can decide to build from your master branch and associate the *latest* tag to it, and use a release branch to build a different tag (i.e *1.0* tag from a *1.0* release branch)

Discussion

In addition to builds being automatically triggered when you push to your GitHub or Bitbucket repository, you can trigger builds by sending an http POST request to a specific url generated on the *Build Trigger* page, see Figure 2-7 below. To prevent abusing the system, builds maybe be ignored.

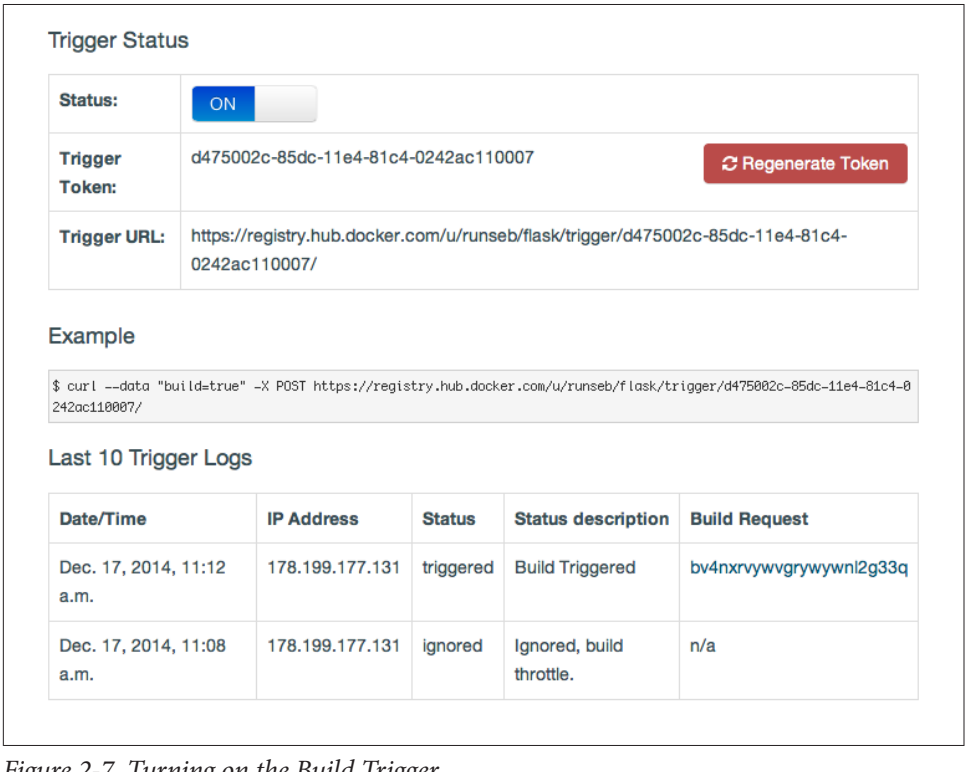


Figure 2-7. Turning on the Build Trigger

Finally, whether you build automatically or use a trigger on your own, you can also use Webhooks. In the *Build details* page of your automated build you will be able to access the *Web hooks* page. In it you can add URLs that will receive an http POST

when a successful build happens. The body of this POST request will contain a call back URL. In response, you will need to send another http POST with a json payload containing the *state* key and the value of either *success*, *failure* or *error*. On receiving a successfull state, the automated build can call another webhook, henceforth allowing you to chain several actions together.

See Also

- The Automated builds reference documentation

2.11 Setting up a Local Automated Build Using a Git Hook and a Private Registry

Problem

Automated builds using Docker Hub and GitHub or Bitbucket are great (see Recipe 2.10), but you might be using a private registry (i.e local hub) and may want to trigger docker builds when you commit to your local git projects.

Solution

Create a `post-commit` git hook that triggers a build and pushes the new image to your private registry.

In the root tree of your git project, create a bash script `./git/hooks/post-commit` that contains something as simple as this

```
#!/bin/bash

tag=`git log -1 HEAD --format="%h"`
docker build -t flask:$tag /home/sebgoa/docbook/examples/flask
```

Make it executable with `chmod +x .git/hooks/post-commit`

Now everytime you will make a commit to this git project, this post-commit bash script will run. It will create a `tag` using the short version of the sha of the latest commit and then trigger the build based on the Dockerfile referenced. It will then create a new image with the name `flask` and the computer tag.

```
$ git commit -m "fixing hook"
9c38962
Sending build context to Docker daemon 3.584 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu:14.04
 ---> 9bd07e480c5b
Step 1 : RUN apt-get update
```

```
 ---> Using cache
 ---> e659c9e9ba21
Step 2 : RUN apt-get install -y python
 ---> Using cache
 ---> 22ebd8b6f3e6
Step 3 : RUN apt-get install -y python-pip
 ---> Using cache
 ---> 54280d866a09
Step 4 : RUN apt-get clean all
 ---> Using cache
 ---> 6667589085ed
Step 5 : RUN pip install flask
 ---> Using cache
 ---> dc4a9a43bb7f
Step 6 : ADD hello.py /tmp/hello.py
 ---> 8f60127ef5b1
Removing intermediate container 3af40a03b3f3
Step 7 : EXPOSE 5000
 ---> Running in 05c13744c7bf
 ---> b464df2bc5ca
Removing intermediate container 05c13744c7bf
Step 8 : CMD python /tmp/hello.py
 ---> Running in 124cd2ada52d
 ---> 9a50c7b2bee9
Removing intermediate container 124cd2ada52d
Successfully built 9a50c7b2bee9
[master 9c38962] fixing hook
 1 file changed, 1 insertion(+), 1 deletion(-)
sebimac:flask sebgoa$ docker images
REPOSITORY         TAG             IMAGE ID        CREATED         VIRTUAL SIZE
flask              9c38962         9a50c7b2bee9    5 days ago      354.6 MB
```

While this works very nicely and is achieved with two lines of bash, if the build were to take a long time, it would not be practical to build the image as a post-commit task. It would be better to use the post-commit hook to trigger a remote build and then register this image in a private repo.

Discussion

Unfinished recipe. Might use stackstorm or striderCD to showcase this a bit better.

CHAPTER 3

Docker Networking

This chapter consists of recipes about Docker networking. It will cover Docker built-in networking capabilities as well as a few third party networking solutions. You can send me suggestions at *how2dock@gmail.com*

3.1 Introducing Docker Containers Networking

Problem

You would like to understand the basics of networking Docker containers.

Solution

In the default installation of Docker, a linux bridge `docker0` is created on the Docker host. This bridge gets a private address and a subnet associated to it. While it is random, most of the time your `docker0` bridge will get the address 172.17.42.1. All containers that will attach to this bridge will get an address in the 172.17.42.0/24 network. Containers networking interfaces get attached to this bridge and will use the `docker0` interface has a networking gateway. When a container gets created, Docker creates a pair of "peer" network interfaces that are placed in two separate networking namespace. One interface in the networking namespace of the container (i.e `eth0`) and one interface in the networking namespace of the host, attached to the `docker0` bridge.

To illustrate this setup, let's have a look at a Docker host and let's start a container. You can use an existing Docker host or use the Vagrant box prepared for this book like so:

```
$ git clone https://github.com/how2dock/docbook
$ cd ch03/simple
```

```
$ vagrant up
$ vagrant ssh
```

The diagram below shows the network configuration of this Vagrant box. It contains a single NAT interface to get to the outside network. Inside the host is a linux bridge `docker0` and two containers are depicted.

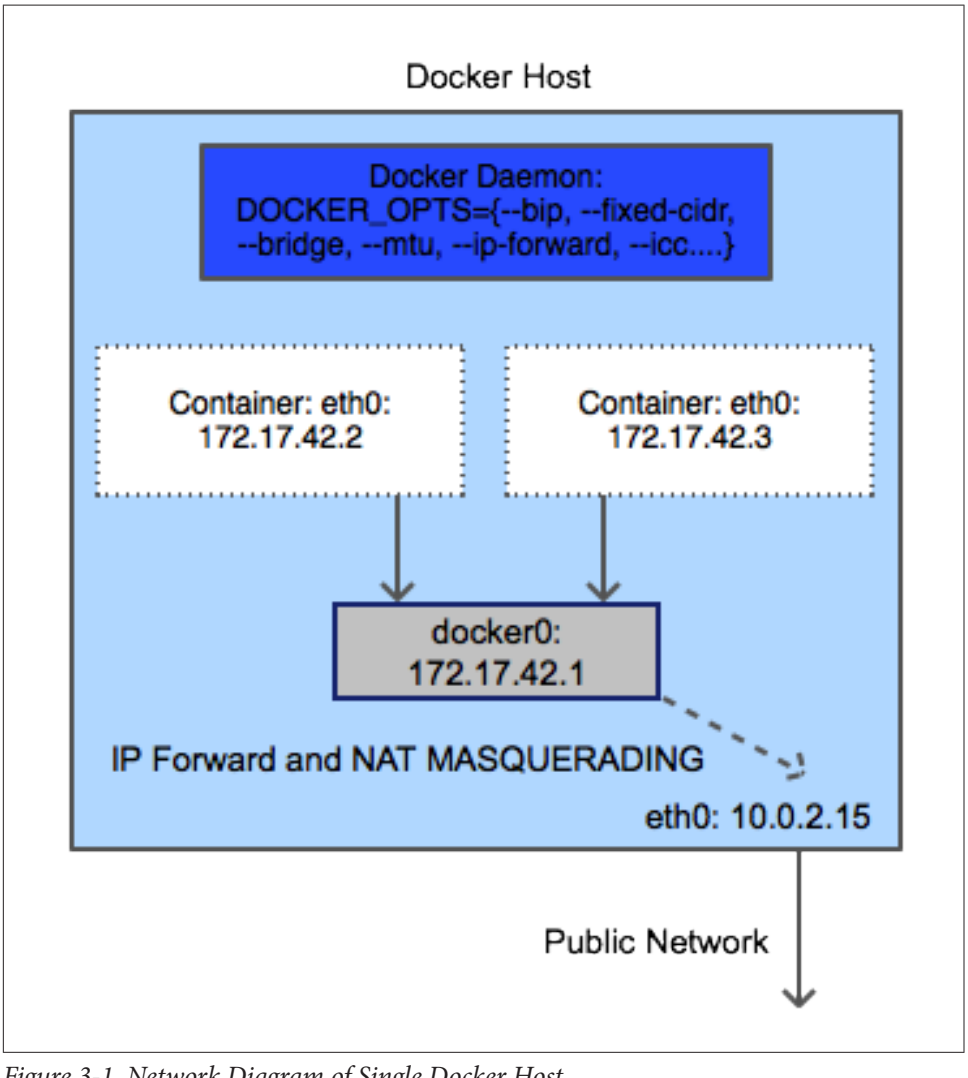


Figure 3-1. Network Diagram of Single Docker Host.

Once connected to he host, if you list the network links you will see the loopback device, an `eth0` interface and the `docker0` bridge as depicted in the diagram. Docker

started when the machine was booted and automatically created a bridge and assigned a subnet to it.

```
$ ip -d link show
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 promiscuity 0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast ...
    link/ether 08:00:27:98:a7:ad brd ff:ff:ff:ff:ff:ff promiscuity 0
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue ...
    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff promiscuity 0
    bridge
```

Now let's start a container and check its network interface:

```
$ docker run -ti --rm ubuntu:14.04 bash
root@4e3ffb9bc381:/# ip addr show eth0
6: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:11:2a:03 brd ff:ff:ff:ff:ff:ff
    inet 172.17.42.3/24 scope global eth0
...
```

Indeed the container got an IP (i.e 172.17.42.3) in the 172.17.42.0/24 network. On the host itself a virtual interface (i.e `veth450b81a` below) was created and attached to the bridge. You can see this by listing the links on the Docker host with the `ip` tool (or `ifconfig`) and the `brctl` tool if you installed the `bridge-utils` package.

```
$ ip -d link show
...
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP ...
    link/ether aa:85:e0:61:69:2d brd ff:ff:ff:ff:ff:ff promiscuity 0
    bridge
7: veth450b81a: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast master docker0 ...
    link/ether aa:85:e0:61:69:2d brd ff:ff:ff:ff:ff:ff promiscuity 1
    veth
$ brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.aa85e061692d       no              veth450b81a
```

From the container you can ping the network gateway 172.17.42.1 (i.e `docker0`) other containers on the same host and you can ping the outside world.

Start another container on a separate terminal and try to ping each container. Verify that the second container interface is also attached to the bridge. Since there are no IP tables rules dropping traffic, both containers can communicate with each other on any port.

Discussion

Outbound networking traffic is forwarded to the other interfaces of your Docker host via IP forwarding and will go through NAT translation using an IP table masquerad-

ing rule. On your Docker host you can check that IP forwarding has been enabled with:

```
$ cat /proc/sys/net/ipv4/ip_forward
1
```

Try turning off IP forwarding, you will see that your container will loose outbound network connectivity.

```
# echo 0 > /proc/sys/net/ipv4/ip_forward
```

You can also check the NAT rule that does the IP masquerading for outbound traffic:

```
$ sudo iptables -t nat -L
...
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  172.17.42.0/24       anywhere
...
```

In Recipe 3.5 we will see how to create this configuration from scratch.

See Also

- Docker official networking documentation

3.2 Choosing a Container Networking Stack

Problem

When starting a container you want to be able to choose a specific network stack.

Solution

In Recipe 3.1 we started a container using the defaults of the `docker run` command. This attached the container to a linux bridge and created a the appropriate network interfaces. It takes advantage of IP forwarding and IP tables managed by the Docker engine to provide outbound network connectivity and NAT.

However you can start a container with a different type of networking, namely the host networking stack, no networking stack at all or the networking stack of another container using the `--net` option of `docker run`.

Let's start a container on a Docker host without any networking stack using `--net=none`:

```
$ docker run -it --rm --net=none ubuntu:14.04 bash
root@3a22f5076f9a:/# ip -d link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT ...
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00 promiscuity 0
root@3a22f5076f9a:/# route
        Kernel IP routing table
        Destination     Gateway     Genmask     Flags Metric Ref    Use Iface
```

When listing the networking links, you only see a link local address. There is no other network interfaces and no networking routes. You will have to bring up the network manually if you need it (see Recipe 3.5).

Now let's start a container with the networking stack of the host using `--net=host`:

```
$ docker run -it --rm --net=host ubuntu:14.04 bash
root@foobar-server:/# ip -d link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT ...
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00 promiscuity 0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state ...
    link/ether 08:00:27:98:a7:ad brd ff:ff:ff:ff:ff:ff promiscuity 0
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state ...
    link/ether c6:4b:6b:b7:4b:98 brd ff:ff:ff:ff:ff:ff promiscuity 0
    bridge
```

When listing the networking links within this container, you see exactly the same interfaces as in the host, including the `docker0` bridge. This means that while the container processes are isolated in their own namespace and resource limited through cgroup, the network namespace of the container is the same as the one of the host. You see in the example above that the hostname of the container is actually the Docker host hostname (you cannot use the `-h` option to set a hostname when using the host networking stack). Note however that you will not be able to reconfigure the host network from such a container. For example you cannot bring down interfaces:

```
root@foobar-server:/# ifconfig eth0 down
SIOCSIFFLAGS: Operation not permitted
```

While it can be very handy, it needs to be handled with lots of care.

Starting a container with `--net=host` can be very dangerous, especially if you start a privileged container with `--privileged=true`

The final option is to use the network stack of another already running container. Let's start a container with the hostname *cookbook*:

```
$ docker run -it --rm -h cookbook ubuntu:14.04 bash
root@cookbook:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:02
```

```
          inet addr:172.17.0.2  Bcast:0.0.0.0  Mask:255.255.0.0
                  ...
```

We see at the prompt that the hostname has been set to *cookbook* and that the IP is 172.17.02. It got attached to the `docker0` bridge. Now let's start another container and let's use the same network namespace. First we list the running containers to get the name of the container with just started. The convention calls to use `--net=container:CONTAINER_NAME_OR_ID`.

```
$ docker ps
CONTAINER ID    IMAGE         COMMAND  ...   NAMES
cc7f72826c36    ubuntu:14.04  "bash"   ...   cocky_galileo
$ docker run -ti --rm --net=container:cocky_galileo ubuntu:14.04 bash
root@cookbook:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:02
          inet addr:172.17.0.2  Bcast:0.0.0.0  Mask:255.255.0.0
                  ...
```

As we see above, the new container has the same hostname as the first container started and of course has the same IP. The processes in each container will be isolated and exist in their own process namespace, but they share the same networking namespace and can communicate on the loopback device.

Discussion

Which networking namespace to use is up to the application you are running and what you want the network to look like. Docker networking is extremely flexible and will allow you to build any topology and secure network scenarios between your container processes.

See Also

- How Docker networks containers

3.3 Configuring the Docker Daemon IP tables and IP forwarding settings

Problem

You may not like the fact that by default the Docker daemon turned on IP forwarding as well as modified your IP tables. You would like more control on how traffic flows on your host, between your containers and with the outside world.

Solution

This behavior is customizable when you start the Docker daemon through the `--ip-forward`, `--iptables` options.

To try this, stop the Docker daemon on the host that you are using. On Ubuntu/Debian based systems edit `/etc/default/docker` and set these options to false (on CentOS/RHEL systems edit `/etc/sysconfig/docker`).

```
$ sudo service docker stop
$ sudo su
# echo DOCKER_OPTS=\"--iptables=false --ip-forward=false\" >> /etc/default/docker
```

You may have to remove the POSTROUTING rule manually first as well as set the IP forwarding rule to zero, before restarting the Docker daemon. To do this, try the following on your Docker host:

```
# iptables -t nat -D POSTROUTING 1
# echo 0 > /proc/sys/net/ipv4/ip_forward
```

With this configuration, traffic on the Docker bridge `docker0` will not be forwarded to the other networking interfaces and the POSTROUTING masquerading rule will not be present. This means that all outbound connectivity from your containers to the outside world will be dropped.

Verify this behavior by starting a container and trying to reach the outside world, for example:

```
$ docker run -it --rm ubuntu:14.04 bash
WARNING: IPv4 forwarding is disabled.

root@ba12d578e6c8:/# ping -c 2 -W 5 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.

--- 8.8.8.8 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1009ms
```

To re-enable communication to the outside manually, enable IP forwarding and set the POSTROUTING rule on the Docker host like so:

```
# echo 1 > /proc/sys/net/ipv4/ip_forward
# iptables -t nat -A POSTROUTING -s 172.17.0.0/16 -j MASQUERADE
```

Go back to the terminal of your container and try pinging `8.8.8.8` again.

With `--iptables=false` set for the Docker daemon, you will not be able to restrict traffic between containers since Docker will not be able to manage the IP table rules. This means that all containers started on the same bridge will be able to communicate on all ports. See the Discussion below for more on this topic.

Discussion

By default the Docker daemon is allowed to manage the IP table rules on the Docker host, this means that it can add rules that restrict traffic between containers and provide network isolation between them.

If you disallow Docker to manipulate the IP table rules, then it will not be able to add rules that restrict traffic between containers.

If you do allow Docker to manipulate the IP table rules, then you can set the `--icc=false` option for the Docker daemon. This will add a default DROP rule for all packets on the bridge and containers will not be able to reach each other.

You can try this by editing the Docker config file (i.e `/etc/default/docker` on Ubuntu/Debian and `/etc/sysconfig/docker` on CentOS/RHEL) and adding the `--icc=false` option. Restart Docker and start two containers on your host you will see that you cannot ping one container from another.

Since this drastically restricts traffic between containers, how can you have them communicating ? This is solved with container linking, which creates specific IP table rules (see Recipe 3.4).

3.4 Linking Containers in Docker

Problem

Solution

Discussion

Allow ping from the Docker host to all the containers

```
$ sudo iptables -A DOCKER -p icmp --icmp-type echo-request -j ACCEPT
$ sudo iptables -A DOCKER -p icmp --icmp-type echo-reply -j ACCEPT
```

3.5 Using Pipework to Understand Container Networking

Problem

Docker built-in networking capabilities work great but you would like a hands-on approach where you use traditional networking tools to create network interfaces for your containers.

Solution

Use pipework. Pipework was created by Jerome Petazzoni from Docker inc back in 2013, it manipulates cgroups and network namespaces to build networking scenarios for your containers. At first it supported pure LXC containers but now it also supports Docker containers. If you start a container with the `--net=none` option, `pipework` is very handy to add networking to that container.

While almost everything you can do with pipework is built-in within Docker, it is a great tool to reverse engineer Docker networking and get a deeper understanding of how the containers communicate with each other and the outside world. This recipe aims at showing you a few examples to deconstruct Docker networking capabilities and become a little more comfortable dealing with different networking namespaces.

`pipework` is a single bash script that you can download. For convenience I created a Vagrant box that contains `pipework`, you can get it by cloning the repository and starting the Vagrant VM like so:

```
$ git clone https://github.com/how2dock/docbook
$ cd ch03/simple
$ vagrant up
$ vagrant ssh
vagrant@foobar-server:~$ cd /vagrant
vagrant@foobar-server:/vagrant$ ls
pipework  Vagrantfile
```

Let's start a container without any network using `--net=none` like shown in Recipe 3.2.

```
$ docker run -it --rm --net none --name cookbook ubuntu:14.04 bash
root@556d04d8637e:/# ip -d link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode ...
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00 promiscuity 0
```

In another terminal on the Docker host, let's use `pipework` to create a bridge `br0`, assign an IP address to the container and set the correct routing from the container to the bridge.

```
$ cd /vagrant
$ sudo ./pipework br0 cookbook 192.168.1.10/24@192.168.1.254
Warning: arping not found; interface may not be immediately reachable
```

In the container verify that the interface `eth1` is up and that the routing is in place:

```
root@556d04d8637e:/# ip -d link show eth1
7: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP ...
    link/ether a6:95:12:b9:8f:55 brd ff:ff:ff:ff:ff:ff promiscuity 0
    veth
root@556d04d8637e:/# route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
```

```
default         192.168.1.254   0.0.0.0         UG    0      0        0 eth1
192.168.1.0     *               255.255.255.0   U     0      0        0 eth1
```

Now if you list the network links on the host, you will see a bridge `br0` in addition to the default `docker0` bridge, and if you list the bridges (using `brctl` from the `bridge-utils` package), you will see the virtual ethernet interface attached to `br0` by `pipework`.

```
$ ip -d link show
...
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mode DEFAULT gro
    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff promiscuity 0
    bridge
8: br0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mode DEFAULT group d
    link/ether 22:43:24:f5:91:7e brd ff:ff:ff:ff:ff:ff promiscuity 0
    bridge
10: veth1pl31668: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast master br0 state D
    link/ether 22:43:24:f5:91:7e brd ff:ff:ff:ff:ff:ff promiscuity 1
    veth
$ brctl show
bridge name     bridge id               STP enabled     interfaces
br0             8000.224324f5917e       no              veth1pl31668
docker0         8000.000000000000       no
```

At this stage you can reach the container from the host or reach any other containers from the container `cookbook`. However, if you try to reach outside the Docker host you will notice that it will not work. There is no NAT masquerading rule in place -rule that is added automatically by Docker when you use the defaults-. Add the rule manually on the Docker host and try to ping `8.8.8.8` (for example) from the container interactive terminal.

```
# iptables -t nat -A POSTROUTING -s 192.168.0.0/16 -j MASQUERADE
```

On the container verify that you can reach outside your Docker host:

```
root@556d04d8637e:/# ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=61 time=22.6 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=61 time=23.8 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=61 time=23.9 ms
```

`pipework` can do a lot more, make sure to check the README file and don't hesitate to pick inside the bash script to gain an even greater understanding of networking namespace.

Discussion

While `pipework` is extremely powerfull and allowed us to build a proper networking stack for a container started with `--net=none`, it also hid some of the details of manipulating the container network namespace. If you read the code of `pipework`

you will see what it does. A very good explanation is also available in the Docker documentation and is a very good exercise, both in networking and containers. I highly recommend it.

This discussion is not about `pipework` specifically, it aims to show you all the steps necessary to build a networking stack for a container. It is extremely useful to obtain a better understanding of container networking and reverse engineer how Docker works.

Let's look back at this single `pipework` command:

```
$ sudo ./pipework br0 cookbook 192.168.1.10/24@192.168.1.254
```

It did several almost magical things at once:

- It created a bridge `br0` on the host
- It assigned IP address `192.168.1.254` to it
- It created a network interface inside the container and assigned it IP address `192.168.1.10`
- Finally it added a route inside the container setting up the bridge as the default gateway.

Let's do it step by step but without `pipework` this time. To get started, let's add a bridge `br0` and give it the IP `192.168.1.254`. If you have worked on virtual machine virtualization before, this should be very familiar. If not, follow along, we create the bridge with the `brctl` utility, we use `ip` to add the IP address to the bridge and we finish by bringing the bridge up.

```
$ sudo brctl addbr br0
$ sudo ip addr add 192.168.1.254/24 dev br0
$ sudo ip link set dev br0 up
```

If you want to delete this bridge and start over, just bring it down and delete it:

```
$ sudo ip link set br0 down
$ sudo brctl delbr br0
```

The tricky part compared to full network virtualization comes from the fact that we are dealing with containers and that the networking stack is in fact a different network namespace on the host. To assign network interfaces to the container you need to assign an interface to the network namespace that the container will use. The interfaces that you can assign to a specific network namespace are virtual ethernet interface pairs. These pairs act as a pipe, with one end of the pipe in the container namespace an the other end on the bridge that we just created on the host.

Therefore let's create a `veth` pair *foo, bar* and attach *foo* to the bridge `br0`:

```
$ sudo ip link add foo type veth peer name bar
$ sudo brctl addif br0 foo
$ sudo ip link set foo up
```

The result can be seen with `ip -d link show`. A new bridge br0 and `foo` interface of type `veth` attached to it.

```
$ ip -d link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00 promiscuity 0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNOWN mode DEFAULT gr
    link/ether 08:00:27:98:a7:ad brd ff:ff:ff:ff:ff:ff promiscuity 0
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mode DEFAULT gro
    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff promiscuity 0
    bridge
6: br0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT group defau
    link/ether ee:7d:7e:f7:6f:18 brd ff:ff:ff:ff:ff:ff promiscuity 0
    bridge
8: foo: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast master br0 state UP mode DEFAU
    link/ether ee:7d:7e:f7:6f:18 brd ff:ff:ff:ff:ff:ff promiscuity 1
    veth
$ brctl show
bridge name     bridge id               STP enabled     interfaces
br0             8000.ee7d7ef76f18       no              foo
docker0         8000.000000000000       not
```

Do not call each end of your veth pair the traditional `eth0` or `eth1` as it could conflict with existing physical interfaces on the host.

To complicate things a bit, when we started our container with `--net=none` it did creat a network namespace for it, just that there was nothing in it except teh loopback device. Now that we want to configure it (e.g adding an interface, setting up a route) we need to find the network namespace ID. Docker keeps its network namespaces in `/var/run/docker/netns` which is a non-default location. To be able to use the `ip` tool properly we are going to do a little non-conventional hack and symlink `/var/run/docker/netns` to `/var/run/netns` which is the default location where the `ip` tool looks for network namespaces. Doing so we can list the existing network namespaces. Below we see that the namespace ID of our container is actually the container ID.

```
$ cd /var/run
$ sudo ln -s /var/run/docker/netns netns
$ sudo ip netns
c785553b22a1
$ NID=$(sudo ip netns)
```

Now, let's put the *bar* veth in the container namespace using `ip link set netns` and use `ip netns exec` to give it a name and a MAC address inside this namespace.

```
$ sudo ip link set bar netns $NID
$ sudo ip netns exec $NID ip link set dev bar name eth1
$ sudo ip netns exec $NID ip link set eth1 address 12:34:56:78:9a:bc
$ sudo ip netns exec $NID ip link set eth1 up
```

The final thing to do is to assign an IP address to *eth1* of the container and define a default route so that the container can reach the Docker host and beyond.

```
$ sudo ip netns exec $pid ip addr add 192.168.1.1/24 dev eth1
$ sudo ip netns exec $pid ip route add default via 192.168.1.254
```

That's it. At this stage your container should have the exact same networking stack than the one built with `pipework` earlier with a single command.

Remember that if you want to reach outside your container, you need to add the IP NAT masquerading rule:

```
$ sudo iptables -t nat -A POSTROUTING -s 192.168.0.0/16 -j MASQUERADE
```

See Also

- pipework has an extensive readme that covers multiple scenarios.
- How Docker networks containers.
- `ip netns` man page
- Introduction to linux nework namespace

3.6 Setting up a Custom Bridge for Docker

Problem

You would like to set up your own bridge for Docker to use instead of using the default.

Solution

Create a bridge and change the start-up options of the Docker daemon to use.

In the Recipe 3.5 solution section we saw how to create a full networking stack for a container started with the `--net=none` option. In that section we actually showed how to create a bridge. Let's re-use what we discussed there.

First let's turn off the Docker daemon, delete and create a bridge called `cookbook`:

```
$ sudo service docker stop
$ sudo brctl addbr cookbook
$ sudo ip link set cookbook up
$ sudo ip addr add 10.0.0.1/24 dev cookbook
```

Now that the bridge is up we can edit the Docker daemon configuration file and restart the daemon (e.g on Ubuntu).

```
$ sudo su
# echo 'DOCKER_OPTS="-b=cookbook"' >> /etc/default/docker
# service docker restart
```

You can start a container and list the IP addres assigned to it and check network connectivity.

```
root@c557cdb072ba:/# ip addr show eth0
10: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:0a:00:00:02 brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.2/24 scope global eth0
        ...
```

Automatically as expected, Docker also created the NAT rule for this bridge:

```
$ sudo iptables -t nat -L
...
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  10.0.0.0/24          anywhere
```

Discussion

While you can do this manually, there is nothing different between the bridge `cookbook` that we just created and the default `docker0` bridge.

If you wanted to change the IP range that Docker uses for the containers started with the default networking (i.e bridge) you could use the `--bip` option. You could also restrict this IP range with the `--fixed-cidr` option as well as set the MTU size with `--mtu`.

To bring down the bridge simply execute the following two commands:

```
$ sudo ip link set cookbook down
$ sudo brctl delbr cookbook
```

3.7 Using OVS with Docker

Problem

You know how to use your own bridge to network your Docker containers (see Recipe 3.6), but you would like to use the Open Vswitch virtual switch instead of the standard linux bridge. Maybe you want to build your own GRE or VXLAN based

overlay, or you want to build a software defined network solution with a network controller.

Solution

As of Docker 1.7, Open Vswitch is not yet support natively. You can use it, but you will need to use a tool like `pipework` (see Recipe 3.5) or a manual process to build the network stack of the containers. It should be supported in future version of Docker network (see Recipe 3.9).

Use Open Vswitch(OVS) as your bridge and specify its name in the Docker daemon configuration file.

On your Docker host, start by installing the packages for OVS.

```
$ sudo apt-get -y install openvswitch-switch openvswitch-common
```

If you want a more recent version of Open Vswitch, you can build it from source relatively easily.

Now create a bridge and bring it up:

```
$ sudo ovs-vsctl add-br ovs-cookbook
$ sudo ip link set ovs-cookbook up
$ ifconfig
ovs-cookbook  Link encap:Ethernet  HWaddr 36:b1:d3:e5:fc:44
          inet addr:10.0.0.1  Bcast:0.0.0.0  Mask:255.255.255.0
                  ...
```

You are now ready to use `pipework` (see Recipe 3.5) to build the network stack of containers attached to this Openvswitch bridge. You will need to start containers without network stack (i.e `--net=none`), for example:

```
$ docker run -it --rm --name foobar --net=none ubuntu:14.04 bash
$ sudo su
# ./pipework ovs-cookbook foobar 10.0.0.10/24@10.0.0.1
# ovs-vsctl list-ports ovs-cookbook
veth1pl31350
```

And your container will now have a network interface:

```
root@8fda6e33eb88:/# ifconfig
eth1      Link encap:Ethernet  HWaddr 52:fe:9f:78:b7:fc
          inet addr:10.0.0.10  Bcast:0.0.0.0  Mask:255.255.255.04
....
```

Of course you could also create the interface by hand using `ip netns` like we did in the discussion section of Recipe 3.5.

See Also

- The OpenVswitch website

3.8 Building a GRE Tunnel Between Docker Hosts

Problem

You need to have network connectivity between containers on multiple hosts using their own IP addresses.

Solution

Build a GRE tunnel to encapsulate IPv4 in IPv4 and provide a route between containers using their private addresses. To show case this technique we are going to bring up two docker hosts and setup the network configuration that you can see in the diagram below.

Host 1 has IP address `192.168.33.11`, we will give the `docker0` bridge IP address `172.17.0.1` and create a GRE tunnel endpoint with IP address `172.17.0.2`. Docker will give containers addresses in the `172.17.0.0/17` network.

Host 2 has IP address `192.168.33.12`, we will give the `docker0` bridge IP address `172.17.128.1` and create a GRE tunnel endpoint with IP address `171.17.128.2`. Docker will give containers addresses in the `172.17.128.0/17` network.

Splitting a `/16` network in two `/17` network and assigning each subnet to the two different hosts assures us that containers will not get conflicting IP addresses.

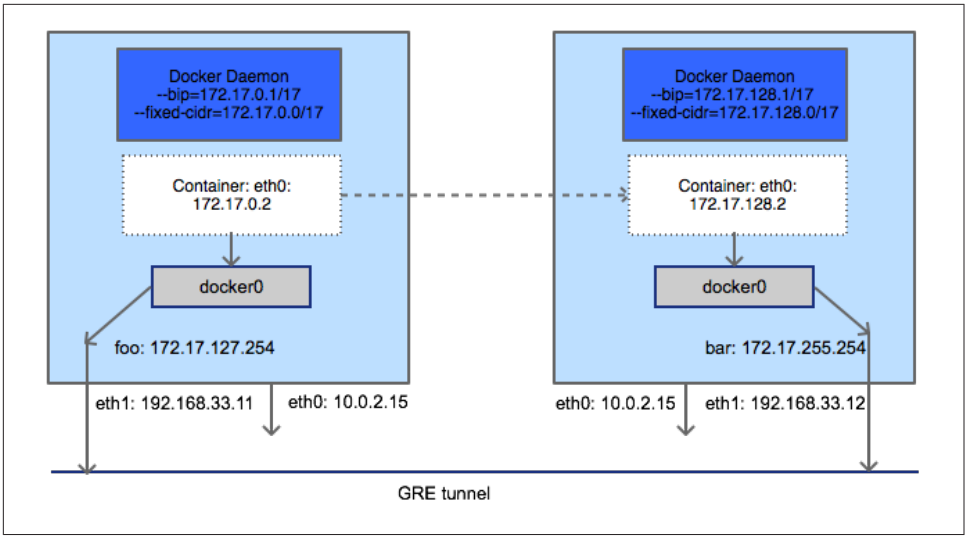


Figure 3-2. Network Diagram of a Two Hosts GRE Tunnel Overlay.

You can start this configuration with this Vagrantfile. Each host has the latest stable version of Docker and two network interfaces: one NAT interface that gives outbound connectivity and one interface on a private network.

The first thing to do to avoid any issues is to stop the Docker engine and remove that `docker0` bridge which was started during the Docker provisioning step. You will need to do this on all your hosts.

```
$ sudo su
# service docker stop
# ip link set docker0 down
# ip link del docker0
```

Now you can create a GRE tunnel between the two hosts. You do not need Open Vswitch for this, you can just use the `ip` tool. If you used the Vagrantfile mentioned earlier, on your first host with IP `192.168.33.11` do the following.

```
# ip tunnel add foo mode gre local 192.168.33.11 remote 192.168.33.12
# ip link set foo up
# ip addr add 172.17.127.254 dev foo
# ip route add 172.17.128.0/17 dev foo
```

If you did not use the Vagrantfile mentioned, just replace the IP addresses for the `local` and `remote` endpoints in the `ip tunnel` command above with the IP addresses of your two Docker hosts. In the previous four commands, with create a GRE tunnel that we named `foo`. We brought the interface up and assigned an IP address to it. Then we setup a route which sends all `172.17.128.0/17` traffic in the tunnel.

On your second host, repeat the previous step to create the other end of the tunnel. We call this other end `bar` and setup a route that sends all `172.17.0.0/17` traffic over in the tunnel.

```
# ip tunnel add bar mode gre local 192.168.33.12 remote 192.168.33.11
# ip link set bar up
# ip addr add 172.17.255.254 dev bar
# ip route add 172.17.0.0/17 dev bar
```

Once the tunnel is up, verify that you can ping back and forth using the tunnel. Now let's bring up Docker on both hosts. Before we do this we need to configure each Docker daemon so that it uses the appropriate subnets for its containers and uses the correct IP address for the `docker0` bridge. To do this we edit the Docker daemon configuration file and use the `--bip` and `--fixed-cidr` options.

On host #1 that would be:

```
# echo DOCKER_OPTS=\"--bip=172.17.0.1/17 --fixed-cidr=172.17.0.0/17\" >> /etc/default/docker
```

And on host #2 that would be:

```
# echo DOCKER_OPTS=\"--bip=172.17.128.1/17 --fixed-cidr=172.17.128.0/17\" >> /etc/default/docker
```

If you have chosen a different partitioning schema or have more than two hosts, repeat this with your values.

Since Docker will turn on IP forwarding, all traffic on `docker0` will get forwarded to `foo` and `bar`, so there is no need to attach the tunnel endpoints to any bridges.

All that is left now is to restart Docker, then you can start one container on each host and you will see that they have direct network connectivity using the private IP address given to them by Docker.

Discussion

There are multiple ways to build a networking overlay for your Docker host, Docker network (see Recipe 3.9) which should be released in Docker 1.8 allows you to build VXLAN overlays using Docker built-in features. Other third party solutions exist like Weave (see Recipe 3.11) or Flannel (see Recipe 3.13). As the Docker plugin framework matures, this type of functionality will change quite significantly. For instance Weave and Flannel will be available as Docker plugins, instead of separate network setup.

See Also

- This recipe was inspired by a blog post from Vincent Viallet on wiredcraft

3.9 Networking Containers on Multiple Hosts with Docker Network

Problem

While you could build tunnels between your Docker hosts manually (see Recipe 3.8), you want to take advantage of the new Docker network feature and use a VXLAN overlay.

Solution

Docker network is a new feature, currently available on the Docker experimental channel. It should be released with Docker 1.8 in August 2015. The recipe provided here is a preview which will give you a taste of what you will be able to find in future Docker releases.

As of this writing, Docker network relies on Consul for key-value store, uses Serf for discovery of the nodes and builds a VXLAN overlay using the standard linux bridge. Since Docker network is under heavy development these requirements and methods may change in the near future.

As common in this book, I prepared a Vagrantfile that will start three virtual machines. One will act as a consul server, and the other two will act as Docker host. The experimental version of Docker is installed on the two Docker hosts, while the latest stable version of Docker is installed on the machine running the Consul server.

The setup is as follows:

- `consul-server` is the Consul server node, based on Ubuntu 14.04, this has IP 192.168.33.10
- `net-1` is the first Docker host based on Ubuntu 14.10, this has IP 192.168.33.11
- `net-2` is the second Docker host based on Ubuntu 14.10, this has IP 192.168.33.12

The following diagram illustrates this setup.

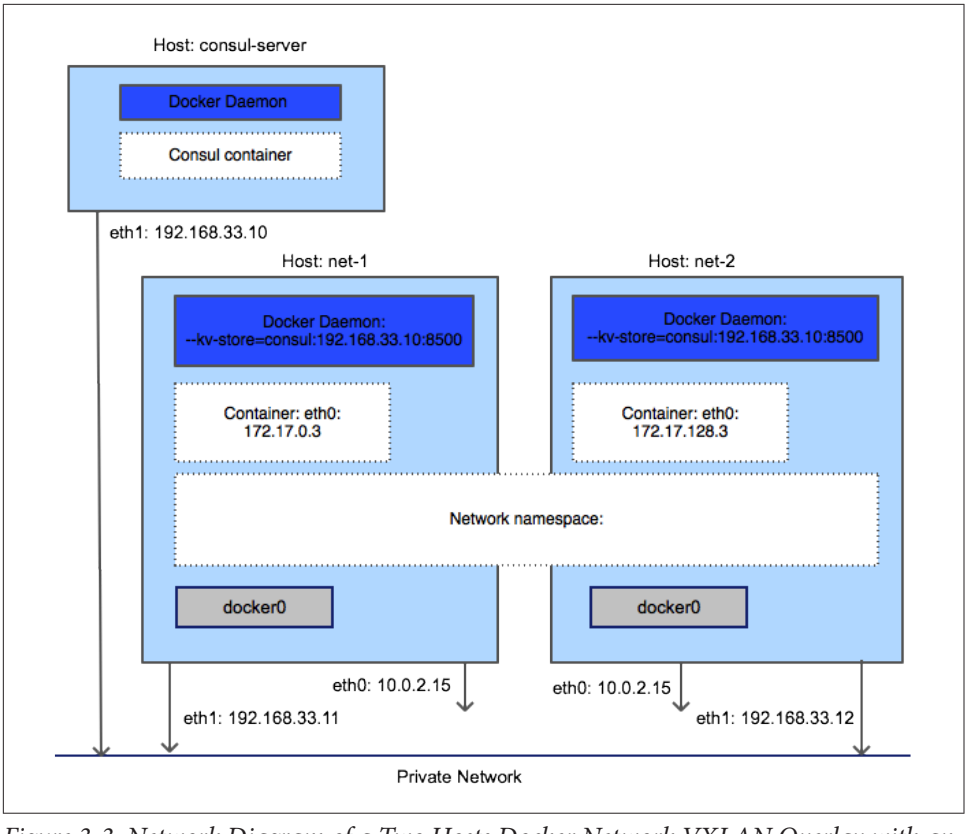


Figure 3-3. Network Diagram of a Two Hosts Docker Network VXLAN Overlay with an Additional Consul Node.

Clone the repository, change to the `docbook/ch03/networks` directory and let Vagrant do the work.

```
$ git clone https://github.com/how2dock/docbook/
$ cd docbook/ch03/network
$ vagrant up
$ vagrant status
Current machine states:

consul-server             running (virtualbox)
net-1                     running (virtualbox)
net-2                     running (virtualbox)
```

You are now ready to SSH to the Docker hosts and start containers. You will see that you are running the Docker experimental version `-dev`, the version number that you will see may be different depending on where we are in the release cycle…<

```
$ vagrant ssh net-1
vagrant@net-1:~$ docker version
```

```
Client version: 1.8.0-dev
...<snip>...
```

Check that Docker network is functional by listing the default networks:

```
vagrant@net-1:~$ docker network ls
NETWORK ID          NAME                TYPE
4275f8b3a821        none                null
80eba28ed4a7        host                host
64322973b4aa        bridge              bridge
```

No services has been published so far, so the `docker service ls` will return an empty list:

```
$ docker service ls
SERVICE ID          NAME                NETWORK             CONTAINER
```

Start a container and check the content of `/etc/hosts`.

```
$ docker run -it --rm ubuntu:14.04 bash
root@df479e660658:/# cat /etc/hosts
172.21.0.3      df479e660658
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
172.21.0.3      distracted_bohr
172.21.0.3      distracted_bohr.multihost
```

In a separate terminal on `net-1` list the networks again. You will see that the *multihost* overlay now appears. The overlay network *multihost* is your default network. This was setup by the Docker daemon during the Vagrant provisioning. Check `/etc/default/docker` to see the options that were set.

```
vagrant@net-1:~$ docker network ls
NETWORK ID          NAME                TYPE
4275f8b3a821        none                null
80eba28ed4a7        host                host
64322973b4aa        bridge              bridge
b5c9f05f1f8f        multihost           overlay
```

Now in a separate terminal, SSH to `net-2`, check the network and services. The networks will be the same, and the default network will also be *multihost* of type overlay. But the service will show the container started on `net-1`:

```
$ vagrant ssh net-2
vagrant@net-2:~$ docker service ls
SERVICE ID          NAME                NETWORK             CONTAINER
b00f2bfd81ac        distracted_bohr     multihost           df479e660658
```

Start a container on `net-2` and check the `/etc/hosts`.

```
vagrant@net-2:~$ docker run -ti --rm ubuntu:14.04 bash
root@2ac726b4ce60:/# cat /etc/hosts
172.21.0.4      2ac726b4ce60
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
172.21.0.3      distracted_bohr
172.21.0.3      distracted_bohr.multihost
172.21.0.4      modest_curie
172.21.0.4      modest_curie.multihost
```

You will see not only the container that you just started on `net-2` but also the container that you started earlier on `net-1`. And of course you will be able to ping each container.

Discussion

The solution discussed above used the default network overlay that was configured at startup time by specifying in the `/etc/default/docker` configuration file. You can however used a non default overlay network. This means that you can create as many overlays as you want and that container started in each overlay will be isolated.

In the previous test we started containers with regular options `-ti --rm` and these containers got placed automatically in the default network which was set to be the *multihost* network of type overlay.

But you could create your own overlay network and start containers in it. Let's try this, first create a new overlay network with the `docker network create` command.

On one of your Docker hosts, `net-1` or `net-2` do:

```
$ docker network create -d overlay foobar
8805e22ad6e29cd7abb95597c91420fdcac54f33fcdd6fbca6dd4ec9710dd6a4
$ docker network ls
NETWORK ID          NAME                TYPE
a77e16a1e394        host                host
684a4bb4c471        bridge              bridge
8805e22ad6e2        foobar              overlay
b5c9f05f1f8f        multihost           overlay
67d5a33a2e54        none                null
```

Automatically, the second host will also see this network. To start a container on this new network, simply use the `--publish-service` option of `docker run` like so:

```
$ docker run -it --rm --publish-service=bar.foobar.overlay ubuntu:14.04 bash
```

You could directly start a container with a new overlay using the `--publish-service` option and it will create the network automatically.

Check the docker services now:

```
$ docker service ls
SERVICE ID          NAME                NETWORK             CONTAINER
b1ffdbfb1ac6        bar                 foobar              6635a3822135
```

Repeat the getting started steps, by starting another container in this new overlay on the other host, check the `/etc/hosts` file and try to ping each container.

3.10 Diving Deeper Into The Docker Network Namespaces Configuration

Problem

You would like to understand better what Docker network does, especially where do the VXLAN interfaces exist.

Solution

This new Docker multihost networking is made possible via VXLAN tunnels and the use of network namespaces. In Recipe 3.5 we already saw how to explore and manipulate network namespaces. The same can be done for Docker network.

Check the designdocumentation for all the details. But to explore these concepts a bit, nothing beats an example.

Discussion

With a running container in one overlay, check the network namespace:

```
$ docker inspect -f '' 6635a3822135
/var/run/docker/netns/6635a3822135
```

This is a none default location for network namespaces which might confuse things a bit. So let's become root, head over to this directory that contains the network namespaces of the containers and check the interfaces:

```
$ sudo su
root@net-2:/home/vagrant# cd /var/run/docker/
root@net-2:/var/run/docker# ls netns
6635a3822135
8805e22ad6e2
```

To be able to check the interfaces in those network namespace using `ip` command, just create a symlink for `netns` that points to `/var/run/docker/netns`:

```
root@net-2:/var/run# ln -s /var/run/docker/netns netns
root@net-2:/var/run# ip netns show
6635a3822135
8805e22ad6e2
```

The two namespace ID return are the ones of the running container on that host and the one of the actual overlay network the container is in. Let's check the interfaces in the container:

```
root@net-2:/var/run/docker# ip netns exec 6635a3822135 ip addr show eth0
15: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:b3:91:22:c3 brd ff:ff:ff:ff:ff:ff
    inet 172.21.0.5/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:b3ff:fe91:22c3/64 scope link
       valid_lft forever preferred_lft forever
```

Indeed we get back the network interface of our running container, same MAC address, same IP. If we check the links of the overlay namespace we see our vxlan interface and the VLAN ID being used.

```
root@net-2:/var/run/docker# ip netns exec 8805e22ad6e2 ip -d link show
...<snip>...
14: vxlan1: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master br0 state UNKNOWN mode DEFAULT g
    link/ether 7a:af:20:ee:e3:81 brd ff:ff:ff:ff:ff:ff promiscuity 1
    vxlan id 256 srcport 32768 61000 dstport 8472 proxy l2miss l3miss ageing 300
    bridge_slave
16: veth2: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast master br0 state UP mode DEFAULT grou
    link/ether 46:b1:e2:5c:48:a8 brd ff:ff:ff:ff:ff:ff promiscuity 1
    veth
    bridge_slave
```

If you sniff packets on these interfaces you will see the traffic between your containers.

3.11 Running Containers on a Weave Network

Contributed by Fintan Ryan

Problem

Networking of Docker containers across multiple hosts can be solved by the ambassador pattern. However you would like every container to get a routable IP across the Docker hosts that you are using.

Solution

Use Weave from Weaveworks. To help you test Weave we have created a Vagrantfile which starts two hosts running Ubuntu 14.04, installs docker, weave and two example containers. You can test it as follows

```
$ git clone https://github.com/how2dock/docbook.git
$ cd ch03/weavesimple
$ vagrant up
$ ./launch-simple-demo.sh
```

You can now log into our first host, weave-gs-01 and connect to our container on the second host

```
$ vagrant ssh weave-gs-01
$ CONTAINER=$(sudo docker ps | grep weave-gs-ubuntu-curl | awk '{print $1}')
$ sudo docker attach $CONTAINER
$ curl 10.3.1.1

{
    "message" : "Hello World",
    "date" : "2015-03-13 15:03:52"
}
```

Discussion

Weave allows you to connect to your containers in the same manner as you are already famiilar with for your existing infrastructure. Weave creates an overlay network

On your first host, `weave-gs-01`, you have launched a Weave router container. On your second host, `weave-gs-02`, you launched another Weave router container with the IP address of your first host. This command tells the Weave on `weave-gs-02` to peer with the Weave on `weave-gs-01`.

Any containers you launch after this using weave are visible within the Weave network to all other containers no matter what host they are on.

See Also

- Weave Getting Started Guides

3.12 Running a Weave Network on AWS

Contributed by Fintan Ryan

Problem

You would like to use Weave and WeaveDNS on instances deployed in AWS.

Solution

As prequisites you will need

- An account on AWS
- A set of access and secret API keys
- Ansible installed, with the boto package

To help you experiment with Weave on AWS we have created an ansible playbook which starts two hosts running Ubuntu 14.04 on ec2, installs docker and installs weave. We have provided a second playbook specifically for launching a simple demo application using HAProxy as a load balancer in front of containers across our two hosts.

```
$ git clone https://github.com/how2dock/docbook.git
$ cd ch03/weaveaws
$ ansible-playbook setup-weave-ubunu-aws.yml
```

Example 3-1.

You can change your AWS region and AMI in the file `ansible_aws_variables.yml`

To launch your containers call

```
$ ansible-playbook launch-weave-haproxy-aws-demo.yml
```

We have provided a script to quickly connect to your HAProxy container and cycle through a number of requests. Each container will return its hostname as part of its JSON output.

```
$ ./access-aws-hosts.sh

Connecting to HAProxy with Weave on AWS demo

{
    "message" : "Hello Weave - HAProxy Example",
    "hostname" : ws1.weave.local",
    "date" : "2015-03-13 11:23:12"
}

{
    "message" : "Hello Weave - HAProxy Example",
    "hostname" : ws4.weave.local",
    "date" : "2015-03-13 11:23:12"
}
```

```
{
    "message" : "Hello Weave - HAProxy Example",
    "hostname" : ws5.weave.local",
    "date" : "2015-03-13 11:23:12"
}
....
```

Discussion

Using Weave we have placed HAProxy as a load balancing solution in front of a number of containers running a simple application distributed across a number of hosts.

See Also

- Weave Getting Started Guides

3.13 Deploying flannel Overlay Between Docker Hosts

Contributed by Eugene Yakubovich

Problem

You want containers on different hosts to communicate with each other without port mapping.

Solution

Use flannel to create an overlay network for containers. Each container will be assigned an IP that can be reachable from other hosts. Start out with the following Vagrantfile:

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
        config.vm.box = "ubuntu/trusty64"
        config.vm.provider "virtualbox" do |vb|
                vb.customize ["modifyvm", :id, "--memory", "1024"]
        end

        config.vm.provision :shell, :path => "bootstrap.sh"

        config.vm.define "master" do |master|
                master.vm.network "private_network", ip: "192.168.33.10"
        end

        config.vm.define "worker" do |worker|
                worker.vm.network "private_network", ip: "192.168.33.11"
```

```
        end
    end
```

This will define two virtual machines. The "master" will run a key/value store (etcd) that flannel uses for coordination. bootstrap.sh, shown below, is used to install suitable version of Docker, etcd and flannel.

```
ETCD_URL=https://github.com/coreos/etcd/releases/download/v0.4.6/etcd-v0.4.6-linux-amd64.tar.gz
FLANNEL_URL=https://github.com/coreos/flannel/releases/download/v0.3.0/flannel-0.3.0-linux-amd64.t

# Install latest Docker
curl -sSL https://get.docker.com/ubuntu/ | sudo sh
sudo gpasswd -a vagrant docker
sudo service docker stop

# Docker doesn't delete its bridge when stopped.
# Once flannel is started, it will be re-created with different set of options
sudo ip link del docker0

# Download and untar etcd and flannel
sudo mkdir /opt/coreos
cd /opt/coreos
sudo curl -L $ETCD_URL | tar xzv
sudo curl -L $FLANNEL_URL | tar xzv
sudo chown -R vagrant:vagrant /opt/coreos
```

Next, "vagrant ssh master" and start etcd in the background:

```
$ cd /opt/coreos/etcd-v0.4.6-linux-amd64
$ nohup ./etcd &
```

Before starting flannel daemon, write the overlay network configuration into etcd. Be sure to pick a subnet range that does not conflict with other IP addresses.

```
$ ./etcdctl set /coreos.com/network/config '{ "Network": "10.100.0.0/16" }'
```

Now start flannel daemon. Notice that --iface option specifies the IP of the private network given in Vagrantfile. flannel will forward encapsulated packets over this interface.

```
$ cd /opt/coreos/flannel-0.3.0
$ sudo ./flanneld --iface=192.168.33.10 --ip-masq &
$ sudo ./mk-docker-opts.sh -c -d /etc/default/docker
```

flannel will acquire a lease for a /24 subnet to be assigned to docker0 bridge. The acquired subnet will be written out to /run/flannel/subnet.env file. mk-docker-opts.sh utility converts this file into a set of command line options for Docker daemon.

Finally, start the Docker daemon. Verify that everything is running as expected by checking the IP of docker0 bridge. It should be within the 10.100.0.0/16 range.

```
$ sudo service docker start
$ ifconfig docker0
docker0   Link encap:Ethernet  HWaddr 56:84:7a:fe:97:99
```

```
            inet addr:10.100.63.1  Bcast:0.0.0.0  Mask:255.255.255.0
...
```

Over on the "worker" node repeat the procedure of bringing up flannel. Since etcd is running on "master", do not launch it on this node. Instead point flannel to use the instance running on "master".

```
$ sudo ./flanneld --etcd-endpoints=http://192.168.33.10:4001 --iface=192.168.33.11 --ip-masq &
$ sudo ./mk-docker-opts.sh -c -d /etc/default/docker
$ sudo service docker start
```

With both nodes bootstrapped into the flannel network, bring up a simple busybox container on each of the nodes. The containers will have an IP pingable from the remote container.

Discussion

All flannel members use etcd for coordination. Upon startup, flannel daemon reads the overlay network configuration from etcd as well as all other subnets in use by other nodes. It then picks an used subnet (/24 by default) and attempts to claim it by creating a key for it in etcd. If the creation succeeds, the node has acquired a 24 hour lease on the subnet. The associated value contains the host's IP.

Next, flannel uses the TUN device to create a flannel0 interface. IP fragments routed to flannel0 from docker0 bridge will be delivered to the flannel daemon. It encapsulates each IP fragment in a UDP packet and uses the subnet information from etcd to forward it to the correct host. The receiving end unwraps the IP fragment from its encapsulation and sends it to docker0 via the TUN device.

flannel continues to watch etcd for changes in the memberships to keep its knowledge current. Additionally, the daemon will renew its lease an hour before its expiration.

3.14 Using an Ambassador Container to Expose Services

Problem

Solution

Discussion

CHAPTER 4

Docker Configuration and Development

This chapter consists of recipes focused on Docker development, configuration and API usage. It is not meant to be a developer guide or a programming guide. The recipes will illustrates several concepts such as how to build a docker binary. I also plan to illustrate a few namespace concepts. You can send me suggestions at *how2dock@gmail.com*

4.1 Compiling Your Own Docker Binary From Source

Problem

You would like to do develop the Docker software and build your own Docker binary

Solution

Use Git to clone the Docker repository from GitHub and use the `Makefile` to create your own binary.

Docker is built within Docker. In a Docker Host you can clone the Docker repository and use the Makefile rules to build a new binary. This binary is obtained by running a privileged Docker container. The `Makefile` containes several targets including a `binary` target:

```
$ cat Makefile
...
default: binary

all: build
	$(DOCKER_RUN_DOCKER) hack/make.sh
```

```
binary: build
        $(DOCKER_RUN_DOCKER) hack/make.sh binary
...
```

Therefore it is as easy as `sudo make binary`:

The `hack` directory in the root of the Docker repository has been moved to the `project` directory. Therefore the `make.sh` script is in fact at `project/make.sh`. It uses scripts for each bundle that are stored in `project/make/` directory.

```
$ sudo make binary
...
docker run --rm -it --privileged \
                    -e BUILDFLAGS -e DOCKER_CLIENTONLY -e DOCKER_EXECDRIVER \
                    -e DOCKER_GRAPHDRIVER -e TESTDIRS -e TESTFLAGS \
                    -e TIMEOUT \
                    -v "/tmp/docker/bundles:/go/src/github.com/docker/docker/bundles" \
                    "docker:master" hack/make.sh binary

---> Making bundle: binary (in bundles/1.4.1-dev/binary)
Created binary: \
/go/src/github.com/docker/docker/bundles/1.4.1-dev/binary/docker-1.4.1-dev
```

We see that the `binary` target of the `Makefile` will launch a privileged Docker container from the `docker:master` image, with a set of environment variables, a volume mount and call to the `hack/make.sh binary` command.

With the current state of Docker development, the new binary will be located in the `bundles/1.4.1-dev/binary/` directory.

Discussion

To ease this process, you can clone the repository that accompanies this cookbook. A Vagrantfile is provided which starts an Ubuntu 14.04 virtual machine, installs the latest stable Docker release and clones the Docker repository.

```
$ git clone https://github.com/how2dock/docbook
$ cd docbook/ch04/compile/
$ vagrant up
```

Once the machine is up, `ssh` to it and go to the `/tmp/docker` directory which should have been created during the Vagrant provisioning process. Then run `make`. The first time you run the Makefile, the stable Docker installed on the machine will pull the base image being used by the Docker build process `ubuntu:14.04`, then build the `docker:master` image defined in the `/tmp/docker/Dockerfile`. This can take a bit of time the first time you do it.

```
$ vagrant ssh
$ cd /tmp/docker
$ sudo make binary
docker build -t "docker:master" .
Sending build context to Docker daemon 55.95 MB
Sending build context to Docker daemon
Step 0 : FROM ubuntu:14.04
...
```

Once this completes you will have a new Docker binary.

```
$ cd bundles/1.4.1-dev/binary/docker
$ ls
docker  docker-1.4.1-dev  docker-1.4.1-dev.md5  docker-1.4.1-dev.sha256
```

See Also

- How to contribute to Docker on GitHub

4.2 Running the Docker Test Suite for Docker Development

Problem

You have made some changes to the Docker source and you have successfully built a new binary. You also need to make sure that you pass all the tests.

Solution

Use the Makefile `test` target to run the four sets of tests present in the Docker source. Alternatively pick only the set of tests that matters to you.

```
$ cat Makefile
...
test: build
        $(DOCKER_RUN_DOCKER) hack/make.sh binary cross \
                                          test-unit test-integration \
                                          test-integration-cli test-docker-py
test-unit: build
        $(DOCKER_RUN_DOCKER) hack/make.sh test-unit

test-integration: build
        $(DOCKER_RUN_DOCKER) hack/make.sh test-integration

test-integration-cli: build
        $(DOCKER_RUN_DOCKER) hack/make.sh binary test-integration-cli

test-docker-py: build
```

```
        $(DOCKER_RUN_DOCKER) hack/make.sh binary test-docker-py
...
```

You can see in the Makefile that you can choose which set of tests you want to run. If you run all of them with `make test` it will also build the binary.

```
$ sudo make test
....
---> Making bundle: test-docker-py (in bundles/1.4.1-dev/test-docker-py)
+++ exec docker --daemon --debug --storage-driver vfs \
               -exec-driver native \
               --pidfile \
               /go/src/github.com/docker/docker/bundles/1.4.1-dev/test-docker-py/docker.pid
..................................................
----------------------------------------------------------------------
Ran 56 tests in 75.366s

OK
```

Depending on tests coverage, if all the tests pass, you have some confidence that your new binary works.

See Also

- Official Docker development environment documentation

4.3 Replacing Your Current Docker Binary With a New One

Problem

You have built a new Docker binary, ran the unit and integration tests using the recipes described at Recipe 4.1 and Recipe 4.2. Now you would like to use this new binary on your host.

Solution

Starting from within the virtual machine setup in Recipe 4.1.

Stop the current Docker daemon. On Ubuntu 14.04, edit the `/etc/default/docker` file to uncomment the `DOCKER` variable that defines where to find the binary and set it to `DOCKER="/usr/local/bin/docker"`. Copy the new binary to `/usr/local/bin/docker` and finally restart the Docker daemon.

```
$ pwd
/tmp/docker
$ sudo service docker stop
docker stop/waiting
$ sudo vi /etc/default/docker
```

```
$ sudo cp bundles/1.4.1-dev/binary/docker-1.4.1-dev /usr/local/bin/docker
$ sudo cp bundles/1.4.1-dev/binary/docker-1.4.1-dev /usr/bin/docker
$ sudo service docker restart
stop: Unknown instance:
$ docker version
Client version: 1.4.1-dev
Client API version: 1.16
Go version (client): go1.4
Git commit (client): a83e904
OS/Arch (client): linux/amd64
Server version: 1.4.1-dev
Server API version: 1.16
Go version (server): go1.4
Git commit (server): a83e904
```

You are now using the latest Docker version from the master development branch (i.e master branch at Git commit a83e904 at the time of this writing).

Discussion

The Docker bootstrap script used in the Vagrant virtual machine provisioning installs the latest stable version of Docker with:

```
sudo curl -sSL https://get.docker.com/ubuntu/ | sudo sh
```

This puts the Docker binary in `/usr/bin/docker`. This may conflict with your new binary installation. Either remove it or replace it with the new one if you see any conflicts when running `docker version`.

4.4 Using `nsenter`

Problem

You would like to enter a container for debugging purposes, you are using a Docker version older than 1.3.1 or you do not want to use the `docker exec` command.

Solution

Use `nsenter`. Starting with Docker 1.3, `docker exec` allows you to easily enter a running container, hence there is no need to do things like running an SSH server and exposing port 22 or using the now deprecated `attach` command.

`nsenter` was created to solve the problem of entering the namespace (hence, *nsenter*) of a container prior to the availability of `docker exec`. Nonetheless it is a useful tool that merits a short recipe in this book.

Let's start a container that sleeps for the duration of this recipe, and for completeness purposes, let's enter the running container with `docker exec`

```
$ docker pull ubuntu:14.04
$ docker run -d --name sleep ubuntu:14.04 sleep 300
$ docker exec -ti sleep bash
root@db9675525fab:/#
```

`nsenter` gives the same result. Conveniently it comes as an image in Docker hub. Pull the image, run the container and use `nsenter`.

```
$ docker pull jpetazzo/nsenter
$ sudo docker run docker run --rm -v /usr/local/bin:/target jpetazzo/nsenter
```

At this time, it is useful to have a look at the Dockerfile for `nsenter` and check the CMD option. You will see that it runs a script called `installer`. This small bash script does nothing else but detect if a mount point exists at `/target`. If it does, it copies a script called `docker-enter` and a binary called `nsenter` to that mount point. In the `docker run` command, since we specified a volumes (i.e `-v /usr/local/bin:/target`), running the container will have the effect of copying `nsenter` on our local machine. Quite a nice trick with a powerfull effect:

```
$ which docker-enter nsenter
/usr/local/bin/docker-enter
/usr/local/bin/nsenter
```

Note that to copy the files in `/usr/local/bin` I run the container with `sudo`. If you do not want to use this mount point convenience, you can copy the files locally with a command like:

```
$ docker run --rm jpetazzo/nsenter cat /nsenter \
                    > /tmp/nsenter && chmod +x /tmp/nsenter
```

You are now ready to enter the container. You can pass a command, if you do not want to get an interactive shell in the container.

```
$ docker-enter sleep
root@db9675525fab:/#
$ docker-enter sleep hostname
db9675525fab
```

`docker-enter` is nothing else than a wrapper around `nsenter`. You could use `nsenter` directly after finding the process ID of the container with `docker inspect` like so:

```
$ docker inspect --format  sleep
9302
$ sudo nsenter --target 9302 --mount --uts --ipc --net --pid
root@db9675525fab:/#
```

Discussion

Starting with Docker 1.3, you do not need to use `nsenter`, use `docker exec` instead.

```
$ docker exec -h

Usage: docker exec [OPTIONS] CONTAINER COMMAND [ARG...]

Run a command in a running container

  -d, --detach=false         Detached mode: run command in the background
  --help=false               Print usage
  -i, --interactive=false    Keep STDIN open even if not attached
  -t, --tty=false            Allocate a pseudo-TTY
```

See Also

- GitHub page from Jerome Petazzo `nsenter` repository.

4.5 Introducing libcontainer

Problem

Solution

Discussion

4.6 Using `nsinit`

Problem

Solution

Discussion

4.7 Switching Execution Environment

Problem

Solution

Discussion

4.8 Accessing the Docker Daemon Remotely

Problem

The default Docker daemon listens on a local unix socket `/var/run/docker.sock` which is only accessible locally. However you would like to access your Docker host remotely, calling the Docker API from a different machine.

Solution

Switch the listening protocol that the Docker daemon is using by editing the configuration file in `/etc/default/docker` and issue remote API call.

In `/etc/default/docker` add a line that sets the *DOCKER_HOST* to use *tcp* on port 2375. Then restart the docker daemon with `sudo service docker restart`.

```
$ cat /etc/default/docker
...
# Use DOCKER_OPTS to modify the daemon startup options.
#DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4"
DOCKER_OPTS="-H tcp://127.0.0.1:2375"
...
```

You will then be able to use the docker client by specifying a host accessed using *tcp*.

```
$ docker -H tcp://127.0.0.1:2375 images
REPOSITORY      TAG             IMAGE ID        CREATED         VIRTUAL SIZE
ubuntu          14.04           04c5d3b7b065    6 days ago      192.7 MB
```

This method is unencrypted and un-authenticated. You should not use this on a publicly routable host. This would expose your Docker daemon to anyone. You will need to properly secure your Docker daemon if you want to do this in production (See Recipe 4.10)

Discussion

With the Docker daemon listening over *tcp* you can now use *curl* to make API calls and explore the response. This is a good way to learn the Docker remote API.

```
$ curl -s http://127.0.0.1:2375/images/json | python -m json.tool
[
    {
        "Created": 1418673175,
        "Id": "04c5d3b7b0656168630d3ba35d8889bdaafcaeb32bfbc47e7c5d35d2",
        "ParentId": "d735006ad9c1b1563e021d7a4fecfd384e2a1c42e78d8261b83d6271",
        "RepoTags": [
            "ubuntu:14.04"
        ],
        "Size": 0,
        "VirtualSize": 192676726
    }
]
```

Above, we pipe the output of the *curl* command through `python -m json.tool` to make the json object that is returned readable. And the `-s` option removes the information of the data transfer.

4.9 Exploring the Docker remote API to automate Docker tasks.

Problem

After being able to access the Docker daemon remotely (See Recipe 4.8), you want to explore the Docker remote API in order to write programs. This will allow you to automate Docker tasks.

Solution

The Docker remote API is fully documented. It is currently on version 1.16. It is a REST API, in the sense that it manipulates resources (e.g images and containers) through HTTP calls using various HTTP methods (e.g GET, POST, DELETE). The *attach* and *pull* APIs are not purely REST as noted in the documentation.

We already saw how you can make the Docker daemon listen on a *tcp* socket (Recipe 4.8) and use curl to make API calls.

Table 4-1. A sample of the API for container actions

Action on containers	HTTP method	URI
List containers	GET	/containers/json
Create container	POST	/containers/create
Inspect a container	GET	/containers/(id)/json
Start a container	POST	/containers/(id)/start
Stop a container	POST	/containers/(id)/stop
Restart a container	POST	/containers/(id)/restart
Kill a container	POST	/containers/(id)/kill
Pause a container	POST	/containers/(id)/pause
Remove a container	DELETE	/containers/(id)

Table 4-2. A sample of the API for image actions

Action on images	HTTP method	URI
List images	GET	/images/json

Action on images	HTTP method	URI
Create an image	POST	/images/create
Tag an image into a repository	POST	/images/(name)/tag
Remove an image	DELETE	/images/(name)
Search images	GET	/images/search

For example, let's download the Ubuntu 14.0 image from the public registry (a.k.a Docker Hub), create a container from that image and start it. Remove it and then remove the image. Note that in this toy example, running the container will cause it to exit immediately since we are not passing any commands.

```
$ curl -X POST -d "fromImage=ubuntu" -d "tag=14.04"
                                        http://127.0.0.1:2375/images/create
$ curl -X POST -H 'Content-Type: application/json'
               -d '{"Image":"ubuntu:14.04"}'
                            http://127.0.0.1:2375/containers/create
{"Id":"6b6bd46f483a5704d4bced62ff58a0ac5758fb0875ec881fa68f0e2be2f42681","Warnings":null}
$ docker ps
CONTAINER ID    IMAGE           COMMAND         CREATED         STATUS         ...
$ docker ps -a
CONTAINER ID    IMAGE           COMMAND         CREATED         STATUS         ...
6b6bd46f483a    ubuntu:14.04    "/bin/bash"     16 seconds ago                 ...
$ curl -X POST http://127.0.0.1:2375/containers/6b6bd46f483a/start
$ docker ps -a
CONTAINER ID    IMAGE           COMMAND         CREATED              ...
6b6bd46f483a    ubuntu:14.04    "/bin/bash"     About a minute ago   ...
```

Now let's clean things up

```
$ curl -X DELETE http://127.0.0.1:2375/containers/6b6bd46f483a
$ curl -X DELETE http://127.0.0.1:2375/images/04c5d3b7b065
[{"Untagged":"ubuntu:14.04"}
,{"Deleted":"04c5d3b7b0656168630d3ba35d8889bd0e9caafcaeb3004d2bfbc47e7c5d35d2"}
,{"Deleted":"d735006ad9c1b1563e021d7a4fecfd75ed36d4384e2a1c42e78d8261b83d6271"}
,{"Deleted":"70c8faa62a44b9f6a70ec3a018ec14ec95717ebed2016430e57fec1abc90a879"}
,{"Deleted":"c7b7c64195686444123ef370322b5270b098c77dc2d62208e8a9ce28a11a63f9"}
,{"Deleted":"511136ea3c5a64f264b78b5433614aec563103b4d4702f3ba7d4d2698e22c158"}
$ docker ps -a
CONTAINER ID    IMAGE           COMMAND         CREATED         STATUS   ...
$ docker images
REPOSITORY      TAG             IMAGE ID        CREATED         VIRTUAL SIZE
```

After enabling remote API access you can set the *DOCKER_HOST* variable to its HTTP endpoint, this relieves you from passing it to the `docker` command as a `-H` option. For example, instead of `docker -H http://127.0.0.1:2375 ps`, you can do `export DOCKER_HOST=tcp://127.0.0.1:2375` and you will be able to simply do `docker ps`.

Discussion

While you can of course use *curl*, or write your own client, existing Docker clients like *docker-py* (see Recipe 4.11) can ease calling the API.

The list of APIs presented in Table 4-1 and Table 4-2 is not exhaustive and you should check the complete API documentation for all API calls, query parameters and response examples.

4.10 Securing the Docker Deamon for Remote Access

Problem

You need to access your Docker daemon remotely and securely.

Solution

Setup a Transport Layer Security (TLS) based access to your Docker daemon. This will use public key cryptography to encrypt and authenticate communication between a Docker client and the Docker daemon that you have setup with TLS.

The basic steps to test this security feature is described on the Docker website. However, it shows how to create your own Certificate Authority (CA) and sign server and client certificates using the CA. In a properly setup infrastructure you will need to contact the CA that you use routinely and ask for server certificates.

To conveniently test this TLS setup, I created an image that contains a script which creates the CA and the server and client certificates and keys. You can use this image to create a container and generate all the needed files.

We start with an Ubuntu 14.04 machine, running the latest Docker version (see Recipe 1.1). Download the image and start a container. You will need to mount a volume from your host and bind mount it to the `/tmp/ca` inside the docker container. You will also need to pass the hostname as argument to running the container (in the example below <*hostname*>). Once you are done running the container, all CA, server and client keys and certificates will be available in your working directory.

```
$ docker pull runseb/dockertls
$ docker run -ti -v $(pwd):/tmp/ca runseb/dockertls <hostname>
```

```
$ ls
cakey.pem  ca.pem  ca.srl  clientcert.pem  client.csr  clientkey.pem
extfile.cnf  makeca.sh  servercert.pem  server.csr  serverkey.pem
```

Stop the running Docker daemon. Create a `/etc/docker` directory and a `~/.docker` directory. Copy the CA, server key and server certificates to `/etc/docker`. Copy the CA, client key and certificate to `~/.docker`.

```
$ sudo service docker stop
$ sudo mkdir /etc/docker
$ mkdir ~/.docker
$ sudo cp {ca,servercert,serverkey}.pem /etc/docker
$ cp ca.pem ~/.docker/
$ cp clientkey.pem ~/.docker/key.pem
$ cp clientcert.pem ~/.docker/cert.pem
```

Edit the `/etc/default/docker` (you need to be root) configuration file to specify `DOCKER_OPTS` like so (replace *test* with your own hostname):

```
DOCKER_OPTS="-H tcp://<test>:2376 --tlsverify --tlscacert=/etc/docker/ca.pem \
                                             --tlscert=/etc/docker/servercert.pem \
                                             --tlskey=/etc/docker/serverkey.pem"
```

Then restart the Docker service with `sudo service docker restart` and try to connect to the Docker daemon:

```
$ docker -H tcp://test:2376 --tlsverify images
REPOSITORY          TAG             IMAGE ID        CREATED         VIRTUAL SIZE
runseb/dockertls    latest          5ed60e0f6a7c    17 minutes ago  214.7 MB
```

Discussion

The `runseb/dockertls` convenience image is automatically built from the *https://github.com/how2dock/docbook/ch04/tls* Dockerfile. Check it out.

By setting up a few environment variables `DOCKER_HOST` and `DOCKER_TLS_VERIFY` you can simply the TLS connection from the CLI:

```
$ export DOCKER_HOST=tcp://test:2376
$ export DOCKER_TLS_VERIFY=1
$ docker images
REPOSITORY          TAG             IMAGE ID        CREATED         VIRTUAL SIZE
runseb/dockertls    latest          5ed60e0f6a7c    19 minutes ago  214.7 MB
```

You can still use `curl` as discussed in Recipe 4.8 but you will need to specify the client key and certificate:

```
$ curl --insecure --cert ~/.docker/cert.pem --key ~/.docker/key.pem \
       -s https://test:2376/images/json | python -m json.tool
[
    {
        "Created": 1419280147,
        "Id": "5ed60e0f6a7ce3df3614d20dcadf2e4d43f4054da64d52709c1559ac",
        "ParentId": "138f848eb669500df577ca5b7354cef5e65b3c728b0c241221c611b1",
        "RepoTags": [
            "runseb/dockertls:latest"
        ],
        "Size": 0,
        "VirtualSize": 214723529
    }
]
```

Note that above we used the `--insecure` curl option, because we created our own Certificate Authority. By default `curl` will check the certificates against the CAs contained in the default CA bundle installed on your server. If you were to get server and client keys and certificates from a trusted CA listed in the default CA bundle you would not have to make a `--insecure+` connection. However this does not mean that the connection is not properly using TLS.

4.11 Using `docker-py` to Access the Docker Daemon Remotely

Problem

While the Docker client is very powerfull, you would like to access the Docker daemon through a Python client. Specifically you would like to write a Python program that calls the Docker remote API.

Solution

Import the `docker-py` Python module from Pip. In a Python script of interactive shell, create a connection to a remote Docker daemon and start making API calls.

While this recipe is about `docker-py`, it serves as an example that you can use your own client to communicate with the Docker daemon and you are not restricted to the default Docker client. There exists Docker clients in several programming languages (e.g Java, Groovy, Perl, PHP, Scala, Erlang etc) and you can write your own by studying the API reference.

`docker-py` is a Python client for Docker. It can be installed from source or simply fetched from the Python Package Index using the `pip` command. First install `python-pip`, then get the `docker-py` package. On Ubuntu 14.04:

```
$ sudo apt-get install python-pip
$ sudo pip install docker-py
```

The documentation tells you how to create aconnection to the Docker daemon. Simply create an instance of the `Client()` class, by passing it a `base_url` argument that specifies how the Docker daemon is listening. If it is listening locally on a unix socket:

```
$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from docker import Client
>>> c=Client(base_url="unix://var/run/docker.sock")
>>> c.containers()
[]
```

If it is listening over *tcp*, as we set it up in Recipe 4.8:

```
$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from docker import Client
>>> c=Client(base_url="tcp://127.0.0.1:2375")
>>> c.containers()
[]
```

You can explore the methods available via `docker-py` by doing a `help(c)` at the Python prompt in the interactive sessions started above.

Discussion

The `docker-py` module has a few basics documented. Of note is the integration with *boot2docker* (Recipe 1.3) which has a helper function to setup the connection. Since the latest *boot2docker* uses TLS for added security in accessing the Docker daemon, the setup is slightly different that what we presented above. In addition there is currently a bug that is worth mentioning for those who will be interested in testing `docker-py`.

Start *boot2docker*:

```
$ boot2docker start
Waiting for VM and Docker daemon to start...
.............oooo
Started.
Writing /Users/sebgoa/.boot2docker/certs/boot2docker-vm/ca.pem
Writing /Users/sebgoa/.boot2docker/certs/boot2docker-vm/cert.pem
```

```
Writing /Users/sebgoa/.boot2docker/certs/boot2docker-vm/key.pem

To connect the Docker client to the Docker daemon, please set:
    export DOCKER_HOST=tcp://192.168.59.103:2376
    export DOCKER_CERT_PATH=/Users/sebgoa/.boot2docker/certs/boot2docker-vm
    export DOCKER_TLS_VERIFY=1
```

It returns a set of environment variables that need to be set. Boot2docker provides a nice convenience utility `$(boot2docker shellinit)` to set everything up. However for `docker-py` to work we need to edit our `/etc/hosts` file and set a different `DOCKER_HOST`. In `/etc/hosts` add a line with the IP of *boot2docker* and its local DNS name (i.e *boot2docker*) then `export DOCKER_HOST=tcp://boot2docker:2376`. Then in a Python interactive shell:

```
>>> from docker.client import Client
>>> from docker.utils import kwargs_from_env
>>> client = Client(**kwargs_from_env())
>>> client.containers()
[]
```

4.12 Using `docker-py` Securely

Problem

You want to use the `docker-py` Python client to access a remote Docker daemon setup with TLS secure access.

Solution

After setting up a Docker host as explained in Recipe 4.10, verify that you can connect to the Docker daemon with TLS. For example assuming a host with hostname *dockerpytls* and client certificate, key and CA located in the default location at `~/.docker/` try:

```
$ docker -H tcp://dockerpytls:2376 --tlsverify ps
CONTAINER ID    IMAGE       COMMAND     CREATED     STATUS      PORTS       NAMES
```

Make sure you have installed `docker_py`:

```
sudo apt-get -y install python-pip
sudo pip install docker-py
```

Once this is successfull, open a Python interactive shell and create a `docker-py` client instance using the following configuration:

```
tls_config = docker.tls.TLSConfig(
  client_cert=('/home/vagrant/.docker./cert.pem', '/home/vagrant/.docker/key.pem'),
```

```
  ca_cert='/home/vagrant/.docker/ca.pem'
)
client = docker.Client(base_url='https://host:2376', tls=tls_config)
```

Which is equivalent to calling the Docker daemon on the command line with:

```
$ docker -H tcp://host:2376 --tlsverify --tlscert /path/to/client-cert.pem \
                                        --tlskey /path/to/client-key.pem \
                                        --tlscacert /path/to/ca.pem ...
```

Recipe not yet finished

Discussion

See Also

- Documentation of `docker=py`
- Docker article on HTTPS support

CHAPTER 5
Kubernetes

This chapter consists of recipes focused on Kubernetes. You can send me suggestions at *how2dock@gmail.com*

As applications grow beyond what can be safely handed on a single host, a need for what has come to be called an *orchestration system*. An orchestration systems helps users view a set of hosts (also referred to as nodes) as a unified programmable reliable cluster. That cluster can be viewed and used as a giant computer.

Kubernetes (often abbreviated as *k8s*) is an open source system started by Google to fill this need. Kubernetes is based on ideas validated through internal Google systems over the last ten years (Borg and Omega). These systems are used to run and manage all of the myriad Google services including Google Search, Google Mail and more. Many of the engineers that built and operated Borg clusters at scale are helping to design and build Kubernetes.

Traditinally, Borg has been one of the key things that ex-Google engineers miss. But now Kubernetes fills this need for engineers that don't happen to work for Google.

Enhanced Capabilties

A Kubernetes cluster coordinates Docker across multiple nodes and provides a unified programming model with enhanced capabilities.

Reliable Container Restart
: Kubernetes can monitor the health of a container and restart it when it fails.

Self Healing
: If a node fails, the Kubernetes management system can automatically reschedule work onto healthy nodes. Dynamic service membership ensures that those new containers can be found and used.

High Cluster Utilization
: By scheduling a diverse set of workloads on a common set of machines users can drive dramatically higher utilization compared to static manual placement. The larger the cluster and the more diverse the workloads, the better the utilization.

Organzation and Grouping
: With large clusters it can be difficult to keep track of all of the containers that are running. Kubernetes provides a flexible labeling system that allows both users and other systems to think in sets of containers. In addition, Kubernetes supports the idea of namespaces so different users or teams can have isolated views into the cluster.

Horizontal Scaling and Replication
: Kubernetes is built to enable easy horizontal scaling. Scaling and load balancing are intrinsic concepts.

Microservice friendly
: Kubernetes clusters are a perfect companion for teams adopting a microservices architecture. Applications can be broken down into smaller parts that are easier to develop, scale and reason about. Kubernetes provides building ways for a service to find (commonly called discovery) and communicate with other services.

Streamlined Operations
: Kubernetes allows for specialized ops team. Management of the Kubernetes system and the nodes that it runs on can be driven by a dedicated team or outsourced to a cloud service. Operational teams for specific apps (or the development team itself) can then focus on running that application without managing the details of individual nodes.

New Concepts

While Docker is great for dealing with containers running on a single node, Kubernetes also has to contend with challenges around cross-node communication and scale. To help, Kubernetes introduces a set of new concepts:

Cluster Scheduling
: The process of placing a container on a specific node to optimize the reliability and utilization of the cluster.

Pods
: A group of containers that *must* be placed on a single node and work together as a team. Allowing a set of containers to work closely together on a single node is a powerful way to make applications even more manageable.

Labels
: Data attached to pods in order to organize a group for monitoring and management.

Replication Controllers
: Agents that work to make sure that a horizontal scaling group or Pods is reliably maintained.

Network Services
: A way to communicate between not just pods, but groups of pods using dynamically configured naming and network proxies.

With that let's jump into understanding and using Kubernetes!

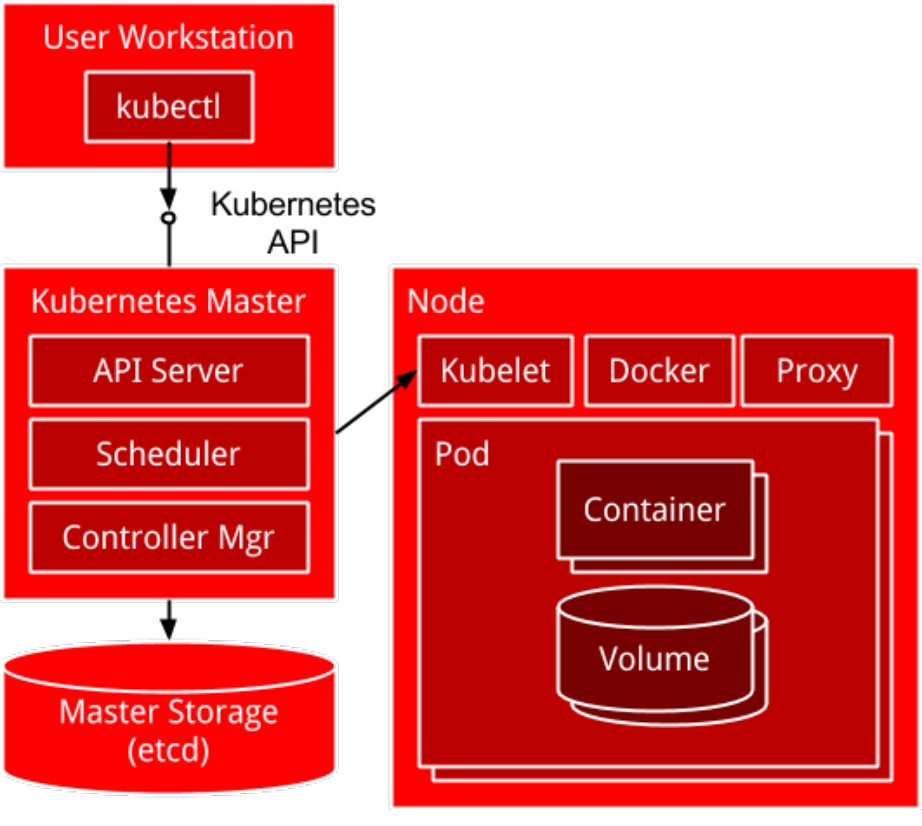
5.1 Understanding Kubernetes Architecture

Contributed by Joe Beda

Problem

You need a container management system that provides scale and fault-tolerance, you would like to understand the architecture of Kubernetes.

Solution



The main architecture parts of a Kubernetes cluster include:

Kubernetes Master Services
: These centralized services (that can run in Docker Containers) provide an API, collect and surface the current state of the cluster and assign pods to nodes. Most users will only ever interact directly with the Master API. This provides a unified view of the entire cluster.

Master Storage
: Currently all persistent Kubernetes state is stored in etcd. It is likely that new storage engines will be built out over time.

Kubelet
: The Kubelet is an agent that runs on every node and is responsible for driving Docker, reporting status to the Master and setting up node level resources (like remote disk storage).

Kubernetes Proxy
: This proxy runs on every node (and can run elsewhere) and provides local containers a single network endpoint to reach an array of pods.

Discussion

A user interacts with a Kubernetes master through tools (like kubectl) that call the Kubernetes API. API documentation (automatically generated from source) is available on the Kubernetes site: *http://kubernetes.io/third_party/swagger-ui/*. The master is responsible for storing user requests (modeled as a specification). It then works to turn that specification into reality. It reports the current state of the cluster as status.

Running on every worker node in the cluster are the kubelet and the proxy. The kubelet is responsible for driving Docker and setting up other node specific state, like storage volumes. The proxy is responsible for providing a stable local endpoint for talking to services (frequently implemented by a set of containers running in cluster).

Kubernetes works to manage Pods. Pods are a grouping of compute resources that provide context for a set of containers. Users can use Pods to force a set of containers that work as a team to be scheduled on a single physical node.

- **Multiple Docker containers** can exist in a Pod. This allows for some advanced scenarios explored in XXX. Each container starts with its file system and process as normal.
- Pods define a **shared network interface**. Unlike regular containers, containers in a Pod all share the same network interface. This allows for efficient and easy access across containers using localhost. It also means that different containers in the same Pod cannot use the same network port.
- **Storage volumes** are defined as part of the Pod. These volumes can be mapped into multiple containers as needed. There also exist specialized types of volumes based on the needs of users and the capabilities of the cluster. See XXX for more details on storage volumes.

The general flow for how work is run with Kubernetes:

1. Via the kubectl tool and the Kubernetes API, the user creates a specification for a Replication Controller with a Pod Template and a count for the number of desired replicas
2. The Kubernetes uses the template in the Replication Controller to create a number of actual pods.
3. The Kubernetes Scheduler (part of the master) looks at the current state of the cluster (which nodes are available and what resources are available on those nodes) and binds a pod to a specific node.

4. The Kubelet on that node watches for a change in the set of pods assigned to the node it is running on. It then starts up or kills pods as necessary. This includes configuring any storage volumes as necessary, downloading the Docker image to that specific node and calling the Docker API to start/stop individual containers.

Fault tolerance is implemented at multiple levels. Individual containers within a Pod can be health checked and monitored by the local Kubelet. If they stop of fail, they can be restarted automatically. If the entire node fails, the master will notice this and update its state to reflect this. At that point Replication Controllers (if used) will create replacements for pods that were on that Node. Multiple levels of monitoring and restarting help to keep applications running even when the cluster is experience problems (software or hardware).

Pods Only Get Scheduled Once

Once a pod is scheduled on a node it will never be moved. If that node is lost or removed from the cluster the pod will not be restarted. This is surprising behavior given that a goal of Kubernetes is reliably keep work running. This is required as networks are imperfect. In the event that the master cannot talk a node, any pod on that node is in an indeterminate state as far as the master is concerned — it may or may not be running. If that same pod were restarted on another machine, there may be 2 pods with the exact same name/idenity running at the same time. This can cause all sorts of problems. For instance, distributed logs might be written from multiple places all keyed to the same pod id. Or the pod id may be used as part of a master election system and clients may be confused as to which pod is really the master.

Instead, to reliably run a workload, it is necessary to use a Replication Controller. This takes a pod template and tries to ensure that there is always a specific number of pods running to accomplish that task. In the case of the master not being able to talk to a node, a Replication Controller is in charge of spinning up a new pod to replace the lost pods. If communication is reestablished, it is up to the Replication Controller to delete one of the redundant pods.

5.2 Networking Pods for Container Connectivity

Contributed by Joe Beda

Problem

You want to control how network traffic gets directed to your containers as they are scheduled across a Kubernetes cluster.

Solution

The solution is to use Kubernetes Services. These can be used to communicate between containers within a cluster or to direct external traffic to a set of pods.

TODO

We need an example here of bringing up a set of pods using a replication controller and then configuring a service to point at it. We should also show how to configure an external IP

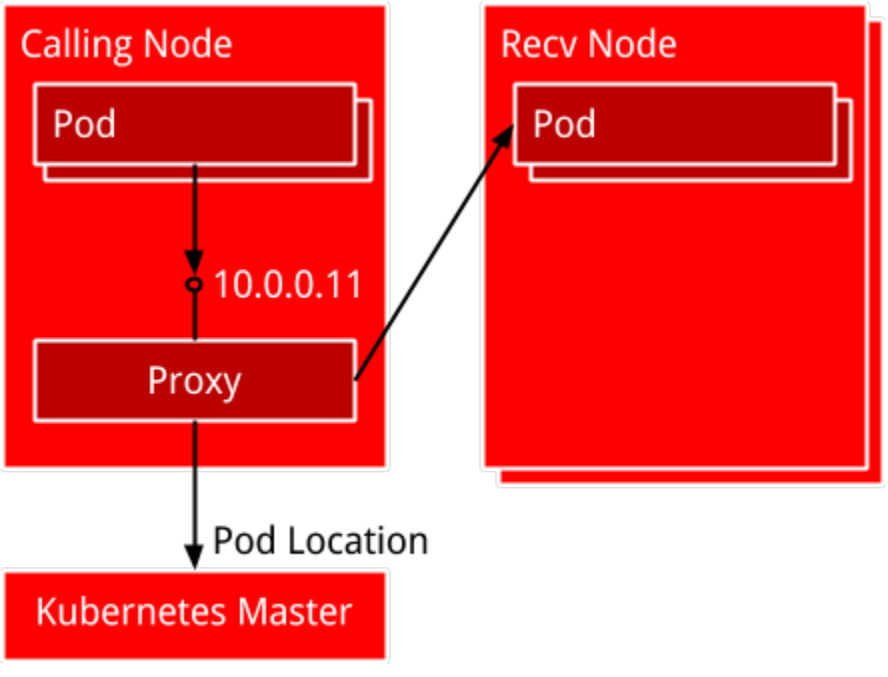
Discussion

Networking Assumption: IP per Pod

Kubernetes assumes a network model where each Pod gets an IP. Each pod can then connect to other pods regardless of which physical node they happen to be running on. The easiest way to get this running on most configuration is to use Flannel from CoreOS. In certain other environments (e.g. Google Compute Engine with advanced routing) communication can be handled directly by the network infrastructure.

However, just because pods can connect directly doesn't mean that is the best or easiest way to communicate between pods. In the event that a pod fails or is replaced on to a new node, the calling code would have to know to reconnect to a new address. This dynamic reconnection is hard to integrate with many existing servers and frameworks.

Kubernetes Services make it easy to connect between Pods. Creating a Kubernetes Service will allocate a new IP address for the Service that is independent of any specific Pod. When a calling Pod then establishes a connection to that Service, it will be handled by the local Kubernetes Proxy that is running on that node. This Proxy will forward the connection on a Pod that is implementing that service. In the case where there are multiple Pods backing a Service, the proxy will load balance across those Pods.



The calling code can find the IP for a service in two ways: environment variables and DNS. The environment variables created for Services are similar to Docker link variables. For example, suppose you have a service called `redis` that exposes port `6379`.

```
REDIS_MASTER_SERVICE_HOST=10.0.0.11
REDIS_MASTER_SERVICE_PORT=6379
REDIS_MASTER_PORT=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP_PROTO=tcp
REDIS_MASTER_PORT_6379_TCP_PORT=6379
REDIS_MASTER_PORT_6379_TCP_ADDR=10.0.0.11
```

When DNS support is configured on a Kubernetes cluster, each Service will also be given a resolvable name. In this example, assuming the default namespace of `namespace` and a DNS domain root of `cluster.local`, the service will be exposed as `redis.default.cluster.local`.

TODO

Joe's Outline:

- Overview of Kubernetes networking with IP-per-node
- Discuss proxy and how it is used to spray traffic across a set of pods. Diagram will help here. Each service gets an IP and that can be optionally surfaced into DNS.
- Discuss how to connect from outside the cluster by configuring the proxy to listen for and handle other IPs. Also mention that for specific clouds an external IP can be provisioned automatically.

See Also

- Documentation on Kubernetes Services

5.3 Using Labels for Container Placement and Application Management

Contributed by Joe Beda

Problem

Solution

Discussion

5.4 Creating a Multi-node Kubernetes Cluster With Vagrant

Problem

You want to get started with Kubernetes and would like to create a small cluster on your local machine using Vagrant.

Solution

You will need to install Vagrant and VirtualBox if you have not done so already. Then set two environment variables, `KUBERNETES_PROVIDER` to specify that you will use Vagrant and `NUM_MINIONS` to set the number of nodes in your cluster (in addition to

the master node). Then you will use the installation script provided by the Kubernetes community. It will read the environment variables, detect your operating system, download the latest stable release of Kubernetes and untar it in a `kubernetes` directory. The following commands show you these steps on the command line:

```
export KUBERNETES_PROVIDER=vagrant
export NUM_MINIONS=2
curl -sS https://get.k8s.io | bash
```

If you do not specify the `NUM_MINIONS` environment variables, only one node will be started in addition to the master node.

Each virtual machine started with Vagrant will use 1GB of RAM, so make sure you have enough memory.

Downloading the Vagrant box being used, and provisioning the virtual machines using Saltstack will take a bit of time. Once it is done, the nodes will got through a validation step and you should see a similar output on stdout:

```
...
Using credentials: vagrant:vagrant
KUBE_MASTER_IP: 10.245.1.2
Minions already detected
current-context: "vagrant"
...
Found 2 nodes.
     1  10.245.1.3
     2  10.245.1.4
Attempt 1 at checking Kubelet installation on node 10.245.1.3 ... [working]
Attempt 1 at checking Kubelet installation on node 10.245.1.4 ... [working]
Cluster validation succeeded
...
```

The `vagrant status` command will list your running VMs:

```
$ vagrant status
Current machine states:

master                   running (virtualbox)
minion-1                 running (virtualbox)
minion-2                 running (virtualbox)
```

At this point you have a working Kubernetes cluster running locally within virtual machines.

Discussion

The Vagrant box used to create this cluster are based on Fedora 20 and use `systemd`. If you connect to these VMs you can list the `systemd` units that are running and make up the Kubernetes system.

On the master node, we find four services running. The *Addon* object manager, the *API* server, the *Controller* manager and the *Scheduler*. Docker is also running.

```
$ vagrant ssh master
Last login: Tue Feb 24 20:08:45 2015 from 10.0.2.2
[vagrant@kubernetes-master ~]$ sudo systemctl list-units | grep kube
kube-addons.service              loaded active running   Kubernetes Addon Object Manager
kube-apiserver.service           loaded active running   Kubernetes API Server
kube-controller-manager.service  loaded active running   Kubernetes Controller Manager
kube-scheduler.service           loaded active running   Kubernetes Scheduler Plugin
```

On the minions, we find two more Kubernetes related services. The *Kube-Proxy* server and the *Kubelet* server. Docker is of course also running.

```
$ vagrant ssh minion-1
Last login: Tue Feb 24 20:08:45 2015 from 10.0.2.2
[vagrant@kubernetes-minion-1 ~]$ sudo systemctl list-units | grep kube
kube-proxy.service             loaded active running   Kubernetes Kube-Proxy Server
kubelet.service                loaded active running   Kubernetes Kubelet Server
```

To interact with the cluster you can use the `kubectl.sh` script on your localhost. This script allows you to manage all Kubernetes resources that make up container scheduling tasks. Here is a snippet of the `kubectl` help:

```
$ ./cluster/kubectl.sh
...
Usage:
  kubectl [flags]
  kubectl [command]

Available Commands:
  get           Display one or many resources
  describe      Show details of a specific resource
  create        Create a resource by filename or stdin
  update        Update a resource by filename or stdin.
  delete        Delete a resource by filename, stdin, or resource and ID.
  namespace     SUPERCEDED: Set and view the current Kubernetes namespace
  log           Print the logs for a container in a pod.
  rollingupdate Perform a rolling update of the given ReplicationController.
  resize        Set a new size for a Replication Controller.
  exec          Execute a command in a container.
  port-forward  Forward one or more local ports to a pod.
  proxy         Run a proxy to the Kubernetes API server
  run-container Run a particular image on the cluster.
  stop          Gracefully shut down a resource by id or filename.
  expose        Take a replicated application and expose it as Kubernetes Service
```

```
  label         Update the labels on a resource
  config        config modifies .kubeconfig files
  clusterinfo   Display cluster info
  apiversions   Print available API versions.
  version       Print the client and server version information.
  help          Help about any command
...
```

To test that you can indeed communicate to the Kubernetes API server running on the master node, try to list the nodes in the cluster:

```
$ ./cluster/kubectl.sh get nodes
NAME         LABELS    STATUS
10.245.1.3   <none>    Ready
10.245.1.4   <none>    Ready
```

To destroy all the virtual machines run the `./cluster/kube-down.sh` script

You are now ready to head over to Recipe 5.5 and create your first containers using Kubernetes.

See Also

- Documentation on the Vagrant provisioning
- Bash script that automatest the creation of a Kubernetes cluster using the latest stable release.

5.5 Starting Containers on a Kubernetes Cluster with Pods

Problem

You know how to start containers using the Docker command line interface, now you would like to use Kubernetes to schedule your containers in a cluster.

Solution

You have a Kubernetes cluster available to you, either through Recipe 5.4 or Recipe 5.11 or a public cloud provider like Google Container Engine. In addition you have downloaded the Kubernetes client `kubectl` and it is setup to use your cluster endpoint with the appropriate authentication (see Recipe 5.17).

As explained in Recipe 5.1, containers get scheduled as a group by defining pods. Therefore to start your first container you need to write a Pod definition in `json` or `yaml` and use the `kubectl` client to submit it to the Kubernetes API server.

Let's start with a fun example and run the 2048 game. A Docker image is available on the Docker hub and I will leave it to you to check out the Dockerfile. Save the YAML file shown below as `2048.yaml`.

```
apiVersion: v1beta3
kind: Pod
metadata:
  name: podname
spec:
  containers:
  - image: cpk1224/docker-2048
    name: imagename
    ports:
    - containerPort: 80
      hostPort: 80
```

You can now submit it to your cluster with:

```
$ kubectl create -f 2048.yaml
pods/podname
```

Once the image is downloaded the container will start running, you should be able to use your browser and open the 2048 game on the IP of the host that is running it. You will need to open any firewall rules that may prevent you to do so.

Discussion

The YAML file specifies the API version (i.e `v1beta3`) and the kind of object it defines (i.e Pod). Then some metadata needs to be set to specify a name for this Pod. In this example, a single container is started but there could be several. All would be defined in the `spec` section under the `container` field. The image used and a name for the container are required parameters. In this example we also define port 80 to be exposed and mapped on port 80 of the host (using the `containerPort` and `hostPort` keys).

You can then list the Pods that you have running with `kubectl get pods`. You will see that the Pod will enter running state, that there is one container in that Pod, what the image is, and its status.

```
$ kubectl get pods
POD       IP           CONTAINER(S)   IMAGE(S)              HOST               LABELS    STATUS

podname   10.132.1.9                                        k8s-node/1.2.3.4   <none>    Running
                       imagename      cpk1224/docker-2048                                    Running
```

To learn the API specification you can query the Pod and return its definition in YAML or JSON with:

```
$ ./kubectl get pods -o yaml podname
apiVersion: v1beta3
kind: Pod
metadata:
  creationTimestamp: 2015-05-18T15:10:47Z
  name: podnameme
...<snip>
```

Once you are done experimenting you can delete the pod easily:

```
$ kubectl delete pods podname
```

5.6 Taking Advantage of Labels For Querying Kubernetes Objects

Problem

In a large Kubernetes cluster you may run thousands of Pods as well as other cluster objects. You would like to easily query and manipulate sets of objects in multi-dimensional ways using a tagging system.

Solution

Tag your objects (e.g Pods) using labels. Labels are key/value pairs that can be attached to any Kubernetes object. These labels are defined primarily in the metadata section of an object definition.

Taking the example from Recipe 5.5 you can add a label `foo=bar` by modifying the Pod `yaml` metadata description like so:

```
apiVersion: v1beta3
kind: Pod
metadata:
  name: podname
  foo: bar
spec:
  containers:
  - image: cpk1224/docker-2048
    name: imagename
    ports:
    - containerPort: 80
      hostPort: 80
```

The end result is that you can now list Pods that have that specific label using the `--selector` option of the `kubectl` CLI. Like so:

```
$ kubectl get pods --selector="foo=bar"
```

Additionally, you can add labels at *runtime* using the `kubectl labels` function:

```
$ kubectl label pods podname env=production
POD               IP           CONTAINER(S)   IMAGE(S)   HOST                    LABELS
podnamemetadata   10.132.1.9                               k8s-yoyo-node/1.2.3.4   env=production,fo
```

Labels follow a specific syntax and must not start and end with a number.

In short, labels are a straightforward tagging system which allows users to add metadata to any resource in their cluster. It helps build cross-functional relationships to manage sets of resources in various stages of an application lifecycle.

See Also

- Introduction, motivation and syntax of labels

5.7 Using a Replication Controller to Manage the Number of Replicas of a Pod

Problem

You need to make sure that several replicas of your pod exist at any time in the cluster.

Solution

Kubernetes is a declarative system where users express what they want the system to do and not how to do it. Using replication controllers you can specify the number of replicas that you want for a Pod. This helps with high load and availability by serving part of an application through a service proxy (see Recipe 5.9).

Replication controllers are one of the three key objects in a Kubernetes cluster (with Pods and Services). You can list all running replication controllers with `kubectl`:

```
$ kubectl get replicationcontrollers
...
$ kubectl get rc
```

To create a replication controller, you simply write a `json` or `yaml` file following the replication controller API specification. It can contains metadata, the number of rep-

licas that you want, a selector to target specific Pods and a template for a Pod. Currently the template is embedded within the replication controller definition but this may change in future version of Kubernetes.

For example, if you want to create a replication controller for the 2048 game that we ran in a single Pod in Recipe 5.5, you can write the following `rc2048.yaml` file:

```
apiVersion: v1beta3
kind: ReplicationController
metadata:
  labels:
    name: rcgame
  name: rcgame
spec:
  replicas: 1
  selector:
    name: game
  template:
    metadata:
      labels:
        name: game
    spec:
      containers:
      - image: cpk1224/docker-2048
        name: test
        ports:
        - containerPort: 80
```

The controller itself will have the `rcgame` label, and will target pods with the label `game`. Once started the controller will ensure that one pod is running at all time. You launch it with the `kubectl create` like so:

```
$ kubectl create -f rc2048.yml
replicationcontrollers/rcgame
$ kubectl get rc
CONTROLLER     CONTAINER(S)   IMAGE(S)                  SELECTOR          REPLICAS
rcgame         test           cpk1224/docker-2048       name=game         1
```

Try killing the pod that was created. You will see that a new one automatically starts again.

You do not need to have an existing pod running before starting a replication controller. It will automatically start a pod that matches the label specified in the definition if it does not exist yet. You can also set the number of replicas to zero.

The magic happens when you want to increase the number of replicas. You can use the `kubectl resize` command and the number of pods will automatically be adjusted.

```
$ kubectl resize --replicas=4 rc rcgame
resized
```

Discussion

In Recipe 5.1 we mentioned that every node in a Kubernetes cluster runs a *kubelet*. This process watches over the pods that are scheduled on a node and makes sure they keep running. But what happens if the node dies ? Kubernetes needs to have a way to re-schedule that pod on another node automatically as well as keep a number of replicas up for availability. This is what replication controllers help you achieve.

While replication controllers are extremely helpful for guaranteed availability and elasticity, they are also a great way to perform application deployment scenarios such as canary deployment. In fact, Kubernetes has a built rolling update mechanism based on replication controllers, it is worth investigating:

```
$ ./kubectl rollingupdate -h
Perform a rolling update of the given ReplicationController.

Replaces the specified controller with new controller, updating one pod at a time to use the
new PodTemplate. The new-controller.json must specify the same namespace as the
existing controller and overwrite at least one (common) label in its replicaSelector.
...<snip>
```

See Also

- Documentation on replication controllers.

5.8 Running Multiple Containers in a Pod

Problem

You know how to run a single container in a Pod, but would like to run multiple ones which will be collocated. You might already have some containers in production that use the Docker linking mechanism on a single host and would like to use Kubernetes to do the same.

Solution

A Pod definition is not restricted to a single container. You can define as many containers as you want as well as volumes (see Recipe 5.10). In Recipe 5.5 we wrote a simple pod definition that started a single container. The example below starts wordpress using the official images from Docker hub for wordpress and mysql. Both run as separate containers and use environment variables to configure the installation. The wordpress container defines the `WORDPRESS_DB_HOST` as environment variables

and sets is to 127.0.0.1. This allows wordpress to reach the mysql database also started within the pod. This works since pods get a single IP address in the current Kubernetes networking model (see ???). Create the following `wordpress.yaml` file:

```
apiVersion: v1beta3
kind: Pod
metadata:
  labels:
    name: wp
  name: wp
spec:
  containers:
  - name: wordpress
    env:
    - name: WORDPRESS_DB_NAME
      value: wordpress
    - name: WORDPRESS_DB_USER
      value: wordpress
    - name: WORDPRESS_DB_PASSWORD
      value: wordpresspwd
    - name: WORDPRESS_DB_HOST
      value: 127.0.0.1
    image: wordpress
    ports:
    - containerPort: 80
      hostPort: 80
  - name: mysql
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: wordpressdocker
    - name: MYSQL_DATABASE
      value: wordpress
    - name: MYSQL_USER
      value: wordpress
    - name: MYSQL_PASSWORD
      value: wordpresspwd
    image: mysql
    ports:
    - containerPort: 3306
```

Create the pod with:

```
$ kubectl create -f wordpress.yaml
```

Once the containers start you will have a working wordpress installation.

You can view the logs of the containers in your pod with the `kubectl` client like so:

```
$ kubectl log wp wordpress
```

Where *wp* is the name of the pod you started and *wordpress* the name of the container you want to see the logs from.

Discussion

While starting multiple containers through a pod is straightforward, accessing the application running within a pod requires using Kubernetes services. Each pod gets its own IP address in a private network. To access an application from outside the Kubernetes cluster through a public IP address, you need to create a service which will bind the application to a public IP address or make us of an external load balancer service.

In Google container engine, using an external load balancer in a service definition is done directly in the YAML file describing the service. For instance, to expose the wordpress application that is running through the pod defined above, you need to create a service file `sgoogle.yml` like so:

```
apiVersion: v1beta3
kind: Service
metadata:
  labels:
    name: wordpress
  name: wordpress
spec:
  createExternalLoadBalancer: true
  ports:
    - port: 80
  selector:
    name: wp
```

The service has metadata associated with it, but the important part is the selector filed in the spec section. In the example above the selector *wp* will allow the service to create a proxy that will bind the IP address given by the load balancer to the pod that matches the *wp* label. Once you obtain the IP address of the load balancer you can access it from the public internet. The Kubernetes service will proxy the request to the node where the pod is running. If the pod has been started with a replication controller, the service will also load balance the requests among all the running pods.

On a Cloud provider whose load balancing system is not yet supported by Kubernetes you can bind the pod to a public IP address manually with a service definition like this (where 1.2.3.4 needs to be replaced with the public IP).

```
apiVersion: v1beta3
kind: Service
metadata:
  labels:
    name: wordpress
  name: wordpress
spec:
  publicIPs: ["1.2.3.4"]
  ports:
    - port: 80
```

```
  selector:
    name: wp
```

5.9 Using Service Proxies For Dynamic Linking of Containers

Problem

You want to *link* containers across multiple hosts in your cluster instead of running multiple containers per pod. This is the more cloud native way of designing an application where layers that can scale and can operate separately from each other, run as separate replication controllers.

Solution

In Recipe 5.8 we started wordpress by running the mysql and wordpress container in a single pod. Which meant that the two containers started on the same host. We took advantage of the fact that a pod has a single IP address to set the wordpress mysql host to localhost. However, we could imagine running a replicated Mysql service and/or a replicated wordpress frontend. This would mean that the containers would run on different hosts in the cluster.

Kubernetes services are smart proxies that keep track of changes in pod cluster allocation and update their port mapping dynamically when pods get re-scheduled.

A better way of running our canonical wordpress example would be to run Mysql as a single pod or replication controller (glancing over the issues with database replication and data persistence) and then exposing this Mysql service through a Kubernetes service definition.

The replication controller would look something like this:

```
apiVersion: v1beta3
kind: ReplicationController
metadata:
  labels:
    name: mysql
  name: mysql
spec:
  replicas: 1
  selector:
    name: mysql
  template:
    metadata:
      labels:
        name: mysql
      name: mysql
    spec:
```

```
    containers:
    - name: mysql
      image: mysql
      ports:
      - containerPort: 3306
      env:
      - name: MYSQL_ROOT_PASSWORD
        value: wordpressdocker
      - name: MYSQL_DATABASE
        value: wordpress
      - name: MYSQL_USER
        value: wordpress
      - name: MYSQL_PASSWORD
        value: wordpresspwd
```

A Mysql service can then defined as a different type of Kubernetes object. It can be managed through the API. A service definition for Msyql would be:

```
kind: Service
apiVersion: v1beta3
metadata:
  name: mysql
  labels:
    name: mysql
spec:
  selector:
    name: mysql
  ports:
    - port: 3306
```

You will note the selector field in the *spec* section. This selector will match all pods that contain the *mysql* label. The service will expose Mysql on port 3306 of all the nodes in the cluster.

The wordpress pod, does not need to reference the database host, as by default it will look for it on localhost and port 3306. An endpoint that will be exposed by the service we just created. Hence the wordpress pod looks like this:

```
apiVersion: v1beta3
kind: Pod
metadata:
  labels:
    name: wp
  name: wp
spec:
  containers:
  - env:
    - name: WORDPRESS_DB_NAME
      value: wordpress
    - name: WORDPRESS_DB_USER
      value: wordpress
    - name: WORDPRESS_DB_PASSWORD
      value: wordpresspwd
```

```
    image: wordpress
    name: wordpress
    ports:
    - containerPort: 80
      hostPort: 80
      protocol: TCP
```

Wordpress is exposed to the public internet the same way that we did in Recipe 5.8, through another service.

Discussion

With pods and replication controllers, services are the three key entities of a Kubernetes. Services bring a locality abstraction on top of pods which is key to self-discovery and a dynamic behavior in a large scale cluster with failures happens. With services, everything appears local and everything can move within the cluster without loosing availability and without requiring any restart.

Namespaces are also key for handling multi-tenancy, but are still a work in progress in the current version of Kubernetes.

Better introduce services.. Do diagram…

See Also

- Wordpress example in Kubernetes documentation.

5.10 Defining Volumes in Pods

Problem

TODO

Solution

Discussion

5.11 Creating a Single Node Kubernetes Cluster Using Docker Compose

Problem

You know how to create a Kubernetes cluster by running the various cluster components (e.g API server, Scheduler, Kubelet) as `systemd` units. But why not taking advantage of Docker itself to run these components. It would simplify deployment of the cluster. To test this deployment scenario you want to try to run a one node Kubernetes cluster locally using only Docker containers.

Solution

The Kubernetes documentation has a good resource about this scenario. In this recipe we will go one step further and take advantage of Docker compose (See Recipe 7.1). To get started you will need a Docker host and Docker compose installed. You can clone the repository that comes with this book and use the Vagrantfile provided like so:

```
$ git clone https://github.com/how2dock/docbook.git
$ cd docbook/ch05/docker
$ tree
.
├── Vagrantfile
├── k8s.yml
└── kubectl
```

The Vagrantfile contains a small bootstrap script that will install Docker in the virtual machine as well as Docker compose. The `k8s.yml` is the *compose* definition to start all the components of Kubernetes as containers. Bring up the machine and run *compose*, all the required images will be downloaded and the containers will start.

```
$ vagrant up
$ vagrant ssh
$ cd /vagrant
$ docker-compose -f k8s.yml up -d
$ docker ps
CONTAINER ID        IMAGE                                          COMMAND
64e0073615c5        gcr.io/google_containers/hyperkube:v0.14.1   "/hyperkube controll  ...
9603f3b5b186        gcr.io/google_containers/hyperkube:v0.14.1   "/hyperkube schedule  ...
3ce44e77989f        gcr.io/google_containers/hyperkube:v0.14.1   "/hyperkube apiserve  ...
1b0bcbb56d59        kubernetes/pause:go                            "/pause"              ...
0b0c3e2735a9        kubernetes/etcd:2.0.5.1                        "/usr/local/bin/etcd  ...
```

```
459c45ef9389        gcr.io/google_containers/hyperkube:v0.14.1   "/hyperkube proxy --  ...
005c5ac1de0e        gcr.io/google_containers/hyperkube:v0.14.1   "/hyperkube kubelet   ...
```

That is it. You now have a one node Kubernetes *cluster* will all components running as containers, the `get nodes` returns your localhost and you can create pods, replication controllers and services.

```
$ ./kubectl get nodes
NAME        LABELS    STATUS
127.0.0.1   <none>    Ready
```

To test that you can create a new pod, we are going to run a single `nginx` container.

```
$ ./kubectl run-container nginx --image=nginx --port=80
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR              REPLICAS
nginx        nginx          nginx      run-container=nginx   1
```

The `run-container` command automatically created a replication controller for this container. List it with `./kubectl get rc`

To access this `nginx` frontend from outside the cluster, we need to expose it as a service. However, when creating the service we pass the host-only network IP of the virtual machine to the `kubectl` command. Otherwise the service will be created, all future pods will be able to access it but we will not be able to reach it from outside the cluster.

```
$ ./kubectl expose rc nginx --port=80 --public-ip=192.168.33.10
NAME      LABELS    SELECTOR              IP          PORT
nginx     <none>    run-container=nginx   10.0.0.98   80
```

Once the `nginx` image is downloaded, the pod will enter running state and you will be able to access the `nginx` welcome page at *`http://192.168.33.10`*

```
$ ./kubectl get pods
POD           IP           CONTAINER(S)         IMAGE(S)                                        ...
nginx-127                  controller-manager   gcr.io/google_containers/hyperkube:v0.14.1      ...
                           apiserver            gcr.io/google_containers/hyperkube:v0.14.1
                           scheduler            gcr.io/google_containers/hyperkube:v0.14.1
nginx-461yi   172.17.0.6   nginx                nginx                                           ...
```

Discussion

The `k8s.yml` *compose* file shows us how this was done:

```
etcd:
  image: kubernetes/etcd:2.0.5.1
  net: "host"
  command: /usr/local/bin/etcd --addr=127.0.0.1:4001 --bind-addr=0.0.0.0:4001 --data-dir=/var/etcd
```

```
master:
  image: gcr.io/google_containers/hyperkube:v0.14.1
  net: "host"
  volumes:
    - /var/run/docker.sock:/var/run/docker.sock
  command: /hyperkube kubelet --api_servers=http://localhost:8080 --v=2 --address=0.0.0.0 \
           --enable_server --hostname_override=127.0.0.1 --config=/etc/kubernetes/manifests
proxy:
  image: gcr.io/google_containers/hyperkube:v0.14.1
  net: "host"
  privileged: true
  command: /hyperkube proxy --master=http://127.0.0.1:8080 --v=2
```

Three containers got started by *compose*. One container running `etcd`, one container running the Kubernetes proxy service and one container running the Kubernetes kubelet. Both the service proxy and the kubelet are running from the same image and using the same binary that is called through the command option. This binary is `hyperkube` a very nice utility binary that you can use to start all the components of a Kubernetes cluster.

The very clever part is that the *master* container calls `hyperkube` by specifying a configuration file in `/etc/kubernetes/manifests` located within the container image. We can check what is in this manifest by running a new ephemeral container:

```
$ docker run --rm -it gcr.io/google_containers/hyperkube:v0.14.1 cat /etc/kubernetes/manifests/mas
{
"apiVersion": "v1beta3",
"kind": "Pod",
"metadata": {"name":"nginx"},
"spec":{
  "hostNetwork": true,
  "containers":[
    {
      "name": "controller-manager",
      "image": "gcr.io/google_containers/hyperkube:v0.14.1",
      "command": [
              "/hyperkube",
              "controller-manager",
              "--master=127.0.0.1:8080",
              "--machines=127.0.0.1",
              "--sync_nodes=true",
              "--v=2"
        ]
    },
    {
      "name": "apiserver",
      "image": "gcr.io/google_containers/hyperkube:v0.14.1",
      "command": [
              "/hyperkube",
              "apiserver",
              "--portal_net=10.0.0.1/24",
```

```
                "--address=127.0.0.1",
                "--etcd_servers=http://127.0.0.1:4001",
                "--cluster_name=kubernetes",
                "--v=2"
          ]
      },
      {
        "name": "scheduler",
        "image": "gcr.io/google_containers/hyperkube:v0.14.1",
        "command": [
                "/hyperkube",
                "scheduler",
                "--master=127.0.0.1:8080",
                "--v=2"
          ]
      }
    ]
  }
}
```

This manifest is given to the `kubelet` which starts the containers defined. In this case, it starts the API server, the scheduler and the controller manager of Kubernetes. These three components form an actual Kubernetes pod themselves and will be watched over by the `kubelet`. Indeed if we list the running pods we get:

```
$ ./kubectl get pods
POD         IP        CONTAINER(S)         IMAGE(S)                                          ...
nginx-127             controller-manager   gcr.io/google_containers/hyperkube:v0.14.1   ...
                      apiserver            gcr.io/google_containers/hyperkube:v0.14.1
                      scheduler            gcr.io/google_containers/hyperkube:v0.14.1
```

See Also

- Running Kubernetes locally via Docker

5.12 Compiling Kubernetes to Create Your Own Release

Problem

You want to build the Kubernetes binaries from source instead of downloading the released binaries.

Solution

Kubernetes is written in Go, the build system uses Docker and builds everything in containers. You can build Kubernetes without using containers and using your local Go environment, however using containers greatly simplifies the setup. Therefore to

build the Kubernetes binaries you will need to install the Go language packages, Docker and Git to get the source code from GitHub. For instance on a Ubuntu 14.04 system:

```
$ sudo apt-get update
$ sudo apt-get -y install golang
$ sudo apt-get -y install git
$ sudo curl -sSL https://get.docker.com/ubuntu/ | sudo sh
```

Verify that you have Go and Docker installed:

```
$ go version
go version go1.2.1 linux/amd64
$ docker version
Client version: 1.6.1
Client API version: 1.18
Go version (client): go1.4.2
Git commit (client): 97cd073
OS/Arch (client): linux/amd64
Server version: 1.6.1
Server API version: 1.18
Go version (server): go1.4.2
Git commit (server): 97cd073
OS/Arch (server): linux/amd64
```

Clone the Kubernetes Git repo to get the Go source code:

```
$ git clone https://github.com/GoogleCloudPlatform/kubernetes.git
$ cd kubernetes
```

You are now ready to build the binaries. A build script `run.sh` is provided in the `/build` directory, just use it. It will ask you if you want to download the Docker image for golang, then start the build. Here is a snippet of a build run:

```
$ ./build/run.sh hack/build-go.sh
+++ [0513 11:51:46] Verifying Prerequisites....
You don't have a local copy of the golang docker image. This image is 450MB.
Download it now? [y/n] Y
...<snip>
+++ [0513 11:58:08] Placing binaries
+++ [0513 11:58:14] Running build command....
+++ [0513 11:58:16] Output directory is local.  No need to copy results out.
```

The binaries will be in the `_output` directory. If you built on a linux 64-bit host they will be in `_output/dockerized/bin/linux/amd64`:

```
~/kubernetes/_output/dockerized/bin/linux/amd64# tree
.
├── e2e
├── genbashcomp
├── gendocs
├── genman
├── ginkgo
```

```
├── hyperkube
├── integration
├── kube-apiserver
├── kube-controller-manager
├── kubectl
├── kubelet
├── kube-proxy
├── kubernetes
├── kube-scheduler
└── web-server
```

Discussion

Similarly you can also build the complete set of release artifacts. They will be delivered as tarballs with `kubernetes.tar.gz` containing all binaries, examples, add-ons and deployment scripts. Creating the release will take more time than simply building the binaries, all end to end tests will run. To build a full release do the following and check that the `/_output/release-tars/` directory contains all the tarballs:

```
$ ./build/release.sh
$ tree _output/release-tars/
_output/release-tars/
├── kubernetes-client-darwin-386.tar.gz
├── kubernetes-client-darwin-amd64.tar.gz
├── kubernetes-client-linux-386.tar.gz
├── kubernetes-client-linux-amd64.tar
├── kubernetes-client-linux-arm.tar.gz
├── kubernetes-client-windows-amd64.tar.gz
├── kubernetes-salt.tar.gz
├── kubernetes-server-linux-amd64.tar.gz
├── kubernetes.tar.gz
└── kubernetes-test.tar.gz
```

In addition to the tarballs, the release process will also create three Docker images for the three main components of a Kubernetes cluster: the API server, the controller and the scheduler.

```
# docker images
REPOSITORY                                          ...
gcr.io/google_containers/kube-controller-manager    ...
gcr.io/google_containers/kube-scheduler             ...
gcr.io/google_containers/kube-apiserver             ...
```

The release contains a Dockerfile that builds an image containing the `hyperkube` binary. This binary can be used to start all the components of a Kubernetes cluster. This is what was used in Recipe 5.11 to run Kubernetes in a single node using Docker containers. You can use this Dockerfile to build your own Hyperkube image and edit the configuration file `master.json` to your liking.

```
$ tree kubernetes/cluster/images/hyperkube/
kubernetes/cluster/images/hyperkube/
├── Dockerfile
├── Makefile
├── master.json
└── master-multi.json
```

See Also

- Building Kubernetes README
- Development environment using godep

5.13 Starting Kubernetes Components with `hyperkube` Binary

Problem

A kubernetes cluster is made of a master node and several worker nodes. Each run several Kubernetes binaries. To ease deployment you would like to use a single binary passing the type of component you want to start as an option to this binary.

Solution

Use `hyperkube`.

As suggested in a tip at the end of the Recipe 5.12 recipe, a release contains all Kubernetes components binaries. The API server, the controller manager, the scheduler, the service proxy and the kubelet. The last two run on each worker node while the first three make up the Kubernetes master together with etcd. `hyperkube` is a single binary that allows you to start all these components.

Assuming you created your own release as shown in Recipe 5.12 you will find hyperkube in the `_output/` directory:

```
# tree ~/kubernetes/_output/release-tars/kubernetes/server/kubernetes/server/bin
/root/kubernetes/_output/release-tars/kubernetes/server/kubernetes/server/bin
├── hyperkube
├── kube-apiserver
```

```
├── kube-apiserver.docker_tag
├── kube-apiserver.tar
├── kube-controller-manager
├── kube-controller-manager.docker_tag
├── kube-controller-manager.tar
├── kubectl
├── kubelet
├── kube-proxy
├── kubernetes
├── kube-scheduler
├── kube-scheduler.docker_tag
└── kube-scheduler.tar
```

To use `hyperkube` you need to specify which component you want to start (i.e `api server`, `controller-manager`, `scheduler`, `kubelet` or `proxy`). Once you specify a component, you can pass all the options that you choose. For example to start the API server, check the `hyperkube` usage:

```
$ ./hyperkube apiserver -h
The main API entrypoint and interface to the storage system. The API server is
also the focal point for all authorization decisions.

Usage:
  apiserver [flags]

Available Flags:
      --address=127.0.0.1: DEPRECATED: see --insecure-bind-address instead
      --admission-control="AlwaysAdmit": Ordered list of plug-ins to do admission control ...
      --admission-control-config-file="": File with admission control configuration.
      --allow-privileged=false: If true, allow privileged containers.
<snip>
```

Discussion

5.14 Exploring the Kubernetes API

Problem

Kubernetes exposes a REST API which you need to learn to be able to manage your Kubernetes cluster and run applications in it.

Solution

Kubernetes is currently under development and the API is not stable yet. Several versions of the API are being served by the API server until the v1 API is stabilized and released (expected to be in June 2015). However you can get started with learning the API using a local Kubernetes environment and some simple `curl` commands.

Start a local Kubernetes cluster using Docker as shown in Recipe 5.11. This is the easiest way to try it out. Once all components are running you can reach the API served by the API server. If you are on the machine running the API server you can reach it at *http://localhost:8080* without any authentication. Using `curl` gives you your first Kubernetes raw API experience. Try listing all the API versions available by calling the *http://localhost:8080/api* route like so:

```
$ curl http://localhost:8080/api
{
  "versions": [
    "v1beta1",
    "v1beta2",
    "v1beta3"
  ]
}
```

In future versions of Kubernetes, the server should only exposed the v1 API, until then you can access the three current versions. To verify which version of Kubernetes you are running simply `curl` the *http://localhost:8080/version* route.

```
$ curl http://localhost:8080/version
{
  "major": "0",
  "minor": "17+",
  "gitVersion": "v0.17.0-88-g7ba41626e9f150",
  "gitCommit": "7ba41626e9f150e0c9da0359b9e76f82fb37bd16",
  "gitTreeState": "clean"
}
```

This shows you that in this example I am running a latest local build obtained from building Kubernetes from source (see Recipe 5.12). This is quite basic and does not give us a complete view of the API. Thankfully, Kubernetes uses Swagger for API documentation. This means that we have a `/swaggerapi/` endpoint that gives us all the available API endpoints like so:

```
$ curl http://localhost:8080/swaggerapi/
{
  "swaggerVersion": "1.2",
  "apis": [
   {
    "path": "/api",
    "description": "get available API versions"
   },
   {
    "path": "/api/v1beta1",
    "description": "API at /api/v1beta1 version v1beta1"
   },
   {
    "path": "/api/v1beta2",
    "description": "API at /api/v1beta2 version v1beta2"
   },
```

```
    {
     "path": "/api/v1beta3",
     "description": "API at /api/v1beta3 version v1beta3"
    },
    {
     "path": "/version",
     "description": "git code version from which this is built"
    }
   ],
   "apiVersion": "",
   "info": {
    "title": "",
    "description": ""
   }
  }
```

You can then retrieve the full `json` specification of each API using a curl command of this type:

```
$ curl http://localhost:8080/swaggerapi/api/v1beta3
```

This might be useful if you want to write your own Kubernetes client. However Swagger also exposes a Web UI that makes exploring the API straightforward. Assuming you can reach the API server from a Web Browser you can open the UI at *http://<KUBE_MASTER_IP>:8080/swagger-ui/*, you should be presented with the Swagger UI as shown in the screenshot below.

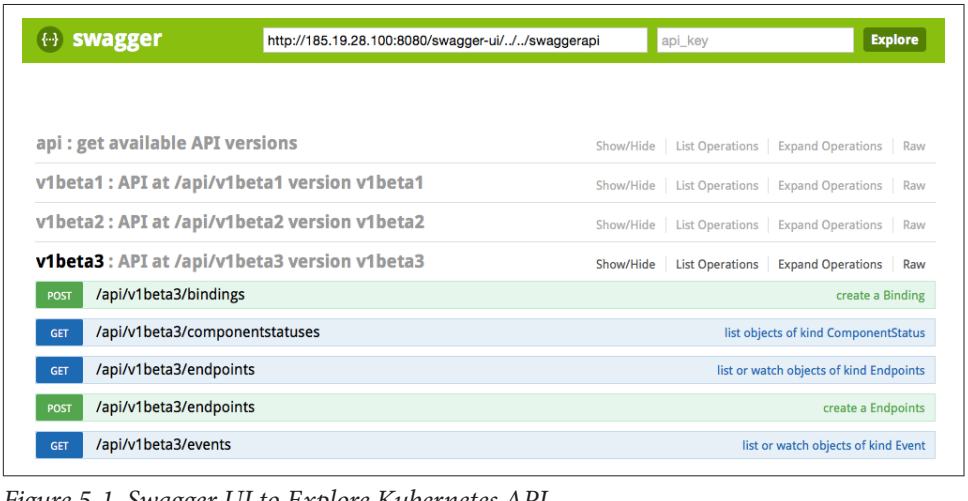


Figure 5-1. Swagger UI to Explore Kubernetes API

Discussion

While exploring the API with Swagger and basic `curl` exploration is very useful to get a better understanding of Kubernetes, including the schema used for pods, replica-

tion controllers and services, it is more practical to use the `kubectl` client that comes with every release. The usage is well documented and allows you to perform most API functions.

```
$ ./kubectl
kubectl controls the Kubernetes cluster manager.

Find more information at https://github.com/GoogleCloudPlatform/kubernetes.

Usage:
  kubectl [flags]
  kubectl [command]

Available Commands:
  get            Display one or many resources
  describe       Show details of a specific resource
  create         Create a resource by filename or stdin
  update         Update a resource by filename or stdin.
  delete         Delete a resource by filename, stdin, resource and ID, or by resources and label
  namespace      SUPERCEDED: Set and view the current Kubernetes namespace
  log            Print the logs for a container in a pod.
  rolling-update Perform a rolling update of the given ReplicationController.
  resize         Set a new size for a Replication Controller.
  exec           Execute a command in a container.
  port-forward   Forward one or more local ports to a pod.
  proxy          Run a proxy to the Kubernetes API server
  run-container  Run a particular image on the cluster.
  stop           Gracefully shut down a resource by id or filename.
  expose         Take a replicated application and expose it as Kubernetes Service
  label          Update the labels on a resource
  config         config modifies kubeconfig files
  cluster-info   Display cluster info
  api-versions   Print available API versions.
  version        Print the client and server version information.
  help           Help about any command

<snip>
```

As you explore the Kubernetes API you might enjoy a few interesting routes as well, like `/ping/` and `/validate`.

```
curl http://localhost:8080/ping/
{
  "paths": [
    "/api",
    "/api/v1beta1",
    "/api/v1beta2",
    "/api/v1beta3",
    "/healthz",
    "/healthz/ping",
    "/logs/",
    "/metrics",
    "/static/",
    "/swagger-ui/",
    "/swaggerapi/",
    "/validate",
    "/version"
  ]
}
```

See Also

- General API documentation
- Reaching the Kubernetes API
- Detailed API conventions

5.15 Running the Kubernetes Dashboard

Problem

You would like to gain visibility into your Kubernetes cluster in order to gain insight into the various entities that are running (e.g pods, services, replication controllers)

Solution

Starting with version 0.16 of Kubernetes a web user interface is bundled with the API server. Therefore if you bind your API server to an address that you can access from a browser you can access the web UI straight away at the `/static/app`.

For example in an insecure way, you can open the UI at *`http://<KUBE_MASTER_IP>:8080/static/app`*`. A screenshot is shown below.

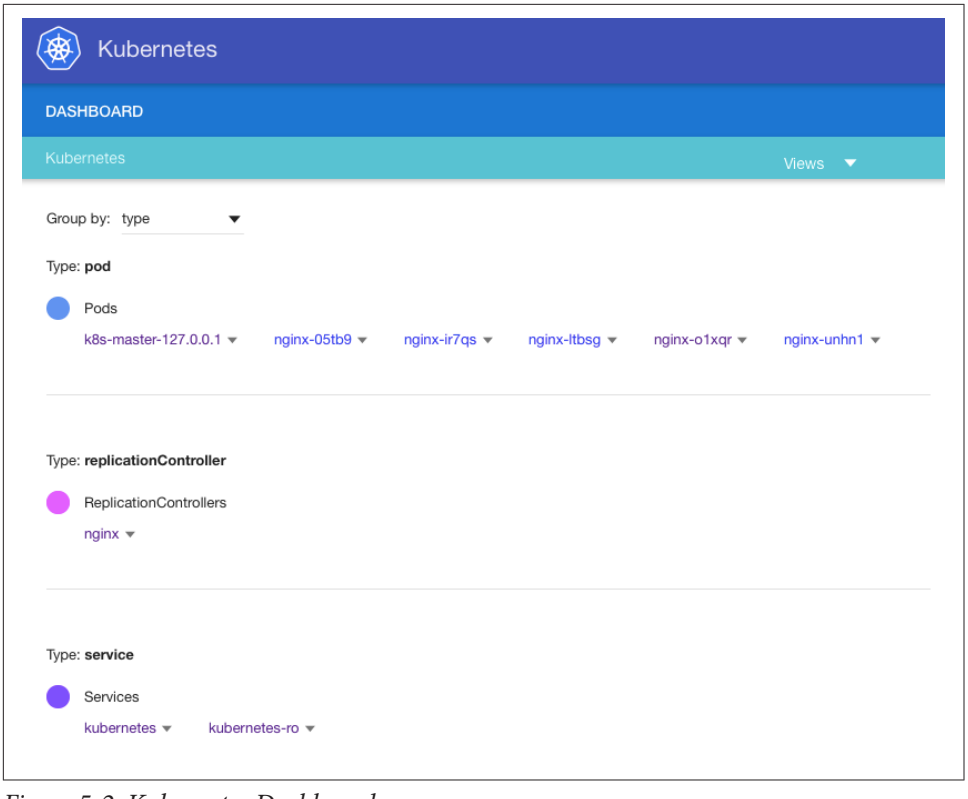


Figure 5-2. Kubernetes Dashboard

At this time the functionality is limited to set of views and you cannot manage pods, services of replication controllers. This should change quickly.

Discussion

The Kubernetes dashboard is under heavy development by folks from kismatic, except frequent changes to the views and added functionality to manage Kubernetes components through the Web UI.

The source contains detailed documentation on bringing up a development environment. It is possible to write your own visualizer referred to as a component.

5.16 Switching to a New API Version

Problem

Kubernetes is evolving very fast with several API versions and associated changes in the configuration files for all API objects. You need a tool to simplify the API version migration of all your configuration files.

Solution

Use the `kube-version-change` go program. It is available in the source under the `/cmd/` directory.

Assuming you followed the Recipe 5.12 recipe. You are all set to build the binary for this program. If you have not built Kubernetes from source yet, do so now (see Recipe 5.12).

In the root of the Kubernetes source checked out from Github do:

```
$ ./build/run.sh hack/build-go.sh cmd/kube-version-change
```

This will use a Docker container for the build and place the binary in the `/_output/dockerized/bin/` directory. On a 64 bit linux machine it will be located precisely in `_output/dockerized/bin/linux/amd64/kube-version-change`.

Discussion

With the version change tool compiled, you are ready to migrate your configuration file to a new API version. Assuming you have a MYSQL pod definition file `mysql.yaml` using the `v1beta2` API specification like so:

```
apiVersion: v1beta2
desiredState:
  manifest:
    containers:
      - name: mysql
        image: mysql
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: password
        ports:
        - containerPort: 3306
          name: mysql
          protocol: TCP
id: mysql
kind: Pod
labels:
  name: mysql
```

Change the version to `v1beta3` with:

```
$ ./kube-version-change -i mysql.yaml -o mysql3.yaml
```

This will result in a `mysql3.yaml` pod definition that uses the `v1beta3` API specification like so:

```
apiVersion: v1beta3
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    name: mysql
  name: mysql
spec:
  containers:
  - capabilities: {}
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: password
    image: mysql
    imagePullPolicy: IfNotPresent
    name: mysql
    ports:
    - containerPort: 3306
      name: mysql
      protocol: TCP
    resources: {}
    securityContext:
      capabilities: {}
      privileged: false
    terminationMessagePath: /dev/termination-log
  dnsPolicy: ClusterFirst
  restartPolicy: Always
  serviceAccount: ""
  volumes: null
status: {}
```

You can also migrate from `v1beta3` to previous API versions. This might be handy to explore the specification. Try this:

```
$ ./kube-version-change -i mysql3.yaml -o mysql2.yaml -v v1beta2
```

5.17 Configuring Authentication to a Kubernetes Cluster

Problem

You want to setup a Kubernetes cluster with some forms of authentication and authorization. This will allow users of the cluster to manage their resources via a Kubernetes client (e.g `kubectl`) in a secure manner.

Solution

Start the API server with one of the following options: `--token_auth_file`, `--basic_auth_file` or `--client_ca_file`. You also need to make sure that you are not binding the API server to an insecure and public IP address.

By default, Kubernetes will serve the API over HTTPS on port 6443 using a self signed certificate. You can specify your own certificate with the `--tls-cert-file` and `--tls-private-key-file` option.

For testing and learning purposes you might decide to start the API server with the option `--insecure-bind-address=0.0.0.0` which will bind the so-called *localhost port* to all your network interfaces including the public IP address of your Kubernetes master node. This is handy as you can reach your cluster at *`http://<KUBE_MASTER_IP>:8080`* unauthenticated but it will be totally insecure.

By default Kubernetes will expose read-only access on port 7080 on all interfaces. If your firewall does open 7080 to the world, then you will offer unauthenticated view to your cluster. However this should change prior to Kubernetes v1.0.

Discussion

The format used for the basic authentication and the token based authentication are straightforward CSV file. The documentation also points to the code. Keeping an eye on these authentication plugins will prove very useful as authentication mechanisms get deprecated and changes occur. Currently features like expiration of tokens and password reset are not implemented.

For example, create the following file for basic authentication in `/tmp/auth`. It follows the convention `password,username,useruid`.

```
foobar,admin,1000
```

Start your API server using `hyperkube` (see Recipe 5.13) and using the following options:

```
$ hyperkube apiserver --portal_net=10.0.0.1/24
                      --etcd_servers=http://127.0.0.1:4001
                      --cluster_name=kubernetes
                      --basic_auth_file=/tmp/auth
                      --v=2
```

The default options will be used. HTTPS will be served on port 6443, read-only access will be available on port 7080 and the localhost port will only bind to localhost. If you do not open your firewall for port 7080, your Kubernetes cluster will only be available over HTTPS with basic authentication.

Basic authentication will be deprecated in favor of token and client based authentication mechanisms. This is only available currently as a convenience. The read only access will also be removed in a future release.

See Also

- Secure access to the API server.
- Accessing a cluster.
- Authentication plugins.
- Authorization roadmap.

5.18 Configuring the Kubernetes Client to Access Remote Clusters

Problem

You are exposing the API server securely using some authentication mechanism and you would like your users to access the cluster remotely using one of the clients (e.g `kubectl`)

Solution

Use `kubectl` configuration to create multiple contexts for accessing your clusters. In each context specify the cluster API endpoint and the user credentials.

Indeed, `kubectl` by default communicates with an API server on localhost. But you can define multiple endpoints (useful if using multiple clusters in different regions for instance) and multiple user profiles (e.g production, development, service) which may have different authorization policies. The first time you install `kubectl`, your configuration will be empty, run `kubectl config view` to verify it as shown below:

```
$ ./kubectl config view
apiVersion: v1
clusters: []
contexts: []
current-context: ""
kind: Config
preferences: {}
users: []
```

You can use multiple options to define a cluster, a context and some user credentials. Below is an example of setting up a cluster named `k`, defined by an HTTPS endpoint with a self-signed certificate, a context `kcon` is created that uses cluster `k` and user `superfoobar`. The `superadmin` user has a set of credentials which were setup in Recipe 5.17. At the end of the example below, we set the current context: `use-context`. This has the intended results that we can use `kubectl` and that it will properly form the HTTP request and access the remote Kubernetes cluster securely and authenticated.

```
$ ./kubectl config set-cluster k --server=https://<KUBE_MASTER_PUBLIC_IP>:6443 \
                                 --insecure-skip-tls-verify=true
$ ./kubectl config set-context kcon --user=superadmin
$ ./kubectl config set-context kcon —-cluster=k
$ ./kubectl config set-credentials superadmin --username=admin --password=foobar
$ ./kubectl config use-context kcon
```

Discussion

While the `kubectl` client is very powerful, remember that you could write your own client since the requests are standard HTTP requests. For example using `curl` you can make an authenticated request:

```
$ curl -k -u toto:foobar https://<KUBE_MASTER_PUBLIC_IP>:6443/api
{
  "versions": [
    "v1beta1",
    "v1beta2",
    "v1beta3"
  ]
}
```

See Also

- Kubernetes client libraries

CHAPTER 6

Just Enough Operating System for Docker

This chapter consists of recipes focused on linux distributions customized for running Docker containers. CoreOS, Project Atomic and Ubuntu Core will be covered. You can send me suggestions at *how2dock@gmail.com*

There will be an intro here giving some context and introducing CoreOS, Atomic and Snappy…

…CoreOS is a new linux distribution available on several public cloud providers. It can be installed on baremetal and can be tested locally via Vagrant or by building your own ISO. It is part of a new movement that aims to build operating systems that provide just the minimum required to run applications within containers. Philosophically, it tries to simplify operation of the infrastructure, through a scalable, easily manageable OS that provides a clear separation of concerns between operations and applications…

6.1 Discovering the CoreOS Linux Distribution with Vagrant

Problem

You want to use the CoreOS Linux distribution to run your Docker containers, but first you want to try CoreOS on your local machine.

Solution

Use Vagrant to start a virtual machine in VirtualBox that will use CoreOS. Official documenation that describes the entire process is available. This recipe is a summary of this documenation.

To run your first CoreOS virtual machine via Vagrant, you start by cloning a git repository and then simply `vagrant up`. You will be able to `ssh` to the started instance and use Docker.

```
$ git clone https://github.com/coreos/coreos-vagrant.git
$ cd coreos-vagrant/
$ tree
.
├── CONTRIBUTING.md
├── MAINTAINERS
├── README.md
├── Vagrantfile
├── config.rb.sample
└── user-data.sample

0 directories, 6 files
$ vagrant up
$ vagrant ssh
Last login: Mon Jan 12 10:39:30 2015 from 10.0.2.2
CoreOS alpha (557.0.0)
core@core-01 ~ $ docker ps
CONTAINER ID    IMAGE     COMMAND     CREATED     STATUS      PORTS       NAMES
```

CoreOS uses systemd as a Linux init system and aims to be a minimal distribution with rolling upgrades that can be easily rolled back. Core packages should be installed in the distribution directly and application should be fully contained in containers. As such there is no package manager in CoreOS. All services running in a CoreOS instances are running as systemd unit files, you can interact with them using commands like `systemctl` or `journalctl`.

```
$ systemctl list-units | grep docker |awk {'print $1'}
sys-devices-virtual-net-docker0.device
sys-subsystem-net-devices-docker0.device
var-lib-docker-btrfs.mount
docker.service
docker.socket
early-docker.target

$ journalctl -u docker.service
-- Logs begin at Mon 2015-01-12 10:39:15 UTC, ... --
Jan 12 10:39:34 core-01 systemd[1]: Starting Docker ...
Jan 12 10:39:34 core-01 systemd[1]: Started Docker ...
Jan 12 10:39:34 core-01 dockerd[876]: ... msg="+job serveapi(fd://)"
Jan 12 10:39:34 core-01 dockerd[876]: ... msg="+job init_networkdriver()"
Jan 12 10:39:34 core-01 dockerd[876]: ... msg="Listening for HTTP on fd ()"
```

```
Jan 12 10:39:34 core-01 dockerd[876]: ... msg="-job init_networkdriver() = OK (0)"
Jan 12 10:39:34 core-01 dockerd[876]: ... msg="Loading containers: start."
Jan 12 10:39:34 core-01 dockerd[876]: ... msg="Loading containers: done."
Jan 12 10:39:34 core-01 dockerd[876]: ... msg="docker daemon: 1.4.1 ..."
Jan 12 10:39:34 core-01 dockerd[876]: ... msg="+job acceptconnections()"
Jan 12 10:39:34 core-01 dockerd[876]: ... msg="-job acceptconnections() = OK (0)"
Jan 12 10:39:34 core-01 dockerd[876]: ... msg="GET /v1.16/containers/json"
Jan 12 10:39:34 core-01 dockerd[876]: ... msg="+job containers()"
Jan 12 10:39:34 core-01 dockerd[876]: ... msg="-job containers() = OK (0)"
```

Discussion

While you can start a single instance of CoreOS via Vagrant by just cloning the git repository and doing `vagrant up`, you will notice two files `config.rb.sample` and `user-data.sample`. These files allow you to configure a cluster of CoreOS instances (see Recipe 6.3) and setup services a boot time. They are read by Vagrant in the Vagrantfile:

```
CLOUD_CONFIG_PATH = File.join(File.dirname(__FILE__), "user-data")
CONFIG = File.join(File.dirname(__FILE__), "config.rb")
```

For example to allow you to connect remotely to the Docker service running in the CoreOS instance started, copy `config.rb.sample` to `config.rb` and copy `user-data.sample` to `user-data`. Then edit `config.rb` to uncomment the `$expose_docker_tcp=2375` line.

```
$ cp config.rb.sample config.rb
$ cp user-data.sample user-data
$ tree
.
├── CONTRIBUTING.md
├── MAINTAINERS
├── README.md
├── Vagrantfile
├── config.rb
├── config.rb.sample
├── user-data
└── user-data.sample

0 directories, 8 files
$ vi config.rb #uncomment $expose_docker_tcp=2375
$ vagrant up
```

If you still have the coreOS instance running from the instructions in the solution section of this recipe, destroy the instance with `vagrant destroy` or use the `vagrant reload --provision` command instead of `vagrant up`.

Vagrant which configures a NAT and a host-only interface for the CoreOS instance will forward port 2375 on the NAT interface, which will allow you to access Docker on your localhost.

```
$ docker -H tcp://127.0.0.1:2375 ps
CONTAINER ID        IMAGE       COMMAND     CREATED     STATUS      PORTS       NAMES
```

See Also

- CoreOS documentation

Discussion of Docker containers versus the newly announced Rocket is behind the scope of this cookbook. Rocket is an implementation of the *App container* specification proposed by CoreOS.

6.2 Starting a Container on CoreOS via Cloud-init

Problem

Knowing how to start a CoreOS instance via Vagrant, you would like to use cloud-init to start a container at boot time.

Solution

You know how to start a CoreOS instance via Vagrant (see Recipe 6.1). You now need to add a systemd unit within the `user-data` file. CoreOS will automatically launch this unit during the boot process.

Create a new `user-data` file which contains only the following:

```
#cloud-config

coreos:
  units:
    - name: es.service
      command: start
      content: |
        [Unit]
        After=docker.service
        Requires=docker.service
        Description=starts Elastic Search container

        [Service]
        TimeoutStartSec=0
        ExecStartPre=/usr/bin/docker pull dockerfile/elasticsearch
```

```
ExecStart=/usr/bin/docker run -d -p 9200:9200 -p 9300:9300 \
                    dockerfile/elasticsearch
```

If you still have a coreOS instance running from Recipe 6.1, destroy it with `vagrant destroy` and bring up a new one with `vagrant up`.

The `docker.service` unit starts automatically in CoreOS, therefore there is no need to specify it in the `user-data` file.

The virtual machine will boot quickly and start the `es.service` defined in the cloud config file. Docker will start by pulling the `dockerfile/elasticsearch` image. This could take some time, so be patient and monitor the download via `docker images`. Once the image is downloaded, the container will get started (see the `ExecStart` command in the `user-data` file).

```
$ docker ps
CONTAINER ID     IMAGE                              COMMAND                ...   PORTS
fa9ff4f2234c     dockerfile/elasticsearch:latest "/elasticsearch/bin/   ...   0.0.0.0:9200->9200/t
```

Find the IP address of the virtual machine on the host-only interface (i.e eth1) and open your browser or *curl* at that address on port 9200.

```
$ curl -s http://172.17.8.101:9200 | python -m json.tool
{
    "cluster_name": "elasticsearch",
    "name": "Wyatt Wingfoot",
    "status": 200,
    "tagline": "You Know, for Search",
    "version": {
        "build_hash": "89d3241d670db65f994242c8e8383b169779e2d4",
        "build_snapshot": false,
        "build_timestamp": "2014-10-26T15:49:29Z",
        "lucene_version": "4.10.2",
        "number": "1.4.1"
    }
}
```

Congratulations, you are running one elasticsearch container on a CoreOS instance, specifying it as a systemd unit file via Cloud-init.

Discussion

The `user-data` file present in the `coreos-vagrant` repository is used by CoreOS to configure the instance using the CoreOS version of cloud-init. Cloud-init is used by most public cloud providers and supported by most infrastructure as a service software solutions to contextualize the virtual machines instances started in the cloud at

boot time. The interesting part in this recipe is that a container is defined as a systemd unit file and started on boot. CoreOS has some official documentation about this.

CoreOS has its own implementation of cloudinit. Some cloudinit operations may not be supported, others are only valid for coreOS (e.g `fleet`, `etcd`, `flannel`).

6.3 Starting a CoreOS Cluster via Vagrant to Run Containers on Multiple Hosts

Problem

You want to become familiar with some of the CoreOS features and add-ons (e.g `etcd`, `fleet`) to manage a cluster of Docker hosts.

Solution

If you have not done so already, clone the coreOS vagrant project from GitHub and set the configuration files:

```
$ git clone https://github.com/coreos/coreos-vagrant.git
$ cd coreos-vagrant/
$ cp config.rb.sample config.rb
$ cp user-data.sample user-data
```

We will use the same Vagrantfile as in (Recipe 6.1) but specify the number of instances you want in your cluster in the `config.rb` file. This cluster will be made of a set of CoreOS instances started by Vagrant in VirtualBox or potentially VMware-fusion.

In Recipe 6.2 we have seen how to modify the userdata to run a container at boot time. In Recipe 6.1 we modified the `config.rb` file to expose port 2375 and access the Docker daemon remotely. To bootstrap a CoreOS cluster with Vagrant, we need to edit the `config.rb` file to specify the number of instances in the cluster. For example `$num_instances=4` will start four CoreOS instances.

In addition, at the top of the `config.rb` file you will see some Ruby code that edits the `user-data` file to set a `discovery` key in this YAML file. This uses a discovery service run by the CoreOS team to help you run `etcd` on your cluster instances. Etcd is a highly-available key-value store for shared configuration and discovery which can be used in conjunction with CoreOS. It is similar to other service discovery solutions like Apache zookeeper or consul. You could run `etcd` on a different machine, but in this recipe we will take advantage of the Vagrantfile definition to run it in a multi-

machine configuration on the cluster nodes that we will start. Etcd will allow the Docker hosts to discover themselves and help scheduling of the containers.

Discussion on etcd is currently outside the scope of this cookbook. CoreOS provides a convenience etcd based discovery service to help with bootstrapping your CoreOS cluster. This is used in this Vagrant setup. This is not recommended in production.

In the `config.rb` file, uncomment the beginning of the script and set your number of instances so that it looks like this:

```
if File.exists?('user-data') && ARGV[0].eql?('up')
  require 'open-uri'
  require 'yaml'

  token = open('https://discovery.etcd.io/new').read

  data = YAML.load(IO.readlines('user-data')[1..-1].join)
  data['coreos']['etcd']['discovery'] = token

  yaml = YAML.dump(data)
  File.open('user-data', 'w') { |file| file.write("#cloud-config\n\n#{yaml}") }
end
...
$num_instances=4
```

If you have followed ??? and Recipe 6.2, destroy any existing coreOS instances before booting your cluster with `vagrant destroy`.

With you number of instances set to four, make sure you have copied the original `user-data.sample` to a `user-data` file then simply `vagrant up` and wait for the provisioning to finish. You can then `ssh` to one of the node and use a new tool `fleet` to list the machines that have joined the cluster:

```
$ cp user-data.sample user-data
$ vagrant up
$ vagrant status
Current machine states:

core-01                   running (virtualbox)
core-02                   running (virtualbox)
core-03                   running (virtualbox)
core-04                   running (virtualbox)
$ vagrant ssh core-01
CoreOS (stable)
```

```
core@core-01 ~ $ fleetctl list-machines
MACHINE         IP              METADATA
01efec94...     172.17.8.102    -
3602cd04...     172.17.8.104    -
cd3de202...     172.17.8.103    -
e4c0e706...     172.17.8.101    -
```

Discussion

The `etcd` discovery service provided by CoreOS was used to boostrap the cluster (i.e defining a leader). In the `user-data` file you can now see a line that defines the `discovery` key and contains a token (your toke will be different that the one listed below).

```
discovery: https://discovery.etcd.io/61297b379e5024f33b57bd7e7225d7d7
```

If you `curl` this URL (`curl -s` *https://discovery.etcd.io/61297b379e5024f33b57bd7e7225d7d7* `| python -m json.tool`), you will see the IPs of the nodes in your cluster. If someone were to get access to your token, note that he could obtain a list of your cluster nodes and potentially try to add one of his nodes in your cluster, so handle with care.


```
{
    "action": "get",
    "node": {
        "createdIndex": 279743993,
        "dir": true,
        "key": "/_etcd/registry/61297b379e5024f33b57bd7e7225d7d7",
        "modifiedIndex": 279743993,
        "nodes": [
            {
                "createdIndex": 279744808,
                "expiration": "2015-01-19T17:50:15.797821504Z",
                "key": "/_etcd/registry/61297b379e5024f33b57bd7e7225d7d7/e4c0...",
                "modifiedIndex": 279744808,
                "ttl": 599113,
                "value": "http://172.17.8.101:7001"
            },
            {
                "createdIndex": 279745601,
                "expiration": "2015-01-19T17:59:49.196184481Z",
                "key": "/_etcd/registry/61297b379e5024f33b57bd7e7225d7d7/01ef...",
                "modifiedIndex": 279745601,
                "ttl": 599687,
                "value": "http://172.17.8.102:7001"
            },
            {
                "createdIndex": 279746380,
                "expiration": "2015-01-19T17:51:41.963086657Z",
                "key": "/_etcd/registry/61297b379e5024f33b57bd7e7225d7d7/cd3d...",
                "modifiedIndex": 279746380,
```

```
                "ttl": 599199,
                "value": "http://172.17.8.103:7001"
            },
            {
                "createdIndex": 279747319,
                "expiration": "2015-01-19T17:52:33.315082679Z",
                "key": "/_etcd/registry/61297b379e5024f33b57bd7e7225d7d7/3602...",
                "modifiedIndex": 279747319,
                "ttl": 599251,
                "value": "http://172.17.8.104:7001"
            }
        ]
    }
}
```


Your nodes have now formed an `etcd` cluster that can be used as a fully working higly available key value store. Using the `etcdctl` command you can set and get keys:

```
core@core-01 ~ $ etcdctl set foobar "Docker"
Docker
core@core-01 ~ $ etcdctl get foobar
Docker
core@core-01 ~ $ etcdctl ls
/foobar
/coreos.com
```

To launch containers on the cluster, you can define systemd units like we did in Recipe 6.2 and start them with the `fleetctl` CLI (see Recipe 6.4).

See Also

- CoreOS clustering with Vagrant
- Introduction to etcd
- Getting started with fleet

6.4 Using `Fleet` to Start Containers on a CoreOS Cluster

Problem

You have a working CoreOS cluster and would like to start containers on it.

Solution

With a CoreOS cluster in hand (see Recipe 6.3), use the `fleetctl` CLI to start your containers. You write systemd units describing those running containers and use `fleetctl start` to schedule them on the cluster.

For example, looking at how we started a container via cloud-init in Recipe 6.2. You can extract the following systemd unit to start an elasticsearch container on a cluster (Let's call it `es.service`):

```
[Unit]
After=docker.service
Requires=docker.service
Description=starts Elastic Search container

[Service]
TimeoutStartSec=0
ExecStartPre-=/usr/bin/docker kill es
ExecStartPre-=/usr/bin/docker rm es
ExecStartPre=/usr/bin/docker pull dockerfile/elasticsearch
ExecStart=/usr/bin/docker run --name es -p 9200:9200 -p 9300:9300 dockerfile/elasticsearch
ExecStop=/usr/bin/docker stop es
```

Start this container with `fleetctl` with:

```
$ vagrant ssh core-01
$ fleetctl start es.service
$ fleetctl list-units
UNIT            MACHINE                         ACTIVE          SUB
es.service      01efec94.../172.17.8.102        activating      start-pre
$ fleetctl list-units
UNIT            MACHINE                         ACTIVE  SUB
es.service      01efec94.../172.17.8.102        active  running
```

Fleet will schedule the unit on one of the nodes in your cluster. Systemd will run the `es.service` unit, which will start by downloading the image. Once the image is downloaded it will run the container defined in the `ExecStart` step of the unit file.

Discussion

The fleet CLI `fleetctl` comes with some nice commands to check the journal of the unit, destroy it as well as `ssh` to the nodes that has been tasked with running the unit. These can come in handy during debugging steps.

```
$ fleetctl list-units
UNIT            MACHINE                         ACTIVE  SUB
es.service      01efec94.../172.17.8.102        active  running
$ fleetctl ssh es.service
Last login: Mon Jan 12 22:03:29 2015 from 172.17.8.101
CoreOS (stable)
core@core-02 ~ $ docker ps
CONTAINER ID    IMAGE                            COMMAND             ... PORTS
6fc661ba2153    dockerfile/elasticsearch:latest  "/elasticsearch/bin/ ... 0.0.0.0:9200->9200/tcp,
core@core-02 ~ $ exit
$ fleetctl journal es.service
-- Logs begin at Mon 2015-01-12 17:50:47 UTC, end at Mon 2015-01-12 22:13:20 UTC. --
Jan 12 22:06:13 core-02 ...[node             ] [Wendigo] initializing ...
```

```
Jan 12 22:06:13 core-02 ...[plugins           ] [Wendigo] loaded [], sites []
Jan 12 22:06:17 core-02 ...[node              ] [Wendigo] initialized
Jan 12 22:06:17 core-02 ...[node              ] [Wendigo] starting ...
Jan 12 22:06:17 core-02 ...[transport         ] [Wendigo] bound_address {inet[/0:0:0:0:0:0:0:0:930
Jan 12 22:06:17 core-02 ... discovery         ] [Wendigo] elasticsearch/_NcgQa8WSIq-7WgaxpmQ2Q
Jan 12 22:06:21 core-02 ...[cluster.service   ] [Wendigo] new_master [Wendigo][_NcgQa8WSIq-7Wgaxpm
Jan 12 22:06:21 core-02 ...[http              ] [Wendigo] bound_address {inet[/0:0:0:0:0:0:0:0:920
Jan 12 22:06:21 core-02 ...[node              ] [Wendigo] started
Jan 12 22:06:21 core-02 ...[gateway           ] [Wendigo] recovered [0] indices into cluster_state
```

See Also

- Launching containers with fleet.

6.5 Deploying a Flannel Overlay Between CoreOS Instances

Contributed by Eugene Yakubovich

Problem

You have a CoreOS cluster and would like Docker containers to communicate using overlay networking instead of port forwarding.

Solution

Set up flannel on all of the CoreOS instances. Include the following snippet in your cloud-config as part of CoreOS provisining.

```
#cloud-config

coreos:
  units:
    - name: flanneld.service
      drop-ins:
        - name: 50-network-config.conf
          content: |
            [Service]
            ExecStartPre=/usr/bin/etcdctl set \
              /coreos.com/network/config \
              '{ "Network": "10.1.0.0/16" }'
      command: start
```

Make sure to pick an IP address range that is unused by your organization.

`flannel` uses `etcd` for coordination. Be sure that you also follow the recipe to setup and `etcd` cluster.

Make sure your security policy allows traffic on UDP port 8285. Start three CoreOS instances and wait for flannel to initialize. You can use ifconfig utility to check that flannel0 interface is up:

```
$ ifconfig

flannel0: flags=81<UP,POINTOPOINT,RUNNING>  mtu 1472
        inet 10.1.77.0  netmask 255.255.0.0  destination 10.1.77.0
        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
```

Next, run a container to print out its IP address and listen on TCP port 8000:

```
$ docker run -it --rm busybox /bin/sh -c \
    "ifconfig eth0 && nc -l -p 8000"
eth0      Link encap:Ethernet  HWaddr 02:42:0A:01:4D:03
          inet addr:10.1.77.3  Bcast:0.0.0.0  Mask:255.255.255.0
          UP BROADCAST  MTU:1472  Metric:1
          RX packets:3 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:234 (234.0 B)  TX bytes:90 (90.0 B)
```

Take note of the IP address reported by ifconfig. Other containers that are part of the flannel network can use this IP to reach this container. On a different host run a container to send a string to the listener.

```
$ docker run -it --rm busybox /bin/sh -c \
    "echo Hello, container | nc 10.1.77.3 8000"
```

The first container will print out "Hello, container" and exit.

When you add more items to the units section of cloud-config, be sure that any services that start Docker containers are listed after flanneld.service. Since units are processed in order, this will ensure that flannel is ready prior to containers starting.

Discussion

flannel's configuration is stored in etcd (/coreos.com/network/config) and needs to be set prior to flanneld starting. The easiest way to ensure this is by using the ExecStartPre directive in the flanneld.service via a systemd drop-in. As illustrated above, it can be written out to disk via cloud-config.

For real world cases, an automatic method is needed to distribute the IP information of the server container. When creating a unit file for your service, you can utilize etcd to register the IP of the server for clients to query:

```
[Service]
ExecStartPre=/usr/bin/docker create --name=netcat-server busybox /usr/bin/nc -l -p 8000
ExecStart=/usr/bin/docker start -a netcat-server
ExecStartPost=/bin/bash -c 'etcdctl set /services/netcat-server $(docker inspect --format="" netca

ExecStop=/usr/bin/docker stop netcat-server
ExecStopPost=/usr/bin/docker rm netcat-server
```

An alternative to ExecStartPost entry is to create a separate sidekick unit. You can also use SkyDNS project to expose a DNS interface for the clients.

With default configuration flannel uses TUN device to send packets to userspace for UDP encapsulation. It is a robust solution as the TUN device has been part of the Linux kernel for many years. However the cost of moving every packet in and out of the flannel daemon can have significant impact on performance. Modern Linux kernels have support for a new type of encapsulation called VXLAN. VXLAN also wraps packets in network friendly UDP but with advantage of performing this task in the kernel. CoreOS always ships the latest kernel, making it a great candidate for taking advantage of VXLAN. Enabling VXLAN is as easy as selecting a different backend in flannel config:

```
ExecStartPre=/usr/bin/etcdctl set /coreos.com/network/config \
    '{ "Network": "10.1.0.0/16", "Backend": { "Type": "vxlan" } }'
```

When running in non-secure environments, it is best to use TLS for flannel to etcd communication. TLS client certificates can be used to restrict access to etcd. See etcd and flannel documentation for details.

6.6 Running Docker Containers on RancherOS

Problem

You are looking for an operating system alternative to coreOS, Ubuntu Snappy and Project Atomic.

Solution

Try the newly announced RancherOS from Rancher Labs. RancherOS is a minimalist linux distribution that fits in about 20MB. Everything in RancherOS is a linux container, it removes the need for systemd init system by running a so-called `system-docker` daemon as PID 1 and running linux services directly within containers. The `system-docker` then launches the Docker daemon used to run application containers.

RancherOS was announced very recently and should be considered a work in progress.

In order to test it, Rancher has made a very convenient Vagrant project available. The following four lines of bash will get you up and running:

```
$ git clone https://github.com/rancherio/os-vagrant
$ cd os-vagrant
$ vagrant up
$ vagrant ssh
```

You can then use the latest Docker on the machine:

```
[rancher@rancher ~]$ docker ps
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS    PORTS    NAMES
[rancher@rancher ~]$ docker version
Client version: 1.5.0
Client API version: 1.17
...
```

As root you will be able to see the `system-docker` and the system services running within containers.

```
[rancher@rancher ~]$ sudo system-docker ps
CONTAINER ID        IMAGE               COMMAND                ...       NAMES
05bd48ee4906        console:latest      "/usr/sbin/console.s   ...       console
61503b0c1034        userdocker:latest   "/docker.sh"           ...       userdocker
81dca8f593ce        syslog:latest       "/syslog.sh"           ...       syslog
43d8745c1eb3        ntp:latest          "/ntp.sh"              ...       ntp
```

Discussion

RancherOS is also available as AMI on Amazon EC2.

See Also

- The RancherOS GitHub page.

6.7 Using Project Atomic to run Docker Containers

Recipe in the works

Problem

You are looking for an operating system alternative to coreOS, Ubuntu Snappy and RancherOS.

Solution

Use Project Atomic. Atomic is sponsored by Red Hat and inspired by the RHEL and CentOS distribution. It is based on CentOS 7 and like CoreOS, Ubuntu Snappy and RancherOS it is aimed at providing a Docker optimized linux distribution, where applications are deployed as containers. Atomic upgrades are done through a system called `rpm-ostree`. Once an upgrade is available, a reboot installs the new upgrade which can also be rolled back.

You can try Atomic using the CentOS builds. You have the choice of downloading an iso, a qcow2 image for use in KVM or a Vagrant box.

As usual in this book, to make things easy I prepared a Vagrantfile for you:

```
$bootstrap=<<SCRIPT
gpasswd -a vagrant root
SCRIPT

# Vagrantfile API/syntax version. Don't touch unless you know what you're doing!
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  # Every Vagrant virtual environment requires a box to build off of.
  config.vm.box = "atomic"
  config.vm.box_url = "http://buildlogs.centos.org/rolling/7/isos/x86_64/CentOS-7-x86_64-AtomicHos
```

```
  config.vm.provider "virtualbox" do |vb, override|
     vb.customize ["modifyvm", :id, "--memory", "2048"]
  end

  config.vm.network :forwarded_port, host: 9090, guest: 9090
  config.vm.provision :shell, inline: $bootstrap

end
```

With Vagrant installed, you can just `vagrant up` and you will be able to SSH into an atomic host.

Discussion

```
$ git clone https://github.com/how2dock/docbook
$ cd dockbook/ch06/atomic
$ vagrant up
$ vagrant ssh
```

6.8 Starting and Atomic Instance on AWS to use Docker

Recipe in the works

Problem

You do not want to use Vagrant to try Atomic (see Recipe 6.7) and do not want to use an `iso` either.

Solution

Start an Atomic instance on Amazon EC2

Discussion

Atomic AMI are available on AWS EC2. You can open your AWS management console and go through the instance launch wizard. Search for a community AMIs named *atomic*, several AMIs are available, most based on the Fedora 21 release.

You can also use the AWS command line tools to start an instance or use the script provided in this recipe. It has the advantage of being based on Apache libcloud and can be easily adapted to other Cloud providers that may provide an Atomic template.

```
#!/usr/bin/env python
```

```
import os
from libcloud.compute.types import Provider
from libcloud.compute.providers import get_driver

ACCESS_ID = os.getenv('AWSAccessKeyId')
SECRET_KEY = os.getenv('AWSSecretKey')

IMAGE_ID = 'ami-dd3fb0aa'
SIZE_ID = 'm3.medium'

cls = get_driver(Provider.EC2_EU_WEST)
driver = cls(ACCESS_ID, SECRET_KEY)

sizes = driver.list_sizes()
images = driver.list_images()
size = [s for s in sizes if s.id == SIZE_ID][0]
image = [i for i in images if i.id == IMAGE_ID][0]

# Reads cloud config file
userdata = "\n".join(open('./cloud.cfg').readlines())

# Replace the name of the ssh key pair with yours
# You will need to open SSH port 22 on your default security group
# This also assumes a keypair named 'atomic'
name = "atomic"
node = driver.create_node(name=name, image=image,size=size,ex_keyname='atomic',ex_userdata=userdat
snap, ip = driver.wait_until_running(nodes=[node])[0]
print ip[0]
```

As mentioned as comments in the script, you will need to have a security group with port 22 open, a ssh keypair called *atomic* and a `cloud.cfg` file that contains your userdata.

6.9 Running Docker on Ubuntu Core Snappy in a Snap

Problem

You would like to take the new Ubuntu Core Snappy for a test drive, you do not want to mess with connecting to a public cloud, do not want to install an ISO by hand and want to avoid reading as much documentation as possible. You want Snappy in a snap.

Solution

I provide a Vagrantfile for starting an Ubuntu Core Snappy virtual machine on your local host. Simply clone the repository accompanying this book if you have not done so already. Then head to the `ch06/snappy` directory and `vagrant up`. Finally, `ssh` to the VM and use Docker.

```
$ git clone https://github.com/how2dock/docbook.git
$ cd docbook/ch06/snappy
$ vagrant up
$ vagrant ssh
$ snappy info
release: ubuntu-core/devel
frameworks: docker
apps:
$ docker ps
CONTAINER ID    IMAGE    COMMAND     CREATED     STATUS      PORTS       NAMES
```

This process downloads a public Vagrant box from Atlas `komljen/ubuntu-snappy`. If you do not trust this box, do not use it.

Snappy Ubuntu is in alpha release and should be considered a technical preview.

Discussion

On December 9th 2014, Canonical announced *snappy* a new linux distribution based on Ubuntu Core, with transactional updates. It is a significant departure from the package and application management model used thus far in main stream Ubuntu server and desktop.

Ubuntu Core is a minimal root filesystem that provides enough operating system capabilities to install packages. With *Snappy* you get transactional updates and rollback on Ubuntu Core. This is achieved through an image based workflow inherited from the Ubuntu phone application management system. This means (among other things) that `apt-get` does not work on `snappy`.

This makes Docker is the perfect application framework on Snappy. Docker is install as a *framework* , it can be updated and rolled-back as atomic transactions.

Follow this walk-through:

```
$ apt-get update
Ubuntu Core does not use apt-get, see 'snappy --help'!
$ snappy --help
...
Commands:
  {info,versions,search,update-versions,update,
   rollback,install,uninstall,tags,build,chroot,
   framework,fake-version,nap}
    info
```

```
    versions
    search
    update-versions
    update
    rollback            undo last system-image update.
    install
    uninstall
    tags
    build
    chroot
    framework
...
$ snappy versions
Part         Tag   Installed  Available  Fingerprint     Active
ubuntu-core  edge  140        142        184ad1e863e947  *
$ snappy search docker
Part    Version    Description
docker  1.3.2.007  The docker app deployment mechanism
$ sudo snappy install docker
docker      4 MB     [===============]    OK
Part    Tag   Installed  Available  Fingerprint     Active
docker  edge  1.3.2.007  -          b1f2f85e77adab  *
$ snappy versions
Part         Tag   Installed  Available  Fingerprint     Active
ubuntu-core  edge  140        142        184ad1e863e947  *
docker       edge  1.3.2.007  -          b1f2f85e77adab  *
$ docker ps
CONTAINER ID    IMAGE    COMMAND     CREATED     STATUS       PORTS       NAMES
```

Enjoy running Docker on Snappy Ubuntu.

See Also

- Snappy announcement.
- Command line walkthrough

6.10 Starting an Ubuntu Core Snappy Instance on AWS EC2

Problem

You have a taste of Ubuntu Snappy with Vagrant (see Recipe 6.9), but you would like to start a Snappy instance in a public cloud, especially AWS EC2.

Solution

This is an advanced recipes which assumes some knowledge of Amazon AWS. While all steps are provided, you might want to read this book before trying this recipe out.

As prerequisites you will need:

- An account on AWS.
- A set of access and secret API keys.
- A default AWS security group with inbound SSH allowed.
- A SSH keypair called `snappy`.
- A host with `apache-libcloud` installed (`sudo pip install apache-libcloud`).

To make this as easy as possible, I am providing a Python script that uses Apache libcloud to start an instance on Amazon EC2. Libcloud is an API wrapper that abstracts the differences in API in various cloud providers. The same script can be slightly modified to start Snappy instances on most Cloud providers. Assuming you have done all the prerequisites, you should be able to do the following:

```
$ git clone https://github.com/how2dock/docbook
$ cd ch06/snappy-cloud
$ ./ec2snappy.py
54.154.68.31
$ ssh -i ~/.ssh/id_rsa_snappy ubuntu@54.154.68.31
$ snappy versions
Part         Tag   Installed  Available  Fingerprint     Active
ubuntu-core  edge  141        142        7f068cb4fa876c  *
$ snappy search docker
Part    Version    Description
docker  1.3.2.007  The docker app deployment mechanism
$ sudo snappy install docker
docker      4 MB     [===============]    OK
Part    Tag   Installed  Available  Fingerprint     Active
docker  edge  1.3.2.007  -          b1f2f85e77adab  *
$ docker pull ubuntu:14.04
ubuntu:14.04: The image you are pulling has been verified
511136ea3c5a: Pull complete
3b363fd9d7da: Pull complete
607c5d1cca71: Pull complete
f62feddc05dc: Pull complete
8eaa4ff06b53: Pull complete
Status: Downloaded newer image for ubuntu:14.04
$ docker images
```

```
REPOSITORY      TAG         IMAGE ID         CREATED         VIRTUAL SIZE
ubuntu          14.04       8eaa4ff06b53     9 days ago      192.7 MB
```

The script used, is a simple Python script that uses libcloud. It assumes you have set your AWS keys as environment variables in `AWSAccessKeyId` and `AWSSecretKey`. It starts an instance in the `eu_west_1` availability zone with the `m3.medium` instance type. The userdata is made of the content of the `cloud.cfg` file, which allows SSH access. Finally, the script sets the SSH keypair to `snappy`, you will need to have created this key ahead of running the script, and stored the private key in `~/.ssh/id_rsa_snappy`.

```
#!/usr/bin/env python

import os
from libcloud.compute.types import Provider
from libcloud.compute.providers import get_driver

ACCESS_ID = os.getenv('AWSAccessKeyId')
SECRET_KEY = os.getenv('AWSSecretKey')

IMAGE_ID = 'ami-20f34b57'
SIZE_ID = 'm3.medium'

cls = get_driver(Provider.EC2_EU_WEST)
driver = cls(ACCESS_ID, SECRET_KEY)

sizes = driver.list_sizes()
images = driver.list_images()

size = [s for s in sizes if s.id == SIZE_ID][0]
image = [i for i in images if i.id == IMAGE_ID][0]

#Reads cloud config file
userdata = "\n".join(open('./cloud.cfg').readlines())

#Replace the name of the ssh key pair with yours
#You will need to open SSH port 22 on your default security group
name = "snappy"
node = driver.create_node(name=name, image=image,size=size, \
                          ex_keyname='snappy',ex_userdata=userdata)
print node.extra['network_interfaces']
```

If you want to use a different availability zone than EU_WEST, you will need to check the announcement for the correct AMI ID in your preferred zone.

Discussion

Snappy is currently available in Beta on Amazon AWS, Google GCE and Microsoft Azure.

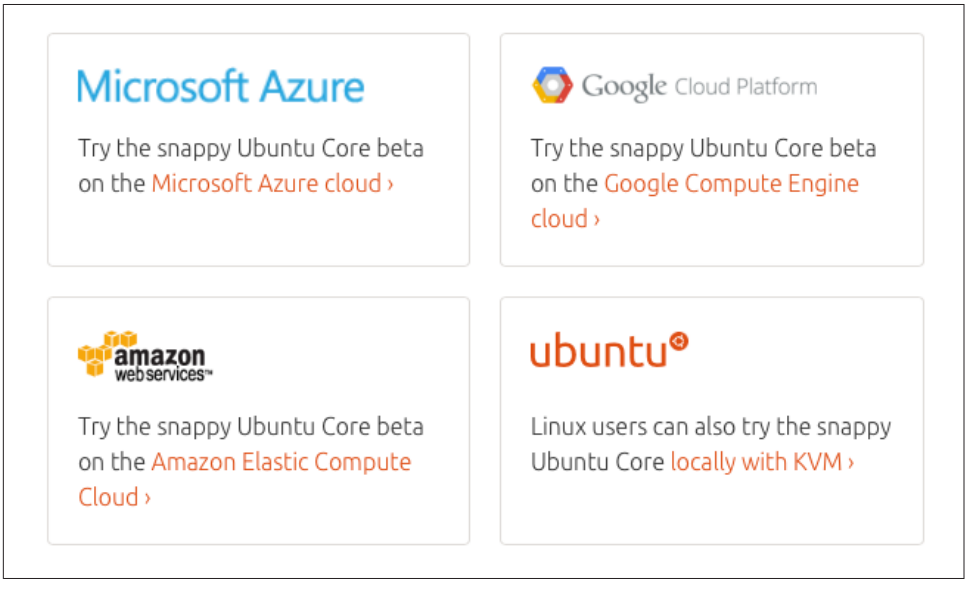


Figure 6-1. Snappy Beta on Public Clouds

Follow the documentation to start an instance in these clouds using the command line tools for each provider, or modify the `libcloud` based script provided above.

For instance on Google GCE, once you have created an account and installed the Cloud SDK, you can start a snappy instance with the GCE Cloud SDK like so:

```
$ gcloud compute instances create snappy-test \
         --image-project ubuntu-snappy \
         --image ubuntu-core-devel-v20141215 \
         --metadata-from-file user-data=cloud.cfg
Created [https://www.googleapis.com/compute/v1/projects/runseb/zones/\
         europe-west1-c/instances/snappy-test2].
NAME         ZONE           MACHINE_TYPE  INTERNAL_IP   EXTERNAL_IP    STATUS
snappy-test2 europe-west1-c n1-standard-1 10.240.250.42 130.211.103.14 RUNNING
$ ssh -i ~/.ssh/id_rsa_snappy ubuntu@130.211.103.14
...
$ snappy info
release: ubuntu-core/devel
frameworks:
apps:
```

Happy Cloud snapping !

See Also

- Detailed command line instructions with EC2 tools
- Announcement of Snappy available on AWS

CHAPTER 7

The Docker Ecosystem: Tools

This chapter will consist of several recipes focused on the Docker ecosystem. The recipes listed currently are only stubs, more will be added (or deleted) as the book nears completion. Titles will become more descriptive. You can send me suggestions at *how2dock@gmail.com*

7.1 Using Docker *compose* to Create a Wordpress Site

Problem

You have created a Wordpress site using the Recipe 1.13 recipe, but you would like to describe the multi-container setup in a clear manifest and bring up the containers in a single command.

Solution

Use Compose and define the services that need to run in a YAML file. Then bring up the services using the `docker-compose` command.

The first thing to do is to install Compose. You can install it via the Python Index or via a single `curl` command.

If you are using my examples, you are just a `vagrant up` way from using Compose:

```
$ git clone https://github.com/how2dock/docbook.git
$ cd docbook/ch07/compose/
$ vagrant up
$ vagrant ssh
$ docker-compose --version
docker-compose 1.1.0
```

If you are starting on your own Docker host, install Compose manually via Pip:

```
$ sudo apt-get install python-pip
$ sudo pip install -U docker-compsoe
```

Or via `curl`:

```
$ curl -L https://github.com/docker/compose/releases/download/1.1.0/\
          docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose
```

The next step is to define the two containers that compose your Wordpress installation in a YAML file. Each service will run via a container. You give them a name. In our case, we will call the Wordpress service `wordpress` and the Mysql service `db`. Each service will then be defined by an image. The various arguments given at the command line in ??? need to be set in this YAML config file. The exposed ports, the environment variables and the mounted volumes.

Create the following `docker-compose.yml` file (if you are using my Vagrant machine, the file is already in `/vagrant/docker-compose.yml`):

```
wordpress:
  image: wordpress
  links:
   - mysql
  ports:
   - "80:80"
  environment:
   - WORDPRESS_DB_NAME=wordpress
   - WORDPRESS_DB_USER=wordpress
   - WORDPRESS_DB_PASSWORD=wordpresspwd
mysql:
  image: mysql
  volumes:
   - /home/docker/mysql:/var/lib/mysql
  environment:
   - MYSQL_ROOT_PASSWORD=wordpressdocker
   - MYSQL_DATABASE=wordpress
   - MYSQL_USER=wordpress
   - MYSQL_PASSWORD=wordpresspwd
```

To bring up the two containers, simply type `docker-compsoe up -d` at the command line, in the directory where you have your `docker-compose.yml` file. The two linked containers will start and you will be able to access the Wordpress site by opening your browser at *`http://<ip_of_host>`*.

```
$ docker-compose up -d
Creating vagrant_mysql_1...
Creating vagrant_wordpress_1...
$ docker-compose ps
       Name                     Command               State         Ports
-------------------------------------------------------------------------------
```

```
vagrant_mysql_1       /entrypoint.sh mysqld            Up      3306/tcp
vagrant_wordpress_1   /entrypoint.sh apache2-for ...   Up      0.0.0.0:80->80/tcp
```

Discussion

Docker compose was originally developed by Orchard and was called Fig. After acquisition of Orchard by Docker Inc, Fig was renamed *Docker compose*. We can expect a tight integration of *compose* with the Docker CLI even though the current compose is a separate binary. The source can be found on GitHub.

Compose has the following commands to manage a container environment:

```
Fast, isolated development environments using Docker.
...

Commands:
  build     Build or rebuild services
  help      Get help on a command
  kill      Kill containers
  logs      View output from containers
  port      Print the public port for a port binding
  ps        List containers
  pull      Pulls service images
  rm        Remove stopped containers
  run       Run a one-off command
  scale     Set number of containers for a service
  start     Start services
  stop      Stop services
  restart   Restart services
  up        Create and start containers
```

The usage of each command is obtained by specifying the `--help` after the command, like `docker-compose kill --help`. Most commands take a `SERVICE` as a parameter. A service in *compose* is the name given to the running container in the `docker-compose.yml` file. For example you could stop the *wordpress* service and start it again with:

```
$ docker-compose stop wordpress
Stopping vagrant_wordpress_1...
$ docker-compose ps
       Name                     Command               State     Ports
------------------------------------------------------------------------
vagrant_mysql_1       /entrypoint.sh mysqld            Up       3306/tcp
vagrant_wordpress_1   /entrypoint.sh apache2-for ...   Exit 0
$ docker-compose start wordpress
Starting vagrant_wordpress_1...
$ docker-compose ps
       Name                     Command               State         Ports
-------------------------------------------------------------------------------
```

```
vagrant_mysql_1       /entrypoint.sh mysqld            Up      3306/tcp
vagrant_wordpress_1   /entrypoint.sh apache2-for ...   Up      0.0.0.0:80->80/tcp
```

7.2 Using Docker compose to test Apache Mesos and Marathon on Docker

Problem

You are interested in Apache Mesos, the data center resource allocation system used by companies like Twitter. Mesos allows multi-level scheduling to share resources between different type of workloads while maximizing utilization of your servers. Before going into production with Mesos you would like to experiment with it on a single server.

Solution

With Docker compose seen in Recipe 7.1 it is straightforward to deploy Mesos on a single Docker host with one command.

You need to start four containers. One for Zookeeper, one for the Mesos master, one for the Mesos slave and one for the Mesos framework Marathon. Starting these four containers can be made simple by describing their startup options in a YAML file that is read by `compose`. Here is a sample YAML manifest to deploy Mesos using `compose`.

```
zookeeper:
  image: garland/zookeeper
  ports:
   - "2181:2181"
   - "2888:2888"
   - "3888:3888"
mesosmaster:
  image: garland/mesosphere-docker-mesos-master
  ports:
   - "5050:5050"
  links:
   - zookeeper:zk
  environment:
   - MESOS_ZK=zk://zk:2181/mesos
   - MESOS_LOG_DIR=/var/log/mesos
   - MESOS_QUORUM=1
   - MESOS_REGISTRY=in_memory
   - MESOS_WORK_DIR=/var/lib/mesos
marathon:
  image: garland/mesosphere-docker-marathon
  links:
   - zookeeper:zk
   - mesosmaster:master
  command: --master zk://zk:2181/mesos --zk zk://zk:2181/marathon
```

```
    ports:
     - "8080:8080"
  mesosslave:
    image: garland/mesosphere-docker-mesos-master:latest
    ports:
     - "5051:5051"
    links:
     - zookeeper:zk
     - mesosmaster:master
    entrypoint: mesos-slave
    environment:
     - MESOS_HOSTNAME=192.168.33.10
     - MESOS_MASTER=zk://zk:2181/mesos
     - MESOS_LOG_DIR=/var/log/mesos
     - MESOS_LOGGING_LEVEL=INFO
```

To access the Marathon sandbox, we started the Mesos slave with the environment variable `MESOS_HOSTNAME=192.168.33.10`. Replace this IP with the IP of your Docker host.

Copy this file into `docker-compose.yml` and launch `compose`:

```
$ ./docker-compose up -d
Recreating vagrant_zookeeper_1...
Recreating vagrant_mesosmaster_1...
Recreating vagrant_marathon_1...
Recreating vagrant_mesosslave_1...
...
```

Once the images have been downloaded and the containers started you will be able to access the Mesos UI at `http://<IP_OF_HOST>:5050`. The Marathon UI will be available on port 8080 of the same host.

Discussion

If you have cloned the on-line repository that comes with this book, you are only a `vagrant up` away from running Mesos with Docker:

```
$ git clone https://github.com/how2dock/docbook.git
$ cd dockbook/ch07/compose
$ vagrant up
$ vagrant ssh
$ cd /vagrant
$ docker-compose -f mesos.yml up -d
```

You can then manage the containers with the `docker-compose` command.

See Also

- Deploying Mesos in seven commands.
- Mesos frameworks

7.3 Looking at Docker Compose as a Replacement to Fig

Problem

You enjoy `Fig` but you would like to only use the Docker CLI to bring up your multi-container based applications.

Solution

Use Docker `compose`, a newly announced feature of Docker that brings the `fig` experience directly into the Docker CLI.

This recipe is now obsolete with the release of Docker compose. The scope and UX experience of the new `compose` is quite different from what is described in this release, but this may be interesting for developers.

Docker Compose is a new Docker feature announced during DockerCon Europe in December 2014 and released on February 26th 2015.

To ease testing, I prepared a Vagrantfile for you. It boots an Ubuntu 14.04 virtual machine, installs Docker but also grabs the test binaries from issue #9459. It replaces the Docker binary with this new one. Get started like so:

```
$ git clone https://github.com/how2dock/docbook.git
$ cd docbook/ch07/compose/
$ vagrant up
$ vagrant ssh
$ docker version
Client version: 1.3.2-dev
Client API version: 1.16
Go version (client): go1.3.3
Git commit (client): c6bf574
OS/Arch (client): linux/amd64
Server version: 1.3.2-dev
Server API version: 1.16
```

```
Go version (server): go1.3.3
Git commit (server): c6bf574
```

Note the development version of the Docker daemon and client are now installed. You can check the bootstrap script (`docker-bootstrap.sh`) to see how it was done.

With `compose` now installed, we are going to start a Wordpress blog similarly to what we did with Fig in ???. The only change is the structure of the YAML file describing the composition of the Wordpress setup (i.e two containers, one for `wordpress` and one for `mysql`) and the way we start the containers (i.e `docker up`)

The `fig.yml` file is replaced by a `group.yml` file that as the following structure:

```
name: blog

containers:
  wordpress:
    image: wordpress
    links:
     - mysql
    ports:
     - "80:80"
    environment:
     - WORDPRESS_DB_NAME=wordpress
     - WORDPRESS_DB_USER=wordpress
     - WORDPRESS_DB_PASSWORD=wordpresspwd
  mysql:
    image: mysql
    environment:
     - MYSQL_ROOT_PASSWORD=wordpressdocker
     - MYSQL_DATABASE=wordpress
     - MYSQL_USER=wordpress
     - MYSQL_PASSWORD=wordpresspwd
```

The group has a name `blog` and the two containers are now listed under the `containers` key. The `group.yml` file is located in `/vagrant` directory in the virtual machine started via Vagrant. Go to this directory and start the containers with `docker up`

```
$ cd /vagrant
$ ls
docker-bootstrap.sh  group.yml  README.md  Vagrantfile
$ docker up
...
mysql     | 2015-01-07 14:42:54 1 [Note] mysqld: ready for connections.
mysql     | Version: '5.6.22'  socket: '/var/run/mysqld/mysqld.sock'  port: 3306  MySQL Community
wordpress | Complete! WordPress has been successfully copied to /var/www/html
mysql     | 2015-01-07 14:43:00 1 [Warning] IP address '172.17.0.3' could not be resolved: Name or
wordpress | AH00558: apache2: Could not reliably determine the ...
wordpress | AH00558: apache2: Could not reliably determine the ...
wordpress | [Wed Jan 07 14:43:00.948827 2015] [mpm_prefork:notice] [pid 1] AH00163: Apache/2.4.10
wordpress | [Wed Jan 07 14:43:00.948951 2015] [core:notice] [pid 1] AH00094: Command line: 'apache
```

With the two containers up, open your browser at `http://192.168.33.10/` and you will see the Wordpress configuration page. Just like Fig, you can daemonize the provisioning:

```
$ docker up -d
Recreating mysql
Starting mysql
Recreating wordpress
Starting wordpress
vagrant@vagrant-ubuntu-trusty-64:/vagrant$ docker ps
CONTAINER ID  IMAGE            COMMAND                CREATED       STATUS        PORTS
685ac8b1a38d  wordpress:latest "/entrypoint.sh apac   3 seconds...  Up 2 seconds  0.0.0.0:80->80/t
f61bca3e3f31  mysql:latest     "/entrypoint.sh mysq   4 seconds...  Up 3 seconds  3306/tcp
```

Discussion

Docker Compose has been proposed on the Docker issues list. It was first described in issue 9175 then 9459. When the first official release of compose came out, issue 9694 was closed. Issue 9459 contains links to binaries as well as examples. This what this recipe is based on.

With this current implementation of Docker compose you can use the Docker CLI with some added functionality to perform queries/operations on groups. For instance you can check the containers that are running for a given group, or execute commands on a container defined by its name in the `group.yml` file (using the `:` prefix)

```
$ docker ps blog
CONTAINER ID    IMAGE            COMMAND                CREATED         STATUS        PORTS
685ac8b1a38d    wordpress:latest "/entrypoint.sh apac  4 minutes ago   Up 4 minutes  0.0.0.0:80->8
f61bca3e3f31    mysql:latest     "/entrypoint.sh mysq  4 minutes ago   Up 4 minutes  3306/tcp

$ docker exec -ti :wordpress /bin/bash
root@685ac8b1a38d:/var/www/html#
```

If you do not want to use the provided Vagrant setup at *https://github.com/how2dock/docbook*, you can clone the Docker repository, add the git remote which contains the development branch of Compose and build a new Docker binary from it.

```
$ git clone git@github.com:docker/docker
$ cd docker
$ git remote add aanand git@github.com:aanand/docker.git
$ git fetch --all
$ git checkout -b composition aanand/composition
```

7.4 Starting Containers on a Cluster with Docker Swarm

Problem

You know how to use Docker with a single host. You would like to start containers on a cluster of hosts while keeping the user experience of the Docker CLI you are accustomed to on a single machine.

Solution

Use Docker swarm.

Docker Swarm is a new Docker feature announced during DockerCon Europe in December 2014. The first beta release of Swarm was announced on February 26th 2015. Thus this recipes represents work in progress. The *Vagrantfile* provided is purely for testing purposes.

To ease testing of Docker *swarm*, I am providing a Vagrant setup and bootstrap scripts that setup a four nodes Swarm cluster. The cluster is composed of one head node and three compute nodes, all running Ubuntu 14.04. To get the cluster up, simply clone the git repository accompanying this book (if you have not done so already), head to the seventh chapter and the `swarm` sub-directory. Use Vagrant to boot the cluster. Like so:

```
$ git clone https://github.com/how2dock/docbook.git
$ cd docbook/ch07/swarm/
$ vagrant up
```

You should see four virtual machines being started by Vagrant. The machines will be bootstrapped via bash scripts defined in the Vagrantfile:

```
...
$bootstrap=<<SCRIPT
apt-get update
curl -sSL https://get.docker.com/ubuntu/ | sudo sh
gpasswd -a vagrant docker
echo "DOCKER_OPTS=\"-H tcp://0.0.0.0:2375\"" >> /etc/default/docker
service docker restart
SCRIPT

$swarm=<<SCRIPT
apt-get update
curl -sSL https://get.docker.com/ubuntu/ | sudo sh
gpasswd -a vagrant docker
docker pull swarm
SCRIPT
...
```

Once the nodes are up and Vagrant has returned, `ssh` to the head node and start a *swarm* container using the `swarm` image that was pulled during the bootstrap process.

```
$ vagrant ssh swarm-head
$ docker run -v /vagrant:/tmp/vagrant -p 1234:1234 -d swarm manage \
             file://tmp/vagrant/swarm-cluster.cfg -H=0.0.0.0:1234
72acd5bc00de0b411f025ef6f297353a1869a3cc8c36d687e1f28a2d8f422a06
```

The *Swarm* server setup shown above uses a file based discovery mechanism. The `swarm-cluster.cfg` file contains the hard coded lists of the *swarm* nodes started by Vagrant. Additional discovery services are available for Swarm. You can use a service hosted by Docker Inc, Consul, Etcd or Zookeeper. You can also write your own discovery interface.

With the *swarm* server running and the worker nodes discovered, you will be able to use the local `docker` client to get information about the cluster and start containers. You will need to use the `-H` option of the Docker CLI to target the Swarm server running in a container instead of the local docker daemon.

```
$ docker -H 0.0.0.0:1234 info
Containers: 0
Nodes: 3
 swarm-2: 192.168.33.12:2375
  └ Containers: 0
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 490 MiB
 swarm-3: 192.168.33.13:2375
  └ Containers: 0
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 490 MiB
 swarm-1: 192.168.33.11:2375
  └ Containers: 0
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 490 MiB
```

Using the local docker client and specifying the Swarm server as a Docker daemon endpoint, we can start containers on the entire cluster. For example let's start `nginx`.

```
$ docker -H 0.0.0.0:1234 run -d -p 80:80 nginx
8399c544b61953fd5610b01be787cb3802e2e54f220673b94d78160d0ee35b33
$ docker -H 0.0.0.0:1234 run -d -p 80:80 nginx
1b2c4634fc6d9f2c3fd63dd48a2580f466590ddff7405f889ada885746db3cbd
 docker -H 0.0.0.0:1234 ps
CONTAINER ID        IMAGE       COMMAND                ... PORTS                                NAM
1b2c4634fc6d        nginx:1.7   "nginx -g 'daemon of   ... 443/tcp, 192.168.33.11:80->80/tcp   swa
8399c544b619        nginx:1.7   "nginx -g 'daemon of   ... 443/tcp, 192.168.33.12:80->80/tcp   swa
```

We just started two `nginx` containers. Swarm scheduled them on two of the nodes in the cluster. You can open your browser at `http://192.168.33.11` and `http://192.168.33.12` and you will see the default `nginx` page.

The `docker run` command can take some time to return, Swarm needs to schedule the container on a node in the cluster and that node needs to pull the `nginx` image.

Discussion

In this setup the Docker `swarm` server is running within a local container on the Swarm head node. You can see it with `docker ps` and you can check the logs with `docker logs`. In the logs you see the requests made to start the `nginx` containers. It is interesting to see that we are using the Docker client on the Swarm head node to communicate with the local Docker daemon and the Swarm server running in a local container.

```
$ docker ps
CONTAINER ID  IMAGE         COMMAND               ...    PORTS                                 NAMES
72acd5bc00de  swarm:latest  swarm manage file:    ...    2375/tcp, 0.0.0.0:1234->1234/tcp   silly_b

$ docker logs 72acd5bc00de
time="2015-03-16T16:06:35Z" level=info msg="Listening for HTTP" addr="0.0.0.0:1234" proto=tcp
time="2015-03-16T16:06:56Z" level=info msg="HTTP request received" method=GET uri="/v1.17/info"
time="2015-03-16T16:21:13Z" level=info msg="HTTP request received" method=GET uri="/v1.17/containe
time="2015-03-16T16:21:27Z" level=info msg="HTTP request received" method=POST uri="/v1.17/contain
time="2015-03-16T16:27:13Z" level=info msg="HTTP request received" method=POST uri="/v1.17/contain
time="2015-03-16T16:30:48Z" level=info msg="HTTP request received" method=GET uri="/v1.17/containe
time="2015-03-16T16:30:52Z" level=info msg="HTTP request received" method=POST uri="/v1.17/contain
time="2015-03-16T16:36:45Z" level=info msg="HTTP request received" method=POST uri="/v1.17/contain
time="2015-03-16T16:45:11Z" level=info msg="HTTP request received" method=GET uri="/v1.17/containe
```

In these logs we clearly see the API calls being made to the Swarm server to launch the `nginx` containers in the cluster.

While Swarm is currently a separate release of Docker, heavy development on it is to be expected. Docker Inc. and Mesosphere announced a partnership to include Apache Mesos as a scheduler in Swarm. This will likely take the form of an Apache Mesos framework for Swarm. In addition, with development of Docker machine, we could see a merge of `machine` and `swarm` where creation of a `swarm` cluster in the Cloud is fully automated (see Recipe 7.5).

7.5 Using Docker Machine to Create a Swarm Cluster Across Cloud Providers

Problem

You understand how to create a *Swarm* cluster manually (see Recipe 7.4), but you would like to create one with nodes in multiple public Cloud Providers and keep the UX experience of the local Docker CLI.

Solution

Use *Docker Machine* to start Docker hosts in several Cloud providers and bootstrap them automatically to create a *swarm* cluster.

This is an experimental feature in *Docker Machine* and is subject to change.

The first thing to do is to obtain a *swarm* discovery token. This will be used during the bootstrapping process when starting the nodes of the cluster. As explained in Recipe 7.4, *swarm* features multiple discovery process. In this recipe, we used the service hosted by Docker, Inc. A discovery token is obtained by running a container based on the *swarm* image and running the `create` command. Assuming we do not have access to a Docker host already, we use `docker-machine` to create one solely for this purpose.

```
$ ./docker-machine create -d virtualbox local
INFO[0000] Creating SSH key...
...
INFO[0042] To point your Docker client at it, run this in your shell: $(docker-machine env local)
$ $(docker-machine env local)
$ docker run swarm create
31e61710169a7d3568502b0e9fb09d66
```

With the token in hand, we can use `docker-machine` and multiple public Cloud drivers to start worker nodes. We can start a *swarm* head node on VirtualBox, a worker on DigitalOcean (see Figure 1-6) and another one on Azure (see Recipe 8.5).

Do not start a *swarm* head in a public cloud and a worker on your localhost with VirtualBox. Chances are the head will not be able to route network traffic to your local worker node. It is possible to do, but you would have to open ports on your local router.

```
$ docker-machine create -d virtualbox --swarm --swarm-master --swarm-discovery token://31e6171016
INFO[0000] Creating SSH key...
...
INFO[0069] To point your Docker client at it, run this in your shell: $(docker-machine env head)
$ docker-machine create -d digitalocean --swarm --swarm-discovery token://31e61710169a7d3568502b0e
...
$ docker-machine create -d azure --swarm --swarm-discovery token://31e61710169a7d3568502b0e9fb09d6
...
```

Your *swarm* cluster is now ready. Your *swarm* head node is running locally in a Virtualbox VM, one worker node is running in DigitalOcean and another one in Azure. You can set the local `docker-machine` binary to use the head node running in VirtualBox and start using the *swarm* subcommands:

```
$ $(docker-machine env --swarm head)
$ docker info
Containers: 4
Nodes: 3
 head: 192.168.99.103:2376
  └ Containers: 2
  └ Reserved CPUs: 0 / 4
  └ Reserved Memory: 0 B / 999.9 MiB
 worker-00: 45.55.160.223:2376
  └ Containers: 1
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 490 MiB
 swarm-worker-01: swarm-worker-01.cloudapp.net:2376
  └ Containers: 1
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.639 GiB
```

Discussion

If you start a container, *swarm* will schedule it in round-robin fashion on the cluster. For example, starting three `nginx` container in a for loop with:

```
$ for i in `seq 1 3`;do docker run -d -p 80:80 nginx;done
```

Will lead to three `nginx` container on the three nodes in your cluster. Remember that you will need to open port 80 on the instances running in the Cloud to access the container (e.g see Recipe 8.5).

```
$ docker ps
CONTAINER ID    IMAGE       COMMAND                ... PORTS                                NAMES
9bff07d8ee18    nginx:1.7   "nginx -g 'daemon of   ... 443/tcp, 104.210.33.180:80->80/tcp   swarm-
457ed59c9bb3    nginx:1.7   "nginx -g 'daemon of   ... 443/tcp, 45.55.160.223:80->80/tcp    worker
6013be18cdbc    nginx:1.7   "nginx -g 'daemon of   ... 443/tcp, 192.168.99.103:80->80/tcp   head/c
```

Do not forget to remove the machine you started in the Cloud.

See Also

- Using Docker machine with Docker swarm.

7.6 Managing Containers through Docker UI

Problem

You have access to a Docker host and know how to manage images and containers, but you would like to use a simple web interface.

Solution

Use the Docker UI. While you can create your own image from source, the Docker UI is also available on Docker Hub which makes it straightforward to run it in a container.

On your Docker host start the Docker UI container:

```
$ docker run -d -p 9000:9000
               --privileged
               -v /var/run/docker.sock:/var/run/docker.sock
               dockerui/dockerui
```

You can then open your browser at `http://<IP_OF_DOCKER_HOST>:9000` and you will have access to the UI. An example is shown below.

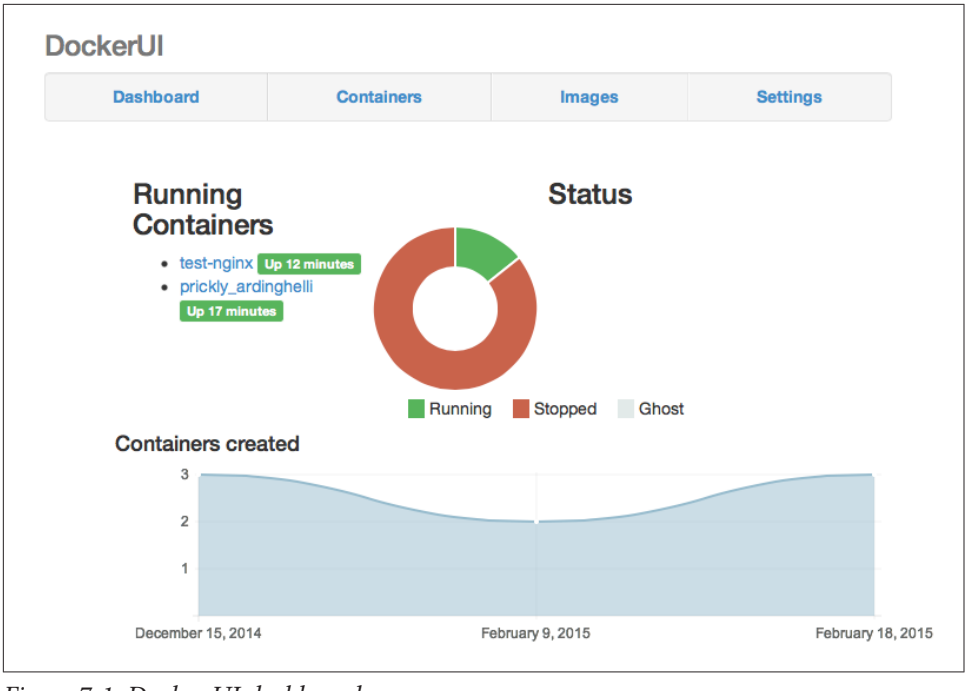


Figure 7-1. Docker UI dashboard

The Docker UI is not part of the official Docker release and is a community maintained project.

Discussion

Once you have access to the UI you can start a container. Go to the *Images* tab, select an image and click on the *create* button. You will be able to specify all the container startup options through the UI. Do not miss the *HostConfig* options where you can set port mappings. The screenshot below shows you a preview of this screen.

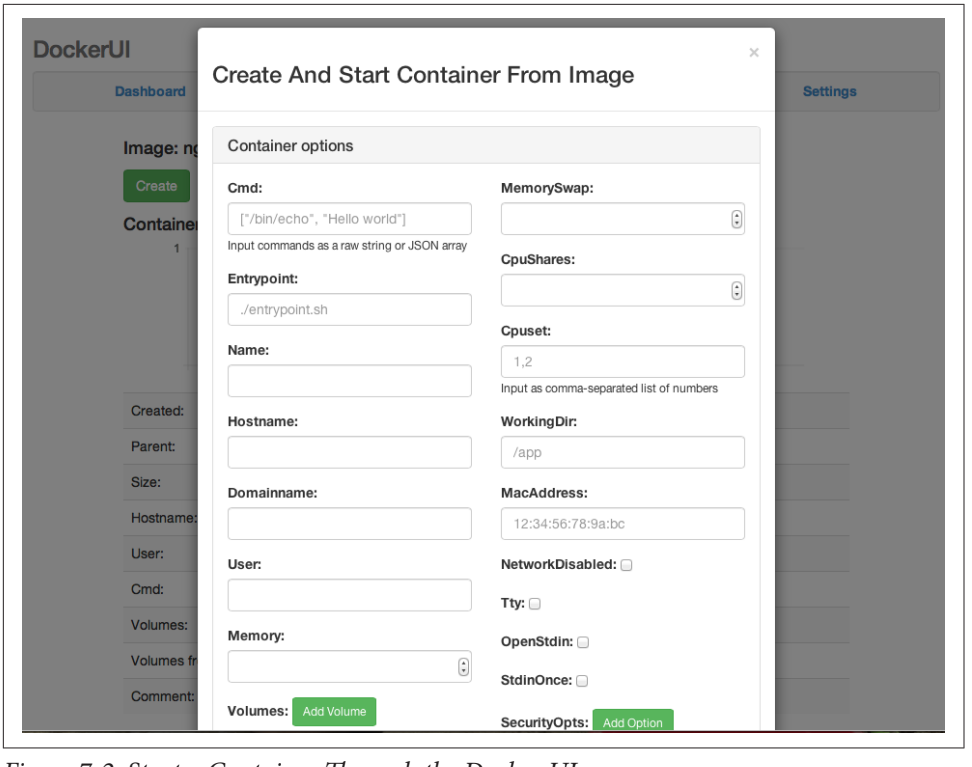


Figure 7-2. Start a Container Through the Docker UI

See Also

- Docker UI wiki which contains additional documentation.

7.7 Orchestrating Containers with Ansible Docker Module

Problem

You have developed some expertise with Ansible to configure your servers and orchestrate application deployment. You would like to take advantage of this expertise and use Ansible to manage Docker containers.

Solution

Use the Ansible Docker module. This module is part of the Ansible core, therefore after installing Ansible no additional packages need to be installed.

Ansible will run from your local machine, connect over SSH to your Docker hosts and use the `docker-py` API client to issue calls to the Docker daemon.

For example to start a `nginx` container in detached mode with a port mapping, you would write an Ansible playbook like this:

```
- hosts: nginx
  tasks:
  - name: Run nginx container
    docker: image=nginx:latest detach=true ports=80:80
```

Discussion about how to use Ansible are beyond the scope of this recipe. See the Ansible documentation.

Discussion

To get you up and running with the Ansible Docker module you can use the Vagrantfile accompanying this recipe. This will start a virtual machine acting as a Docker Host with the `docker-py` client installed. Two playbooks, an inventory file and some Ansible configurations are also available to make it turn key.

The first thing will be to install Ansible on your local machine:

```
$ sudo pip install ansible
```

Then to test the `nginx` playbook follow the instructions below:

```
$ git clone https://github.com/how2dock/docbook.git
$ cd ch07/ansible
$ tree
.
├── README.md
├── Vagrantfile
├── ansible.cfg
├── dock.yml
├── inventory
├── solo
│   ├── Vagrantfile
│   └── dock.yml
└── wordpress.yml
$ vagrant up
```

The `nginx` playbook shown in the *solution* section above is in the `dock.yml` file. To start this container using Ansible simply run the playbook. Once it finishes open your browser at `http://192.168.33.10` and you will see the welcome screen of Nginx. You can also connect to the VM with `vagrant ssh` and check the running container with the usual `docker ps` command.

```
$ ansible-playbook -u vagrant dock.yml
PLAY [nginx] ********************************************************************

GATHERING FACTS *****************************************************************
ok: [192.168.33.10]

TASK: [Run nginx container] *****************************************************
changed: [192.168.33.10]

PLAY RECAP **********************************************************************
192.168.33.10              : ok=2    changed=1    unreachable=0    failed=0
```

You can kill this `nginx` container with `docker kill` within the virtual machine or run a playbook that sets the state of the container to *killed*:

```
- hosts: nginx
  tasks:
  - name: Kill nginx container
    docker: image=nginx:latest detach=true ports=80:80 state=killed
```

If you want to try a little more complex example, check the Wordpress playbook `wordpress.yml`. We have deployed Wordpress several times already (see ??? or Figure 1-9). Run the playbook and open your browser at `http://192.168.33.10` and enjoy Wordpress once again. (You will need to have killed any container using port 80 on the host, otherwise you will get a port conflict error).

```
$ ansible-playbook -u vagrant wordpress.yml

PLAY [wordpress] ****************************************************************

GATHERING FACTS *****************************************************************
ok: [192.168.33.10]

TASK: [Docker pull mysql] *******************************************************
changed: [192.168.33.10]

TASK: [Docker pull wordpress] ***************************************************
changed: [192.168.33.10]

TASK: [Run mysql container] *****************************************************
ok: [192.168.33.10]

TASK: [Run wordpress container] *************************************************
changed: [192.168.33.10]

PLAY RECAP **********************************************************************
192.168.33.10              : ok=5    changed=3    unreachable=0    failed=0
```

Since Ansible playbook are written in YAML, you will notice some similarities with the `fig.yml` file in ???>

```
- hosts: wordpress
  tasks:

  - name: Docker pull mysql
    command: docker pull mysql:latest

  - name: Docker pull wordpress
    command: docker pull wordpress:latest

  - name: Run mysql container
    docker:
      name=mysql
      image=mysql
      detach=true
      env="MYSQL_ROOT_PASSWORD=wordpressdocker,MYSQL_DATABASE=wordpress, \
          MYSQL_USER=wordpress,MYSQL_PASSWORD=wordpresspwd"

  - name: Run wordpress container
    docker:
      image=wordpress
      env="WORDPRESS_DB_NAME=wordpress,WORDPRESS_DB_USER=wordpress, \
           WORDPRESS_DB_PASSWORD=wordpresspwd"
      ports="80:80"
      detach=true
      links="mysql:mysql"
```

We have run the playbooks directly from the local machine, however Vagrant has an Ansible provisioner. This means that you can run the playbook when the VM is started. Go to `ch07/ansible/solo` and `vagrant up`. The `nginx` container will automatically start.

See Also

- Ansible Up and Running book has a section on the Docker module.

7.8 Using Clocker

Problem

Solution

Discussion

7.9 Using Deis

Problem

Solution

Install deisctl

```
$ curl -sSL http://deis.io/deisctl/install.sh | sh -s 1.4.1
```

```
$ git clone https://github.com/deis/deis.git
$ cd deis
$ make discovery-url
$ vagrant up
$ ssh-add ~/.vagrant.d/insecure_private_key
$ export DEISCTL_TUNNEL=172.17.8.100
$ deisctl config platform set sshPrivateKey=$(HOME)/.vagrant.d/insecure_private_key
$ deisctl config platform set sshPrivateKey=$HOME/.vagrant.d/insecure_private_key
$ deisctl config platform set domain=local3.deisapp.com
$ deisctl install platform
```

Discussion

7.10 Using Rancher to Manage Containers on a Cluster of Docker Hosts

Problem

You want to manage containers in production through a system that supports multi-host, an overlay network that allows containers to reach each other without complex port forwarding rules, group management and a powerful dashboard.

Solution

Consider Rancher from Rancher Labs, the makers of Rancher OS (see Recipe 6.6). It is straightforward to setup with a management server running as a container and worker agent running as a container as well.

To ease up testing Rancher and see if it suits your needs, clone the project repository on GitHub and start a virtual machine locally through Vagrant, as shown below:

```
$ git clone https://github.com/rancherio/rancher.git
$ cd rancher
$ vagrant up
```

The virtual machine started is based on CoreOS (see Recipe 6.1) but you could use any other OS that runs Docker. The Vagrantfile contains two provisioning steps that install the management server and the worker agent from Docker images. You can use these commands almost identically to start Rancher on your own machines.

Once in the Rancher dashboard, if you navigate to the *Add Host* button, you will be presented with the exact Docker command to run on another host to join this Rancher deployment.

```
$ docker run -d -p 8080:8080 rancher/server:latest
$ docker run -e CATTLE_AGENT_IP=172.17.8.100 --privileged -e WAIT=true \
             -v /var/run/docker.sock:/var/run/docker.sock \
                        rancher/agent:latest http://localhost:8080
```

Once the Vagrant machine is up and that the rancher images have been downloaded, two containers will start and you will be able to access the Rancher dashboard at *http://localhost:8080*.

If you already have a server running on port 8080 in your local machine, Vagrant will pick a different port to server the Rancher UI on. You can always access it using the host only network at *http://172.17.8.100:8080*

The dashboard will show only one host and no running containers, by clicking on `Add Container` you will be redirected to a page where you can set the container run parameters, see the snapshot below. You can expand the *Advanced Options* area to set parameters like environment variables, volumes, networking and capabilities of the containers (e.g memory, privileged mode). By default the networking will be a so-called managed network, which will use a network overlay. You can still use the default Docker networking.

Considering the changes that will happen with Docker networking, this recipe will not expand on the Rancher overlay itself. See Chapter 3 for more information.

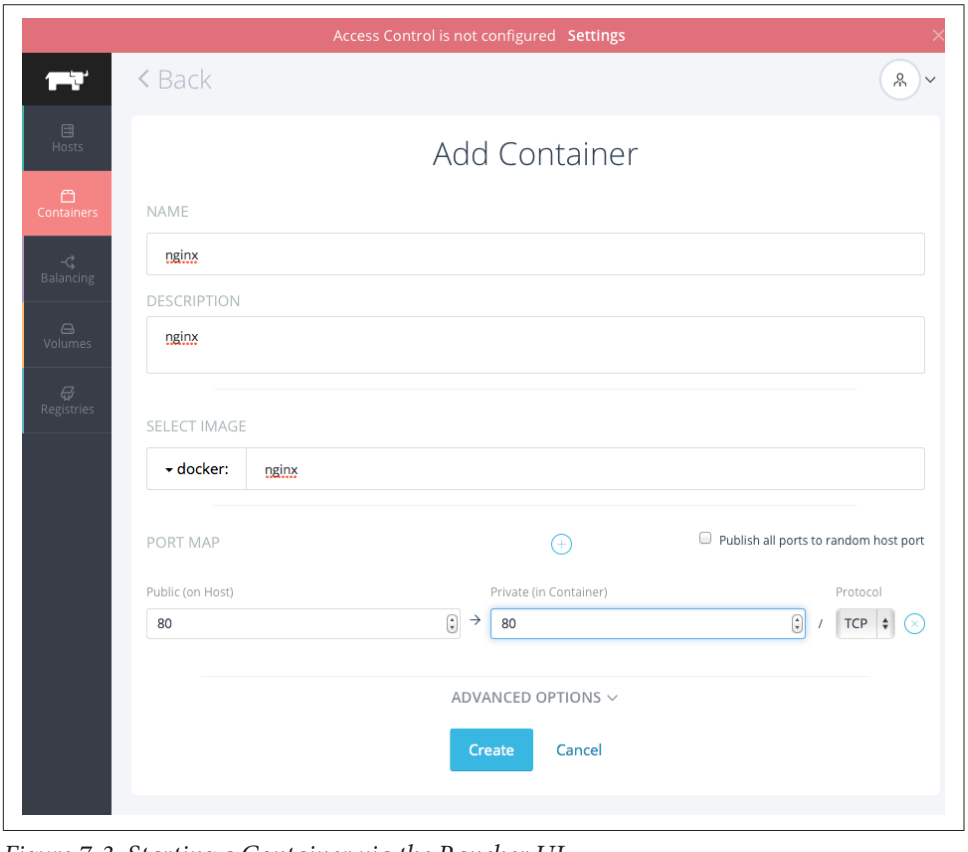


Figure 7-3. Starting a Container via the Rancher UI

Rancher will build a network overlay even though in this case we are using a single host and start the container within the IP range of the overlay. If you map the exposed port of the container to a port on the host, you will be able to access it through your browser. For example, if we start `nginx` and map it to port 80 of the host, you will enjoy the welcome screen of `nginx`. The container creation screen looks like the snapshot below:

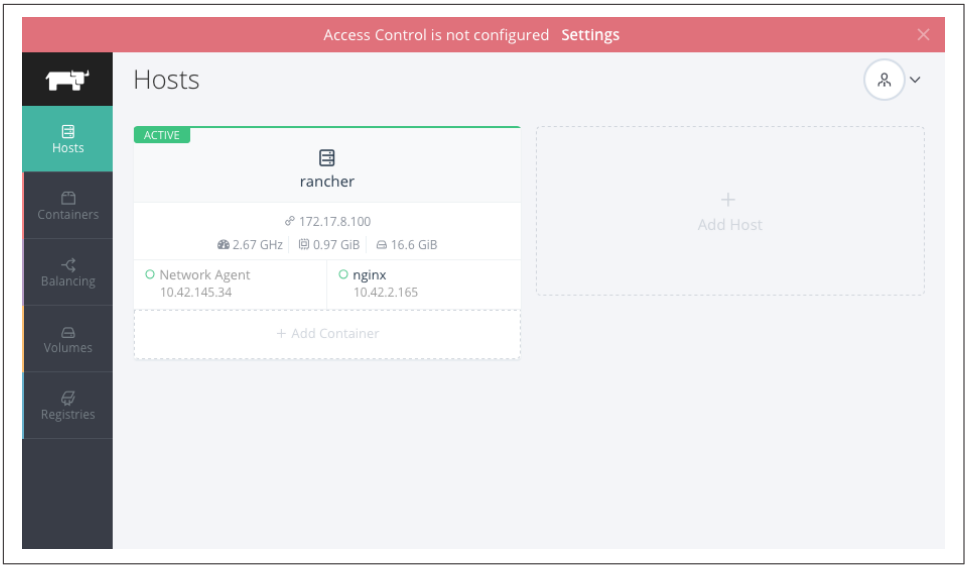


Figure 7-4. Rancher Dashboard with Running Container

At this stage, you have a working Rancher testbed. You can explore the dashboard. The container tab will list all your running containers. You will be able to open a shell into a container, start/stop it, . The volumes tab will list the volumes currently being used, volume manipulation through the dashboard is limited at this time. Finally you will also be able to define an existing private registry, or define load-balancers.

Discussion

While we did everything so far using the dashboard, Rancher also exposes a REST API to manage all its resources. To use the API you will need to generate a set of API access and secret keys. This is done by clicking on the user icon on the top right of the dashboard and selecting the *API & Keys* option. The API is not documented on th GitHub page, but the dashboard offers a nice API explorer.

You can manage a running container through the dashboard, by clicking it you will see an option to *View in API*. This will redirect you the the API explorer. This explorer features the json object describing the container as well as a set of *Actions* that can be performed (Green boxesin the UI). Selecting one of the Action will open a new window that will show you the API request that you can make. This is a perfect way to learn the API and possible write your own client. Below is a snapshot of a request to stop a container.

Figure 7-5. Rancher API Request Example

7.11 Running Containers Via Apache Mesos and Marathon

Problem

You are looking for a cluster scheduler to launch containers on a Docker hosts in your data-center. You might also already be running Apache Mesos to schedule long running jobs, cron jobs or even Hadoop or parallel computing workloads and would like to use it to run containers.

Solution

Use Apache Mesos and the Docker containerizer. Mesos is a cluster resource allocator that leverages multiple scheduling frameworks to maximize utilization of your data-center resources. Mesos is used by large companies like Ebay, Twitter, Netflix or AirBnB and more.

The Mesos architecture is based on one or several master nodes, worked nodes (or Mesos slaves), one or more scheduling frameworks that are deployed in Mesos and a service discovery system that uses Zookeeper. In Recipe 7.2, we already saw how to use Docker compose to start a testing Mesos infrastructure on a single node.

Marathon is one of the Mesos frameworks that can allow you to run tasks on a Mesos cluster. Mesos supports Docker (i.e Docker containerizer). This means that you can launch Marathon tasks that are made of Docker containers.

Amazon ECS service (see Recipe 8.12) can also use Mesos to schedule containers on AWS. Docker Swarm (see Recipe 7.4) is also scheduled to add support for Mesos based scheduling.

This recipe uses Mesos Playa a Mesos sandbox to show you how to run Docker containers with Mesos.

To get started, clone the `playa-mesos` repository from GitHub, start the virtual machine via Vagrant and `ssh` to it.

```
$ git clone https://github.com/mesosphere/playa-mesos.git
$ vagrant up
$ vagrant ssh
```

Once the machine is up you can access the Mesos web interface at `http://10.141.141.10:5050` and the Marathon web interface at `http://10.141.141.10:8080`.

Discussion on how to use Mesos and Marathon are beyond the scope of this book. Refer to the two websites for more information.

Marathon exposes a REST API that you can use to start tasks. Tasks are defined in JSON file and can be submitted to the API endpoint via `curl`. Here is an example task describe in JSON.

```
{
  "id": "http",
  "cmd": "python -m SimpleHTTPServer $PORT0",
  "mem": 50,
  "cpus": 0.1,
  "instances": 1,
  "constraints": [
    ["hostname", "UNIQUE"]
  ],
  "ports": [0]
}
```

The `id` is the name of the task (also called Application in Marathon). The `cmd` is what the application will run. Here a simple HTTP server via Python. What is important to

note is the use of `ports` which is set to a list containing `0`. This means that Marathon will dynamically allocate a port that this application will use. This dynamic port is passed to the `cmd` argument as `$PORT0`.

Save this JSON description in a file called *test.json* and submit this application via `curl` like so:

```
$ curl -is -H "Content-Type: application/json"
           -d @test.json 10.141.141.10:8080/v2/apps
HTTP/1.1 201 Created
...
```

Once the application starts, you will see it in the UI and be able to access the URL that points to the HTTP server that was just started. Note the port that was dynamically allocated.

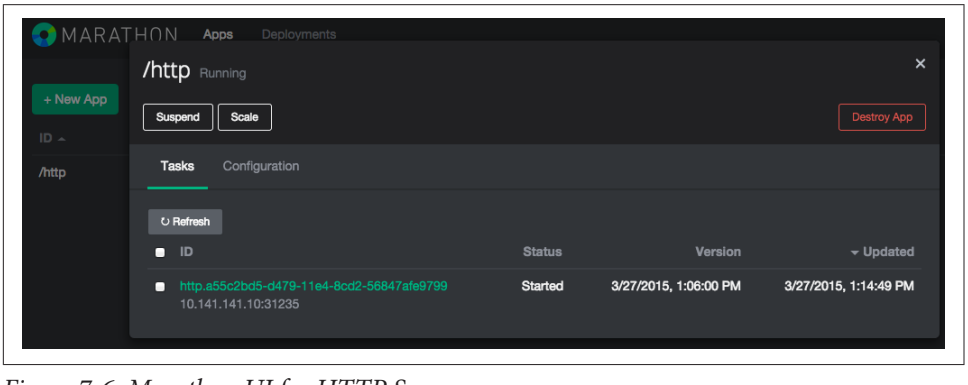


Figure 7-6. Marathon UI for HTTP Server

Let's now move on to starting an application that is made of a Docker container. By default the VM started by Playa Mesos will contain Docker but the Mesos slave is not configured to use it. Therefore we need to do a few configurations and restart `mesos-salve`. In the virtual machine do the following:

```
vagrant@mesos:~$ sudo su
root@mesos:/home/vagrant# cd /etc/mesos-slave
root@mesos:/etc/mesos-slave# echo 'docker,mesos' > containerizers
root@mesos:/etc/mesos-salve# echo '5mins' > executor_registration_timeout
root@mesos:/etc/mesos-slave# service mesos-slave restart
mesos-slave stop/waiting
mesos-slave start/running, process 2581
```

Create the following JSON file (e.g `docker.json`) that describes running an `nginx` container with a dynamic port allocation on the host.

```
{
  "container": {
    "type": "DOCKER",
    "docker": {
```

```
      "image": "nginx",
      "network": "BRIDGE",
      "portMappings": [
        { "containerPort": 80, "hostPort": 0 }
      ]
    }
  },
  "id": "nginx",
  "instances": 1,
  "cpus": 0.5,
  "mem": 512
}
```

Create this application via the Marathon API using `curl` and check the list of running applications:

```
$ curl -si -H 'Content-Type: application/json'
           -d @docker.json 10.141.141.10:8080/v2/apps
$ curl -sX GET -H "Content-Type: application/json" 10.141.141.10:8080/v2/tasks
           | python -m json.tool

{
   "tasks": [
       {
           "appId": "/nginx",
           "host": "10.141.141.10",
           "id": "nginx.404b7376-d47b-11e4-8cd2-56847afe9799",
           "ports": [
               31236
           ],
           "servicePorts": [
               10001
           ],
           "stagedAt": "2015-03-27T12:17:35.285Z",
           "startedAt": null,
           "version": "2015-03-27T12:17:29.312Z"
       },
       {
           "appId": "/http",
           "host": "10.141.141.10",
           "id": "http.a55c2bd5-d479-11e4-8cd2-56847afe9799",
           "ports": [
               31235
           ],
           "servicePorts": [
               10000
           ],
           "stagedAt": "2015-03-27T12:06:05.873Z",
           "startedAt": "2015-03-27T12:14:49.986Z",
           "version": "2015-03-27T12:06:00.485Z"
       }
   ]
}
```

You see the `http` application that we started earlier. And you also see the new `nginx` application which uses Docker. The application will take a little bit of time to deploy, just enough time to `docker pull nginx`. To take into account the time it may take to download an image from a registry we defined the `executor_registration_timeout` before restarting the `mesos-slave`. Marathon also allocated a port dynamically to bind port 80 of the `nginx` container to the host, and in this case it chose 31236. If you open your browser at `http://10.141.141.10:31236` you will see the familiar web page of Nginx.

Discussion

The Docker application definition specified in JSON format can contain volume mounts, it can specify arguments that will overwrite the CMD arguments defined in a Dockerfile, it can specify `docker run` parameters and can also run in privileged mode. The Docker containerizer documentation has more detailed information. But as a quick reference you could also define an application with all those extra functionalities like so:

```
{
    "id": "privileged-job",
    "container": {
        "docker": {
            "image": "mesosphere/inky"
            "privileged": true,
            "parameters": [
                { "key": "hostname", "value": "a.corp.org" },
                { "key": "volumes-from", "value": "another-container" },
                { "key": "lxc-conf", "value": "..." }
            ]
        },
        "type": "DOCKER",
        "volumes": []
    },
    "args": ["hello"],
    "cpus": 0.2,
    "mem": 32.0,
    "instances": 1
}
```

Finally, running Mesos on a single host defeats the purpose of this recipe, and you will want to create a Mesos cluster with the Docker containerizer enabled on all Mesos slaves.

See Also

- Mesosphere documentation for the Docker *containerizer*.
- Marathon example JSON files.

- Original post this recipe is based on.

7.12 Using the Mesos Docker Containerizer on a Mesos Cluster

Problem

In Recipe 7.11 you saw how to test the Mesos Docker containerizer to run containers on a Mesos sandbox. You would like to do the same but on a Mesos cluster.

Solution

Build a Mesos cluster using containers and the images prepared by Mesosphere on the Docker hub. Configure the Mesos slave to use the Docker containerizer.

To ease testing this recipe use the on-line material accompanying this book. For this recipe we are going to use a Vagrantfile that sets up a local three nodes Mesos cluster and uses Ansible to start the container that run the Mesos software components (i.e Zookeeper, Mesos master, Marathon framework and Mesos slave.)

Clone the repository if you have not done so already, head to `ch07/mesos` and use Vagrant to bring up the three node cluster:

If you have enough memory on your local machine, you can add more nodes to this setup or change the memory allocated to each node (see the `Vagrantfile`)

```
$ git clone https://github.com/how2dock/docbook.git
$ cd dockbook/ch07/mesos
$ vagrant up
$ vagrant status
Current machine states:

mesos-head                running (virtualbox)
mesos-1                   running (virtualbox)
mesos-2                   running (virtualbox)
```

If you have followed along recipe by recipe, you will have read Recipe 7.7. If not, read that recipe first to configure Ansible on your machine. We will use a Ansible playbook to start a few containers on the VM. The playbook is `mesos.yml`. To start all containers simply run the playbook:

```
$ ansible-playbook -u vagrant mesos.yml
```

Once the play is done, the Mesos head node will have three containers running (i.e Zookeeper, Mesos Master and the Marathon framework). The two slaves will have one container running (i.e the Mesos slave). All images come from Docker hub

Open your browser at *http://192.168.33.10:5050* to access the Mesos UI, then open your browser at *http://192.168.33.10:8080* to access the Marathon UI.

To start a `nginx` container in that Mesos cluster, create a Mesos application in the Marathon framework using the API.

```
$ curl -si -H 'Content-Type: application/json' -d @docker.json 192.168.33.10:8080/v2/apps
```

Once the image has been downloaded you will be able to access the `nginx` welcome screen in a similar fashion than described in Recipe 7.11.

The `docker.json` application definition specifies 128 MB of RAM. If your slaves do not have enough memory the application could be stuck in *deploying* stage. Make sure that your slave have enough RAM of reduce the memory constraint of your application.

Discussion

The inventory used by Ansible is harcoded in the `inventory` file. If you change the IP address of the nodes or add more nodes, make sure to update the inventory as well.

The current play executes `docker run` commands remotely over SSH. If you want to use the Ansible Docker module, comment the `command` tasks and uncomment the `docker` tasks.

You will notice that the Mesos slave actually runs as a container. When starting the container we pass the environment variable `MESOS_CONTAINERIZERS=docker,mesos` which configures the slave to use Docker. The slave will actually start other containers on the host itself. This is achieved by mounting `/var/run/docker.sock`, `/usr/bin/docker` and `/sys` from the host to the container. While it works in testing scenario, the Mesos containerizer is not made to do this. You should consider running the slave on the host themselves until Mesos development recommends running the slave in containers for production.

I am currently unable to obtain a workable cluster using the play that uses the docker module. If you find out why, don't hesitate to send me a pull request. Thanks.

See Also

- Apache Mesos configuration

7.13 Discovering Docker Services with Registrator

Problem

You are building a distributed applications with services based on containers started on multiple hosts. You need to automatically discover these services to configure your application. This is needed when services migrate from one host to another or when they are started automatically.

Solution

Use *registrator*. It runs as a container on the hosts in your system. By mounting the Docker socket `/var/run/docker.sock` it listens to containers that come and go and register or unregisters them on a data store backend. Currently several backend data store are available (e.g etcd, consul, skyDNS2) and can possibly support more. These service registries are not specific to Docker even though *etcd* comes bundled in the coreOS distribution (see Recipe 6.3)

To use *registrator*, you first need to setup one backend for service registries. Since there are available as static binary, you can just download them and run them in the foreground for testing. For example to use *etcd*:

```
$ curl -L  https://github.com/coreos/etcd/releases/download/v0.4.6/\
           etcd-v0.4.6-linux-amd64.tar.gz
       -o etcd-v0.4.6-linux-amd64.tar.gz
$ tar xzvf etcd-v0.4.6-linux-amd64.tar.gz
$ cd etcd-v0.4.6-linux-amd64
$ sudo ./etcd
2015/03/26 14:02:21 no data-dir provided, using default data-dir ./default.etcd
2015/03/26 14:02:21 etcd: listening for peers on http://localhost:2380
2015/03/26 14:02:21 etcd: listening for peers on http://localhost:7001
2015/03/26 14:02:21 etcd: listening for client requests on http://localhost:2379
2015/03/26 14:02:21 etcd: listening for client requests on http://localhost:4001
...
```

Leave `etcd` running and in another terminal session create a directory in the `etcd` key value-store (e.g *cookbook* below). This directory will hold the services when they are discovered:

```
$ cd etcd-v0.4.6-linux-amd64
$ ./etcdctl mkdir cookbook
$ ./etcdctl ls
/cookbook
```

Then download the `registrator` image from Docker hub and run it.

You define the registry service backend as an argument to the `gliderlabs/registrator` image. Do not forget the key that you directory name that you set just above.

```
$ docker pull gliderlabs/registrator
$ docker run -d -v /var/run/docker.sock:/tmp/docker.sock
               -h 192.168.33.10
               gliderlabs/registrator
               -ip 192.168.33.10
               etcd://192.168.33.10:4001/cookbook
```

Replace `192.168.33.10` above with the IP address of your own setup. In this particular example I ran everything on the same host. But you will most likely want to run an `etcd` cluster separate from your cluster of Docker hosts where you will run registrator.

You can now start any container, expose ports to the host and you will see the registration in your `etcd` key-value store

```
$ docker run -d -p 80:80 nginx
$ ./etcdctl ls /cookbook
/cookbook/nginx-80
$ ./etcdctl ls /cookbook/nginx-80
/cookbook/nginx-80/192.168.33.10:pensive_franklin:80
$ ./etcdctl get /cookbook/nginx-80/192.168.33.10:pensive_franklin:80
192.168.33.10:80
```

If you look at the logs of the `registrator` container you will see that it is listening to Docker events and registering the ports exposed to the host:

```
$ docker logs <CONTAINER_ID>
2015/03/26 ... registrator: Forcing host IP to 192.168.33.10
2015/03/26 ... registrator: Using etcd registry backend at \
                                etcd://192.168.33.10:4001/cookbook
2015/03/26 ... registrator: ignored: 6f8043d9973f no published ports
2015/03/26 ... registrator: Listening for Docker events...
2015/03/26 ... registrator: ignored 8c033ca03a82 port 443 not published on host
2015/03/26 ... registrator: added: 8c033ca03a82 192.168.33.10:pensive_franklin:80
```

In the logs above, `6f8043d9973f` is the container ID of the registrator container and `8c033ca03a82` is the container ID of the `nginx` container that we started.

Discussion

The naming convention for the keys stored in `etcd` is based on the *Service* object created by `registrator` and passed to the registry backend. From the GitHub repo, the *Service* structure is defined like this:

```
type Service struct {
    ID    string               // <hostname>:<container-name>:<internal-port>
                               //[:udp if udp]
    Name  string               // <basename(container-image)>
                               //[-<internal-port> if >1 published ports]
    Port  int                  // <host-port>
    IP    string               // <host-ip> || <resolve(hostname)> if 0.0.0.0
    Tags  []string             // empty, or includes 'udp' if udp
    Attrs map[string]string    // any remaining service metadata from environment
}
```

The key for the service is defined by:

```
<registry-uri-path>/<service-name>/<service-id>
```

In the example above it is then (see the ID definition in the *Service* object):

```
cookbook/nginx-80/192.168.33.10:pensive_franklin:80
```

And set to `<ip>:<port>` or in the example above `192.168.33.10:80` (see the Port and IP definitions in the *Service* object)

If you do not want to use `etcd` but rather use `consul`, you can switch registry backend. You can easily try this on a single host, by using the `progrium/consul` image from Docker hub. Pull the image and run the `consul` agent in one terminal session (the consul container is not detached in this example)

```
$ docker pull progrium/consul
$ docker run -p 8400:8400 -p 8500:8500 -p 8600:53/udp
             -h cookbook progrium/consul -server
             -bootstrap -ui-dir /ui
```

In another session start `registrator` but change the registry URI to `consul://192.168.33.10:8500/foobar`

```
$ docker run -d -v /var/run/docker.sock:/tmp/docker.sock
             -h 192.168.33.10 gliderlabs/registrator
             -ip 192.168.33.10 consul://192.168.33.10:8500/foobar
```

You can now start an `nginx` container:

```
$ docker run -d -p 80:80 nginx
```

And now, if you check the `consul` UI at *`http://192.168.33.10:8500/ui`* you will see that a `foobar` directory has been created with several keys in them. The keys for the consul container itself and the key for your `nginx` container. See the snapshot below:

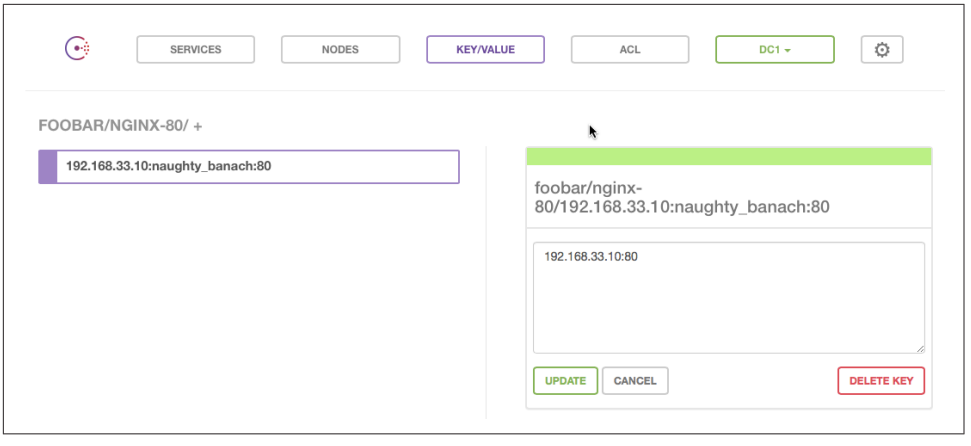


Figure 7-7. Consul UI

With Docker service registration under control you can start thinking about dynamically reconfiguring other services (see ???)

See Also

- GitHub repository of registrator
- Original Blog post from Jeff Lindsay

CHAPTER 8
Docker in the Cloud

This chapter will consist of several recipes focused on using Docker in Public and Private Clouds. The four recipes listed currently are only stubs, more will be added as the book nears completion. You can send me suggestions at *how2dock@gmail.com*

With the advent of pubic and private clouds, enterprises have moved an increasing number of workloads to the clouds. A significant portion of IT infrastructure is now provisioned on public Clouds like Amazon Web Services(AWS), Google Compute Engine(GCE) and Microsoft Azure(Azure). In addition, companies have deployed private clouds to provide a self-service infrastructure for IT needs.

While Docker, like any software, runs on bare metal servers, running a Docker host in a public or private Cloud (i.e on virtual machines) and orchestrating containers started on those hosts is going to be a critical part of new IT infrastructure needs.

In this chapter we look at the top three public Clouds(i.e AWS, GCE and Azure) and some of the Docker services they offer. We briefly review how to use CLI to start instances in the Cloud and install Docker, then look at some container optimized instances or linux distributions offered. While Docker machine (see Recipe 1.5) will ultimately remove the need to use these provider CLIs, learning how to start instances will be useful to use the other Docker related services. Indeed, we also cover two new services currently in preview: The Amazon Container Service and the Google container engine.

AWS, GCE and Azure are the recognized top three public Cloud providers in the world. However, Docker can be installed on any public cloud where you can run a linux distribution supported by Docker (e.g Ubuntu, CentOS, coreOS).

Docker machine as been covered in Recipe 1.5, once Docker 1.5 is released the recipe will be updated and I will problably move it to this chapter.

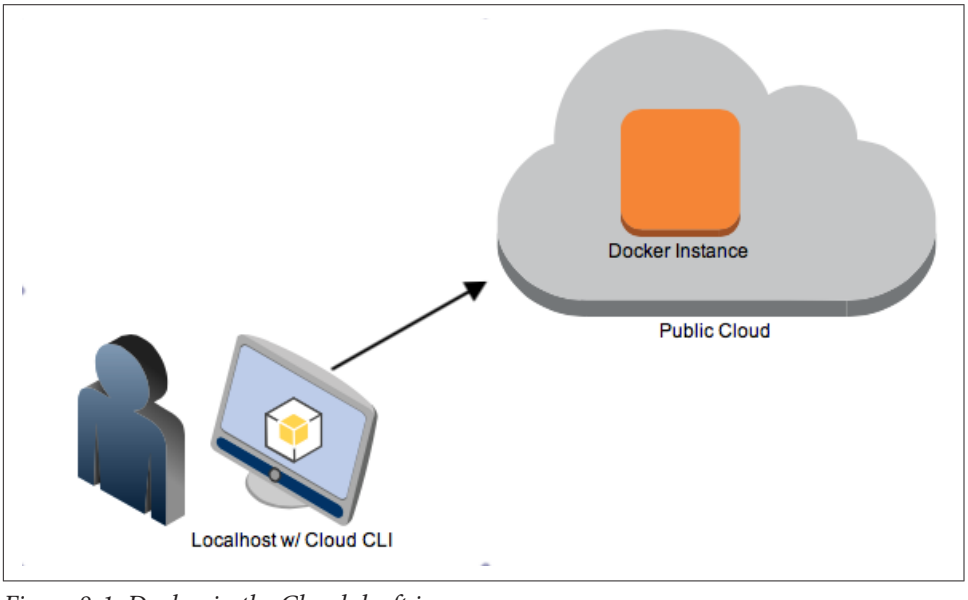


Figure 8-1. Docker in the Cloud draft image

8.1 Accessing Public Clouds to Run Docker

Problem

You need access to a Public Cloud to run Docker in cloud instances.

Solution

If you do not already have access, create an account on your public Cloud provider of choice.

- For GCE, you can start with a free trial. You will need a Google account. You will then be able to log into the console
- For Azure, you can start with a free trial.
- For AWS, you can have access to a free tier. Once you create an account, you can log into the console

Log into the web console of the provider that you want to use and go through the launch instance wizard. Make sure you can start an instance that you can connect to via `ssh`.

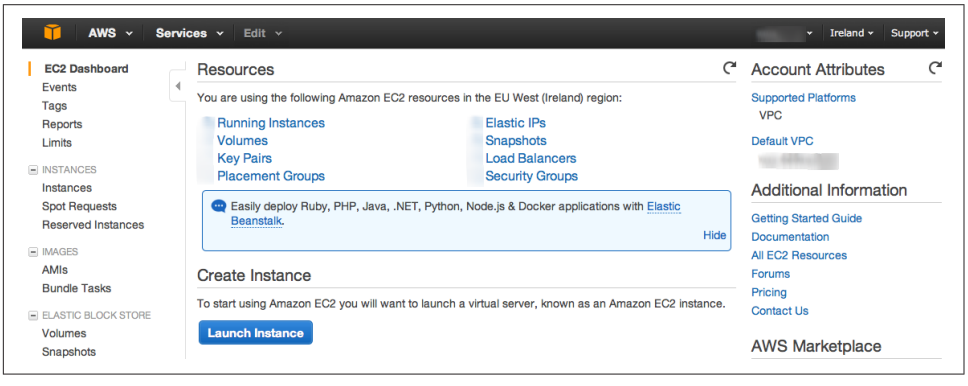


Figure 8-2. AWS Console

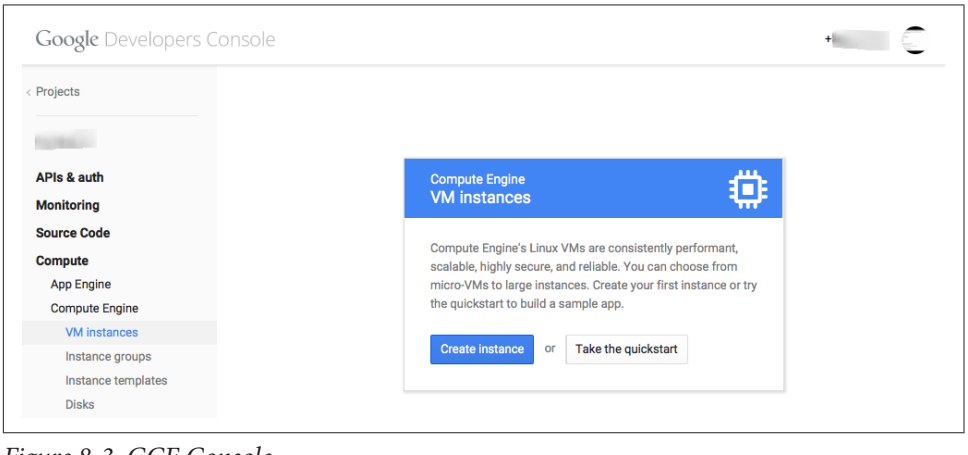


Figure 8-3. GCE Console

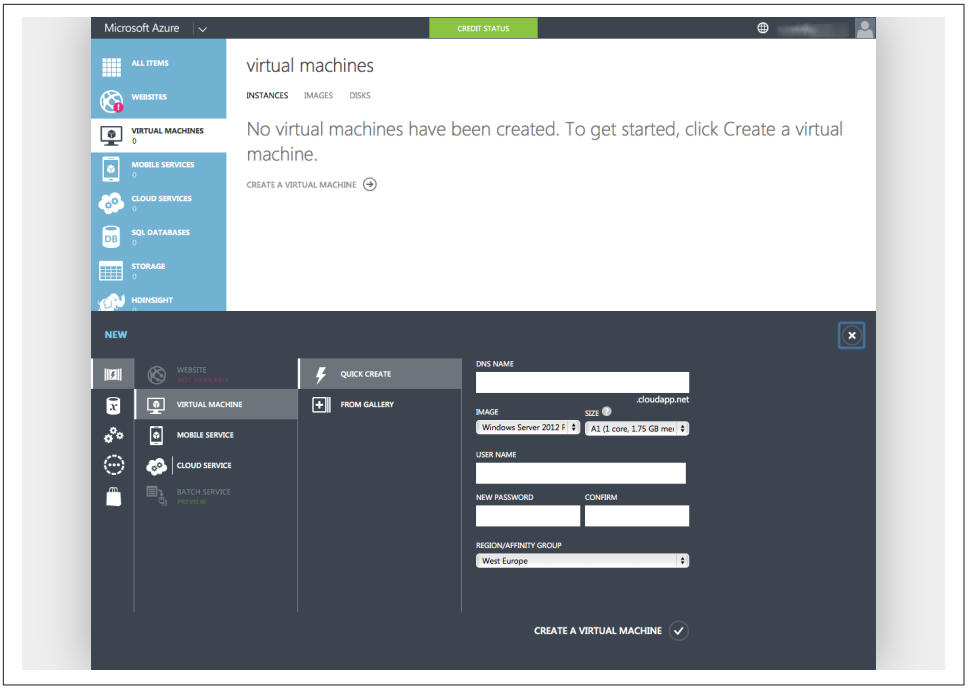


Figure 8-4. Azure Console

Discussion

If you are not familiar with one of these Clouds and have not completed this setup, you will not be able to follow the recipes in this chapter. However a complete step by step walkthrough of using these Clouds is beyond the scope of this cookbook.

These instructions are not Docker specific. Once you create an account on one of these clouds you will have access to any of the Cloud services provided.

On AWS, the recipes in this chapter will make use of the Elastic Compute Cloud(EC2) service. To start instances, you will need to become familiar with four basic principles:

- AWS API keys to use with the AWS command line interface (CLI).
- SSH Key pairs to connect to your instances via *ssh*.
- Security Groups to control traffic to and from EC2 instances.

- Instance user data to configure your instances at startup time.

On GCE, we will use the Google compute engine service. The AWS principles also apply to GCE:

- GCE authentication. In the rest of the chapter we will use the `gcloud` CLI, which uses OAuth2 for authentication. There are other types of authentication and authorization mechanisms for GCE.
- Using an SSH key to connect to an instance.
- Instance firewall
- Instance metadata

See Also

- Programming Amazon Web Services
- AWS getting started guide
- Automating Microsoft Azure Infrastructure Services
- GCE getting started guide

8.2 Starting a Docker Host on AWS EC2

Problem

You want to start a VM instance on AWS EC2 Cloud and use it as a Docker Host.

Solution

While you can start an instance and install Docker in it via the EC2 web console, we will use the AWS command line interface (CLI). Before you do that as mentioned in Recipe 8.1 you should obtain a set of API keys. In the web console, select your account name on the top right of the page and go to the *security credentials* page. You will be able to create a new access key. The secret key corresponding to this new access key will be given to you only once, so make sure that you store it securely.

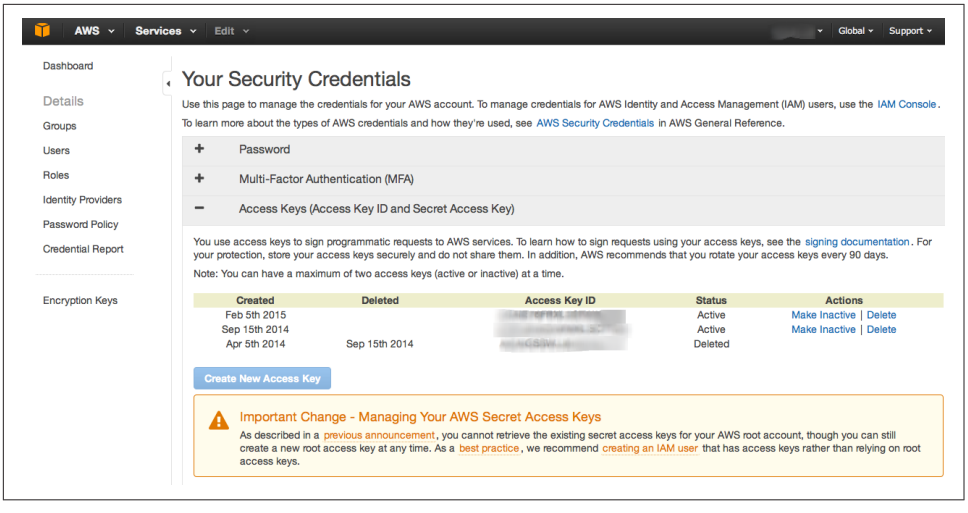


Figure 8-5. AWS Security Credentials Page

You can then install the AWS CLI and configure it to use your newly generated keys. Select an AWS region where you want to start your instances by default.

The AWS CLI, `aws` is a Python package that can be installed via the Python Package Index (i.e *pip*). For example on Ubuntu:

```
$ sudo apt-get -y install python-pip
$ sudo pip install awscli
$ aws configure
AWS Access Key ID [****************n-mg]: AKIAIEFDGHQRTW3MNQ
AWS Secret Access Key [****************UjEg]: b4pWYhMUosg976arg9869Qd+Yg1qo22wC
Default region name [eu-east-1]: eu-west-1
Default output format [table]:
$ aws --version
aws-cli/1.7.4 Python/2.7.6 Linux/3.13.0-32-generic
```

To access your instance via `ssh` you will need to have an SSH key pair setup in EC2. Create a key pair via the CLI, copy the returned private key into a file in your `~/.ssh` folder and make that file only readable and writable by you. Verify that the key has been created, either via the CLI or by checking the web console.

```
$ aws ec2 create-key-pair --key-name cookbook
$ vi ~/.ssh/id_rsa_cookbook
$ chmod 600 ~/.ssh/id_rsa_cookbook
$ aws ec2 describe-key-pairs
-----------------------------------------------------------------------------
|                            DescribeKeyPairs                               |
+---------------------------------------------------------------------------+
||                               KeyPairs                                  ||
|+-------------------------------------------------------------+-----------+|
||                      KeyFingerprint                         |  KeyName  ||
```

```
|+------------------------------------------------------------+-----------+|
||  69:aa:64:4b:72:50:ee:15:9a:da:71:4e:44:cd:db:c0:a1:72:38:36  |  cookbook ||
|+------------------------------------------------------------+-----------+|
```

You are now ready to start an instance on EC2. The standard linux images from AWS now contain a Docker repository. Hence when starting an EC2 instance from an Amazon Linux AMI we will be one step away from running Docker (i.e `sudo yum install docker`).

Use a para virtualized (i.e PV) Amazon Linux AMI, so that you can use a `t1.micro` instance type. In addition, the default security group allows you to connect via `ssh`, hence you do not need to create any additional rules in the security group if you only need to `ssh` to it.

```
$ aws ec2 run-instances --image-id ami-7b3db00c
                        --count 1
                        --instance-type t1.micro
                        --key-name cookbook
$ aws ec2 describe-instances
$ ssh -i ~/.ssh/id_rsa_cookbook ec2-user@54.194.31.39
The authenticity of host '54.194.31.39 (54.194.31.39)' can't be established.
RSA key fingerprint is 9b:10:32:10:ac:46:62:b4:7a:a5:94:7d:4b:2a:9f:61.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '54.194.31.39' (RSA) to the list of known hosts.

       __|  __|_  )
       _|  (     /   Amazon Linux AMI
      ___|\___|___|

https://aws.amazon.com/amazon-linux-ami/2014.09-release-notes/
[ec2-user@ip-172-31-8-174 ~]$
```

Install the Docker package, start the docker daemon and verify that the Docker cli is working.

```
[ec2-user@ip-172-31-8-174 ~]$ sudo yum install docker
[ec2-user@ip-172-31-8-174 ~]$ sudo service docker start
[ec2-user@ip-172-31-8-174 ~]$ sudo docker ps
CONTAINER ID    IMAGE       COMMAND     CREATED     STATUS      PORTS     NAMES
```

Do not forget to terminate the instance or you might get charged for it:

```
$ aws ec2 terminate-instances --instance-ids <instance id>
```

Discussion

While we spent some time in this recipe creating API access keys and installing the CLI. Hopefully you see the ease of creating Docker hosts in AWS. The standard AMI are now ready to go to install Docker in two commands.

The Amazon Linux AMI also contains *cloud-init* which has become the standard for configuring cloud instances at boot time. This allows you to pass so called *user data* at instance creation. Cloud init will parse the content of the user data and execute the commands. Using the AWS cli we can pass some user data to automatically install Docker. The small downside is that it needs to be base64 encoded.

Create a small bash script with the two commands that we did earlier:

```
#!/bin/bash
yum -y install docker
service docker start
```

Encode this script and pass it to the instance creation command:

```
$ udata="$(cat docker.sh | base64 )"
$ aws ec2 run-instances --image-id ami-7b3db00c
                        --count 1
                        --instance-type t1.micro
                        --key-name cookbook
                        --user-data $udata
$ ssh -i ~/.ssh/id_rsa_cookbook ec2-user@<public IP of the created instance>
$ sudo docker ps
CONTAINER ID    IMAGE       COMMAND     CREATED     STATUS      PORTS     NAMES
```

With Docker daemon running, if you wanted to access it remotely you would need to setup TLS access (see Recipe 4.10), and open port 2376 in your security group.

Using this CLI is not Docker specific. This CLI gives you access to the complete set of AWS APIs. However, using it to start instances and install Docker in them significantly streamlines the provisioning of Docker hosts.

See Also

- Installing the AWS CLI.
- Configuring the AWS CLI.
- Launching an instance via the AWS CLI.

8.3 Starting a Docker Host on Google GCE

Problem

You want to start a VM instance on Google GCE Cloud and use it as a Docker Host.

Solution

Install the `gcloud` cli (you will need to answer a few questions), then log into the Google cloud. If the CLI can open a browser you will be redirected to a web page and ask to sign in and accept the terms of use. If your terminal cannot launch a browser you will be given a URL to open in a browser. This will give you an access token to enter at the command prompt.

```
$ curl https://sdk.cloud.google.com | bash
$ gcloud auth login
Your browser has been opened to visit:
    https://accounts.google.com/o/oauth2/auth?redirect_uri=...
...
$ gcloud compute zones list
NAME           REGION       STATUS    NEXT_MAINTENANCE TURNDOWN_DATE
asia-east1-c   asia-east1   UP
asia-east1-a   asia-east1   UP
asia-east1-b   asia-east1   UP
europe-west1-b europe-west1 UP
europe-west1-c europe-west1 UP
us-central1-f  us-central1  UP
us-central1-b  us-central1  UP
us-central1-a  us-central1  UP
```

If you have not setup a project, set one up in the web console. Projects allow you to manage authorization or resources. It is the equivalent of Amazon IAM for the Google Cloud resources.

To start instances it is handy to set some defaults for the region and zone that you would prefer to use. (Even though deploying a robust system in the Cloud will involved instances in multiple regions and zones.) To do this use te `gcloud config set` command, for example:

```
$ gcloud config set compute/region europe-west1
$ gcloud config set compute/zone europe-west1-c
$ gcloud config list --all
```

To start an instance you will need an image name and an instance type. Then the `gcloud` tool will do the rest:

```
$ gcloud compute instances create cookbook \
                --machine-type n1-standard-1 \
                --image ubuntu-14-04 \
                --metadata startup-script="sudo apt-get -y install docker.io"
...
$ gcloud compute ssh cookbook
sebastiengoasguen@cookbook:~$ sudo docker ps
CONTAINER ID    IMAGE    COMMAND     CREATED     STATUS    PORTS    NAMES
...
$ gcloud compute instances delete cookbook
```

In the example above we created an Ubuntu 14.04 instance, of machine type `n1-standard-1` and we passed some metadata specifying that it was to be used as a startup script. The bash command specified installed the docker.io package from the standard Ubuntu repository. This led to a running instance with Docker running. The GCE metadata is relatively equivalent to the AWS EC2 user-data and is processed by cloud-init in the instance.

Discussion

If you list the images available in a zone you will see that some images are very interesting for Docker specific tasks.

```
$ gcloud compute images list
NAME                              PROJECT           ALIAS       DEPRECATED STATUS
...
coreos-alpha-584-0-0-v20150205    coreos-cloud                             READY
coreos-beta-557-2-0-v20150204     coreos-cloud                             READY
coreos-stable-522-6-0-v20150128   coreos-cloud      coreos                 READY
...
container-vm-v20141208            google-containers container-vm           READY
container-vm-v20150112            google-containers container-vm           READY
container-vm-v20150129            google-containers container-vm           READY
...
```

Indeed, GCE provides coreOS images, as well as so called container VMs. CoreOS is discussed in Chapter 6 chapter. Container VMs are Debian 7 based instances that contain the Docker daemon and the Kubernetes kubelet. Kubernetes is discussed in Chapter 5 and we will go in more details about the container VM in Recipe 8.8.

If you want to start a coreOS instance, you can use the image alias. You will not need to specify any metadata to install Docker:

```
$ gcloud compute instances create cookbook --machine-type n1-standard-1
                                           --image coreos
$ gcloud compute ssh cookbook
...
CoreOS (stable)
sebastiengoasguen@cookbook ~ $ docker ps
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS    PORTS    NAMES
```

Using the `gcloud` CLI is not Docker specific. This CLI gives you access to the complete set of GCE APIs. However, using it to start instances and install Docker in them significantly streamlines the provisioning of Docker hosts.

8.4 Starting a Docker Host on Microsoft Azure

Problem

You want to start a VM instance on Microsoft Azure Cloud and use it as a Docker Host.

Solution

First you will need an account on Azure (see Figure 8-1). If you do not want to use the Azure portal you will need to install the Azure CLI. On a fresh Ubuntu 14.04 machine you would do:

```
$ sudo apt-get update
$ sudo apt-get -y install nodejs-legacy
$ sudo apt-get -y install npm
$ sudo npm install -g azure-cli
$ azure -v
0.8.14
```

Then you will need to setup your account for authentication from the CLI. Several methods are available. One is to download your account settings from the portal and import them on the machine you are using the CLI from:

```
$ azure account download
$ azure account import ~/Downloads/Free\ Trial-2-5-2015-credentials.publishsettings
$ azure account list
```

You are now ready to use the Azure CLI to start VM instances. Pick a location and an image.

```
$ azure vm image list | grep Ubuntu
$ azure vm location list
info:    Executing command vm location list
+ Getting locations
data:    Name
data:    ----------------
data:    West Europe
data:    North Europe
data:    East US 2
data:    Central US
data:    South Central US
data:    West US
data:    East US
data:    Southeast Asia
data:    East Asia
data:    Japan West
info:    vm location list command OK
```

To create an instance with `ssh` access using password authentication use the `azure vm create` command like so:

```
$ azure vm create cookbook --ssh=22
                           --password #@$#%#@$
                           --userName cookbook
                           --location "West Europe"
                           b39f27a8b8c64d52b05eac6a62ebad85__Ubuntu-14_04_1-LTS-amd64-server-20150
...
$ azure vm list
...
data:    Name      Status     Location     DNS Name               IP Address
data:    --------  ---------  -----------  ---------------------  -------------
data:    cookbook  ReadyRole  West Europe  cookbook.cloudapp.net  100.91.96.137
info:    vm list command OK
```

You can then `ssh` to the instance and setup Docker like you did in Recipe 1.1.

Discussion

The Azure CLI is still under active development. The source can be found on GitHub and a Docker machine driver is available.

The Azure CLI also allows you to create a Docker host automatically by using the `azure vm docker create` command.

```
$ azure vm docker create goasguen -l "West Europe" \
                                  b39f27a8b8c64d52b05eac6a62ebad85__Ubuntu-14_04_1-LTS-amd64-serve
info:    Executing command vm docker create
warn:    --vm-size has not been specified. Defaulting to "Small".
info:    Found docker certificates.
...
info:    vm docker create command OK
$ azure vm list
info:    Executing command vm list
+ Getting virtual machines
data:    Name      Status     Location     DNS Name               IP Address
data:    --------  ---------  -----------  ---------------------  -------------
data:    goasguen  ReadyRole  West Europe  goasguen.cloudapp.net  100.112.4.136
```

The host started started will automatically have the Docker daemon running and you will be able to connect to it using the Docker client and a TLS connection.

```
$ docker --tls -H tcp://goasguen.cloudapp.net:4243 ps
CONTAINER ID    IMAGE       COMMAND     CREATED     STATUS      PORTS        NAMES
$ docker --tls -H tcp://goasguen.cloudapp.net:4243 images
REPOSITORY  TAG         IMAGE ID    CREATED     VIRTUAL SIZE
```

Using this CLI is not Docker specific. This CLI gives you access to the complete set of Azure APIs. However, using it to start instances and install Docker in them significantly streamlines the provisioning of Docker hosts.

See Also

- The Azure command line interface
- Starting a coreOS instance on Azure.
- Using Docker machine with Azure.

8.5 Starting a Docker Host on Azure with Docker Machine

Problem

You know how to start a Docker Host on Azure using the Azure CLI but you would like to unify the way you start Docker hosts in the Cloud using Docker machine.

Solution

Use the Docker Machine Azure driver. In Figure 1-6 you saw how to use *Docker Machine* to start a Docker Host on Digital Ocean, the same thing can be done on Microsoft Azure. You will need a valid subscription to Azure.

You need to download the *docker-machine* binary. Go to the documentation site and choose the correct binary for your local computer architecture. For example on OSX:

```
$ wget https://github.com/docker/machine/releases/download/v0.1.0/docker-machine_darwin-amd64
$ mv docker-machine_darwin-amd64 docker-machine
$ chmod +x docker-machine
$ ./docker-machine --version
docker-machine version 0.1.0
```

With a valid Azure subscription, create a X.509 certificate and upload it through the Azure portal. You can create the certificate with the following commands:

```
$ openssl req -x509 -nodes -days 365 -newkey rsa:1024 -keyout mycert.pem -out mycert.pem
$ openssl pkcs12 -export -out mycert.pfx -in mycert.pem -name "My Certificate"
$ openssl x509 -inform pem -in mycert.pem -outform der -out mycert.cer
```

Upload the `mycert.cer` and define the following environment variables:

```
$ export AZURE_SUBSCRIPTION_ID=<UID of your subscription>
$ export AZURE_SUBSCRIPTION_CERT=mycert.pem
```

You can then use `docker-machine` and set your local Docker client to use this remote Docker daemon:

```
$ ./docker-machine create -d azure goasguen-foobar
INFO[0002] Creating Azure machine...
INFO[0061] Waiting for SSH...
INFO[0360] "goasguen-foobar" has been created and is now the active machine.
INFO[0360] To point your Docker client at it, run this in your shell: $(docker-machine env goasgue
```

```
$ ./docker-machine ls
NAME       ACTIVE   DRIVER   STATE     URL                                    SWARM
toto1111   *        azure    Running   tcp://goasguen-foobar.cloudapp.net:2376
$ $(docker-machine env goasguen-foobar)
$ docker ps
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS    PORTS    NAMES
```

In the example above `goasguen-foobar` is the name that I gave to my Docker machine. This needs to be a globally unique name. Chances are that names like *foobar* and *test* have already been taken.

Discussion

With your local Docker client setup to use the remote Docker daemon running in this Azure virtual machine, you can pull images from your favorite registries and start containers.

For example, let's start an *nginx* container:

```
$ docker pull nginx
$ docker run -d -p 80:80 nginx
```

In order to expose port 80 of this remote host in Azure you will need to add an endpoint to the VM that was created. Head over to the Azure portal, select the VM (here, `goasguen-foobar`) and add an endpoint for HTTP request, like in the snapshot below. Once the endpoint is created, you will be able to access *nginx* at *`http://<unique_name>.cloudapp.net`*

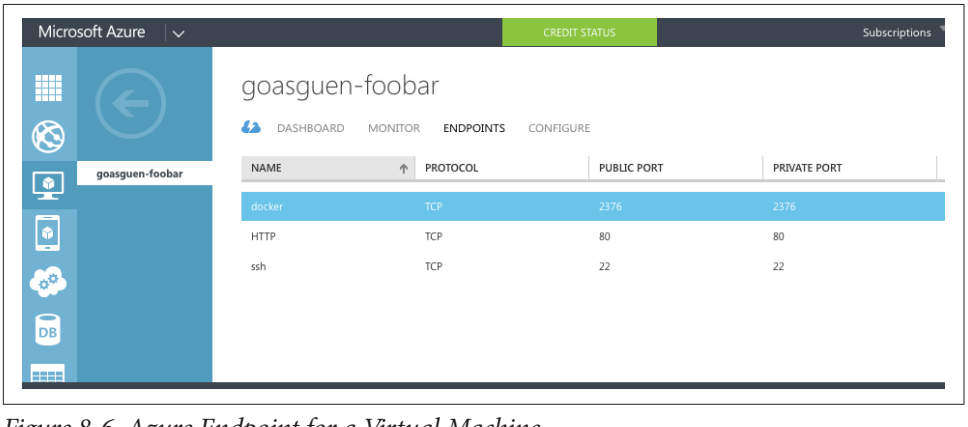


Figure 8-6. Azure Endpoint for a Virtual Machine

See Also

- Docker Machine Azure driver documentation.

8.6 Running Cloud Providers CLI in Docker Containers

Problem

You want to take advantage of containers and run your Cloud provider CLI of choice within a container. This gives you more portability options and avoids having to install the CLI from scratch. You just need to download a container image from the Docker hub.

Solution

For the Google GCE CLI, there is a public image maintained by Google. Download the image via `docker pull` and run your GCE commands through interactive ephemeral containers.

For example using `boot2docker` on a OSX machine:

```
$ boot2docker up
$ $(boot2docker shellinit)
$ docker pull google/cloud-sdk
$ docker images | grep google
google/cloud-sdk    latest      a7e7bcdfdc16    10 days ago     1.263 GB
```

You can then login and issue commands like described in Recipe 8.3. The only difference is that the CLI is actually running within containers. The login command is issue through a named container. That named container is used as a data volume container (i.e `--volumes-from cloud-config`) in subsequent CLI calls. This allows you to use the authorization token that is stored in it.

```
$ docker run -t -i --name gcloud-config google/cloud-sdk gcloud auth login
Go to the following link in your browser:
...
$ docker run --rm \
             -ti \
             --volumes-from gcloud-config google/cloud-sdk \
             gcloud compute zones list
NAME           REGION       STATUS     NEXT_MAINTENANCE TURNDOWN_DATE
asia-east1-c   asia-east1   UP
asia-east1-a   asia-east1   UP
asia-east1-b   asia-east1   UP
europe-west1-b europe-west1 UP
europe-west1-c europe-west1 UP
us-central1-f  us-central1  UP
```

```
us-central1-b  us-central1  UP
us-central1-a  us-central1  UP
```

Using an alias makes things even better:

```
$ alias magic='docker run --rm \
                          -ti \
                          --volumes-from gcloud-config \
                          google/cloud-sdk gcloud'
$ magic compute zones list
NAME           REGION       STATUS     NEXT_MAINTENANCE TURNDOWN_DATE
asia-east1-c   asia-east1   UP
asia-east1-a   asia-east1   UP
asia-east1-b   asia-east1   UP
europe-west1-b europe-west1 UP
europe-west1-c europe-west1 UP
us-central1-f  us-central1  UP
us-central1-b  us-central1  UP
us-central1-a  us-central1  UP
```

Discussion

A Similar process can be used for AWS. If you search for an `awscli` image on Docker hub, you will see several answers. The Dockerfile provided show you how the image was constructed and the CLI installed within the image. If we take the `nathanleclaire/awscli` image, we notice that no volumes are mounted to keep the credentials from container to container. Hence we need to pass the AWS access keys as environment variables when we launch a container:

```
$ docker pull nathanleclaire/awscli
$ docker run --rm \
             -ti \
             -e AWS_ACCESS_KEY_ID="AKIAIUCASDLGFIGDFGS" \
             -e AWS_SECRET_ACCESS_KEY="HwQdNnAIqo/XSVASGayqerwy9797arghqQERfrgot" \
             nathanleclaire/awscli \
             --region eu-west-1 \
             --output=table \
             ec2 describe-key-pairs
-----------------------------------------------------------------------------
|                            DescribeKeyPairs                               |
+---------------------------------------------------------------------------+
||                               KeyPairs                                  ||
|+-----------------------------------------------------------+-----------+|
||                      KeyFingerprint                       |  KeyName  ||
|+-----------------------------------------------------------+-----------+|
||  69:aa:64:4b:72:50:ee:15:9a:da:71:4e:44:cd:db:c0:a1:72:38:36  |  cookbook ||
|+-----------------------------------------------------------+-----------+|
```

We also notice that `aws` was setup as an entrypoint in this image, therefore there is no need to specify it and we should only pass arguments to it.

You can build your own AWS CLI image which allows you handle API keys more easily.

See Also

- Official documentation on the containerized Google SDK

8.7 Using Google Container Registry to Store your Docker Images

Problem

You have used a Docker private registry hosted on your own infrastructure (see Recipe 2.9) but you would like to take advantage of a hosted service, specifically you would like to take advantage of the newly announced Google container registry currently in Beta release.

There are other hosted private registry solutions. Like Docker Hub Enterprise or Quay.io. This recipe does not represent an endorsement of one versus another.

Solution

If you have not done so yet, go through Recipe 8.1 to sign up on Google Cloud Platform. Then download the Google Cloud CLI and create a project (see Recipe 8.3). Since the Google Container Registry (GCR) is a preview, update your `gcloud` CLI on your Docker host to load the preview components. You will have access to `gcloud preview docker` which is a wrapper around the `docker` client.

```
$ gcloud components update preview
$ gcloud preview docker help
Usage: docker [OPTIONS] COMMAND [arg...]

A self-sufficient runtime for linux containers.
...
```

In this example, we have created a *cookbook* project on Google Cloud with project ID *sylvan-plane-862*. Your project name and project ID will differ.

As an example, on the Docker host that we are using, we have a *busybox* image that we to upload to GCR. You need to tag the image you want to push to the GCR so that it follows the namespace naming convention of the GCR (i.e _gcr.io/project_id/image_name). You can then upload the image with `gcloud preview docker push`

```
$ docker images | grep busybox
busybox     latest     a9eb17255234     8 months ago     2.433 MB
$ docker tag busybox gcr.io/sylvan_plane_862/busybox
$ gcloud preview docker push gcr.io/sylvan_plane_862/busybox
The push refers to a repository [gcr.io/sylvan_plane_862/busybox] (len: 1)
Sending image list
Pushing repository gcr.io/sylvan_plane_862/busybox (1 tags)
511136ea3c5a: Image successfully pushed
42eed7f1bf2a: Image successfully pushed
120e218dd395: Image successfully pushed
a9eb17255234: Image successfully pushed
Pushing tag for rev [a9eb17255234] on \
{https://gcr.io/v1/repositories/sylvan_plane_862/busybox/tags/latest}
```

The naming convention of the GCR namespace is such that if you have dashes in your project ID you need to replace them with underscores.

If you navigate to your storage browser in your Google Developers console, you will see that a new bucket has been created and that all the layers making your image have been uploaded.

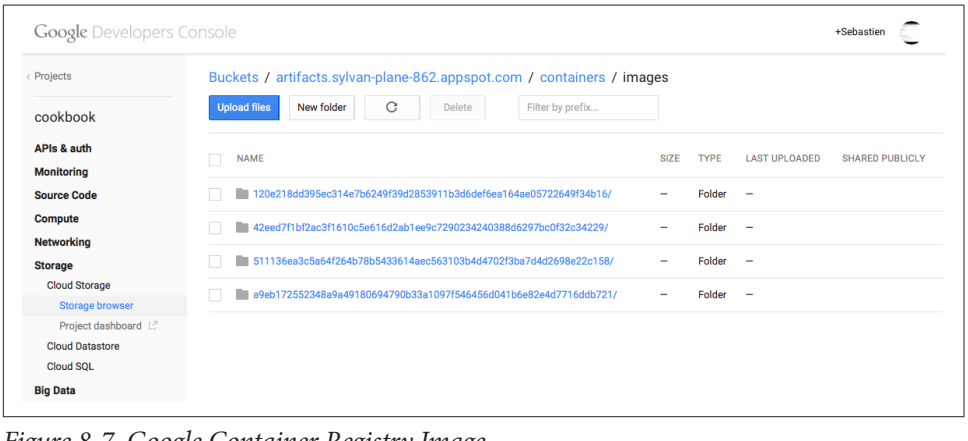


Figure 8-7. Google Container Registry Image

Discussion

Automatically, Google compute instances started in the same project that you used to push the images to, will have the correct privileges to pull that image. If you want other people to be able to pull that image you will need to add them as members to that project. You can set your project by default with `gcloud config set project <project_id>` so you do not have to specify it on subsequent `gcloud` commands.

Let's start an instance in GCE, `ssh` to it and pull the busybox image from GCR.

```
$ gcloud compute instances create cookbook-gce --image container-vm \
                                               --zone europe-west1-c \
                                               --machine-type f1-micro
$ gcloud compute ssh cookbook-gce
Updated [https://www.googleapis.com/compute/v1/projects/sylvan-plane-862].
...
$ sudo gcloud preview docker pull gcr.io/sylvan_plane_862/busybox
Pulling repository gcr.io/sylvan_plane_862/busybox
a9eb17255234: Download complete
511136ea3c5a: Download complete
42eed7f1bf2a: Download complete
120e218dd395: Download complete
Status: Downloaded newer image for gcr.io/sylvan_plane_862/busybox:latest
sebastiengoasguen@cookbook:~$ sudo docker images | grep busybox
gcr.io/sylvan_plane_862/busybox   latest    a9eb17255234    ...
```

To be able to push from a GCE instance you will need to start it with the correct scope `--scopes` *`https://www.googleapis.com/auth/devstorage.read_write`*.

8.8 Using Docker in GCE Google-Container Instances

Problem

You know how to start instances in Google GCE and configure Docker to be setup at boot time, but you would like to use an image that is already configured with Docker

Solution

As mentioned in Recipe 8.3, GCE offers some *container optimized images*.

Make sure that you set your project to the project ID with `gcloud config set project <project_id>`

```
$ gcloud compute images list
NAME                              PROJECT           ALIAS       DEPRECATED STATUS
...
container-vm-v20141208            google-containers container-vm           READY
container-vm-v20150112            google-containers container-vm           READY
container-vm-v20150129            google-containers container-vm           READY
...
```

These images are based on Debian 7, they contain the Docker daemon and the Kubernetes kubelet service.

Kubernetes is discussed in more details in the Chapter 5 upcoming chapter.

The `kubelet` service running in instances based on these images, allows the user to pass a manifest (known as *pod*) that describes the set of containers that need to run in the instance. The `kubelet` will start the containers and monitor them. A pod manifest is a YAML file like so:

```
version: v1beta2
containers:
  - name: nginx
    image: nginx
    ports:
      - name: nginx
        hostPort: 80
        containerPort: 80
```

Your image in the pod manifest can reference an image in the Google Container Registry (GCR, see Recipe 8.7) for instance `gcr.io/<your_project_name>/busybox`.

This simple manifest, describes a single container based on the *nginx* image and an exposed port. You can pass this manifest to the *gcloud* instance creation command. Save the above YAML file in `nginx.yml` then to start the instance:

```
$ gcloud compute instances create cookbook-gce \
         --image container-vm \
         --metadata-from-file google-container-manifest=nginx.yml \
         --zone europe-west1-c \
         --machine-type f1-micro
```

In your Google GCE console, you can browse to the started instance. You can allow HTTP traffic as well as see the container manifest you passed. Once the containers

defined in the pod manifest have started, open your browser at the IP of the instance on port 80 and you will see the `nginx` welcome page.

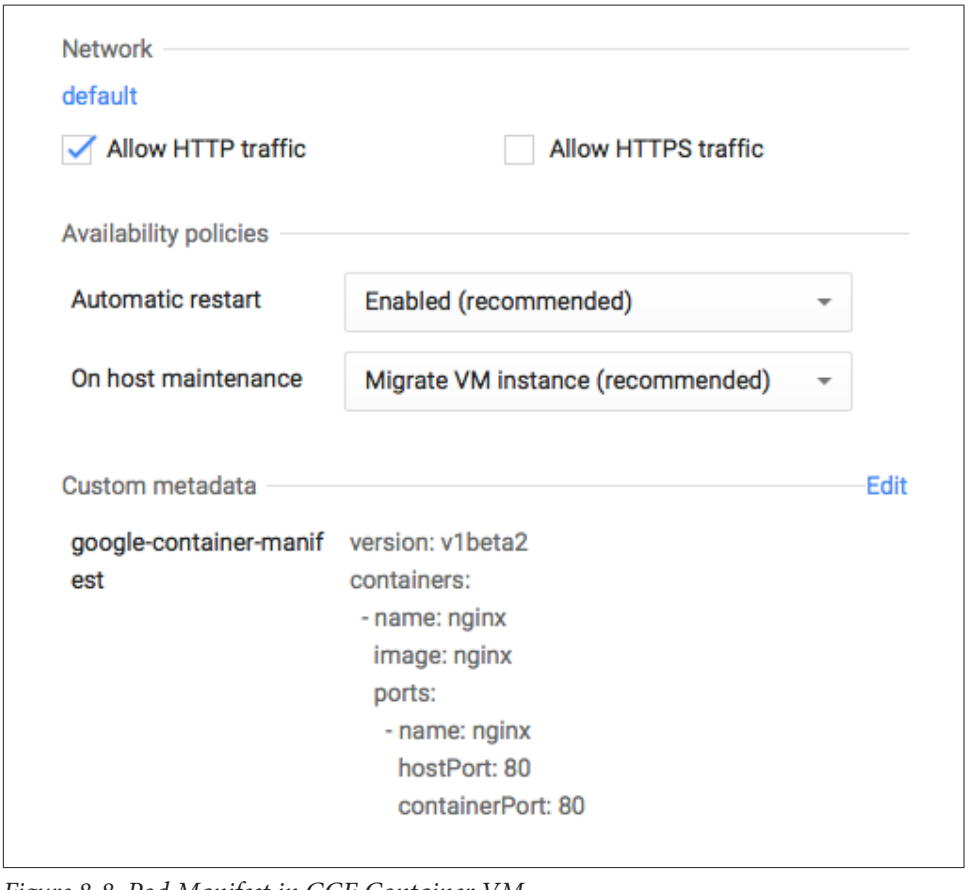


Figure 8-8. Pod Manifest in GCE Container VM

Discussion

If you connect to the instance directly via `ssh` you can list the containers that are running. You will see a `google/cadvisor` container used for monitoring and two `kubernetes/pause:go` container. The last two acts as network proxy to the `cadvisor` monitoring container and to the pod exposed ports.

```
$ gcloud compute ssh cookbook-gce
...
sebastiengoasguen@cookbook-gce:~$ sudo docker ps
CONTAINER ID        IMAGE                   COMMAND                ...
1f83bb1197c9        nginx:latest            "nginx -g 'daemon of   ...
b1e6fed3ee20        google/cadvisor:0.8.0   "/usr/bin/cadvisor"    ...
```

```
79e879c48e9e        kubernetes/pause:go     "/pause"                ...
0c1a51ab2f94        kubernetes/pause:go     "/pause"                ...
```

In Chapter 9 we will discuss `cadvisor`.

8.9 Starting a Docker Host on AWS Using Docker Machine

Problem

You understand how to use the AWS CLI to start an instance in the Cloud and know how to install Docker (see Recipe 8.2). But you would like to use a streamlined process integrated with the Docker user experience.

Solution

Use Docker machine and its AWS EC2 driver.

Download the release candidate binaries for Docker machine. Set some environment variables so that Docker Machine knows your AWS API keys and your default VPC in which to start the Docker host. Then use Docker Machine to start the instance. Docker will automatically setup a TLS connection and you will be able to use this remote Docker host started in AWS. On a 64 bit linux machine.

```
$ wget https://github.com/docker/machine/releases/download/\
      v0.1.0-rc2/docker-machine_linux-amd64
$ chmod +x docker-machine
$ export AWS_ACCESS_KEY_ID=<your AWS access key>
$ export AWS_SECRET_ACCESS_KEY_ID=<your AWS secret key>
$ export AWS_VPC_ID=<the VPC ID you want to start the instance in>
$ ./docker-machine create -d amazonec2 cookbook
INFO[0000] Launching instance...
INFO[0023] Waiting for SSH ...
...
INFO[0129] "cookbook" has been created and is now the active machine
INFO[0129] To connect: docker $(docker-machine config cookbook) ps
```

Once the machine has been created, you can use your local Docker client to communicate with it. Do not forget to kill the machine once your are done.

```
$ docker $(./docker-machine config cookbook) ps
CONTAINER ID    IMAGE       COMMAND     CREATED     STATUS      PORTS       NAMES
$ ./docker-machine ls
NAME       ACTIVE   DRIVER      STATE     URL
cookbook   *        amazonec2   Running   tcp://<IP_of_your_Docker_machine_in_AWS>:2376
$ ./docker-machine kill cookbook
```

You can manage your *machines* directly from the Docker Machine CLI:

```
$ ./docker-machine -h
...
COMMANDS:
```

```
active       Get or set the active machine
create       Create a machine
config       Print the connection config for machine
inspect      Inspect information about a machine
ip           Get the IP address of a machine
kill         Kill a machine
ls           List machines
restart      Restart a machine
rm           Remove a machine
env          Display the commands to set up the environment for the Docker client
ssh          Log into or run a command on a machine with SSH
start        Start a machine
stop         Stop a machine
upgrade      Upgrade a machine to the latest version of Docker
url          Get the URL of a machine
help, h      Shows a list of commands or help for one command
```

Discussion

Docker machine 0.1.0 is not yet released. It contains drivers for several cloud providers. We already showcase the Digital Ocean driver (see ???).

The AWS driver takes several command line options to set your keys, VPC, key pair, image and instance type. You can set them up as environment variables like we did above or directly on the `machine` command line.

```
$ ./docker-machine create -h
...
OPTIONS:
   --amazonec2-access-key                       AWS Access Key [$AWS_ACCESS_KEY_ID]
   --amazonec2-ami                              AWS machine image [$AWS_AMI]
   --amazonec2-instance-type 't2.micro'         AWS instance type [$AWS_INSTANCE_TYPE]
   --amazonec2-region 'us-east-1'               AWS region [$AWS_DEFAULT_REGION]
   --amazonec2-root-size '16'                   AWS root disk size (in GB) [$AWS_ROOT_SIZE]
   --amazonec2-secret-key                       AWS Secret Key [$AWS_SECRET_ACCESS_KEY]
   --amazonec2-security-group 'docker-machine'  AWS VPC security group [$AWS_SECURITY_GROUP]
   --amazonec2-session-token                    AWS Session Token [$AWS_SESSION_TOKEN]
   --amazonec2-subnet-id                        AWS VPC subnet id [$AWS_SUBNET_ID]
   --amazonec2-vpc-id                           AWS VPC id [$AWS_VPC_ID]
   --amazonec2-zone 'a'                         AWS zone for instance (i.e. a,b,c,d,e) [$AWS_ZONE]
```

Finally, *machine* will create a SSH key pair and a security group for you. The security group will allow traffic on port 2376 to allow communications over TLS from a Docker client.

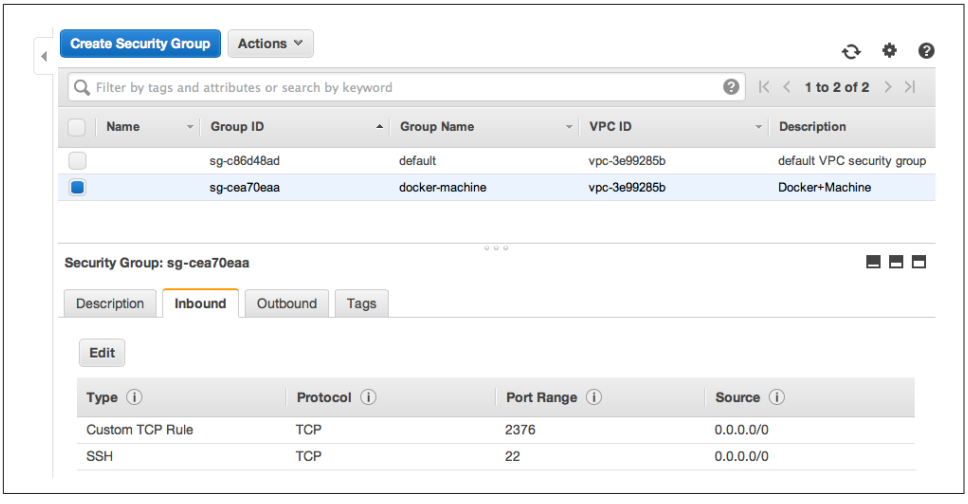


Figure 8-9. Security Group for Machine

8.10 Using Kubernetes in the Cloud via Google Container Engine

Problem

You want to use a group of Docker hosts and manage containers on them. You like the Kubernetes container orchestration engine but would like to use it as a hosted cloud service.

Solution

Use the Google container engine service. This new service allows you to create a Kubernetes cluster on-demand using the Google API. A cluster will be composed of a master node and a set of compute nodes that act as container VMs similar to what was described in Recipe 8.8.

Google container engine is currently in Alpha preview. Kubernetes is under heavy development. Expect frequent changes to the API and use it in production at your own risk. For details on Kubernetes see Chapter 5.

Update your `gcloud` SDK to use the container engine preview. If you have not yet installed the Google SDK, see Recipe 8.3.

```
$ gcloud components update preview
```

To start a Kubernetes cluster using the Google container engine service is a single command

```
$ gcloud preview container clusters create cook --num-nodes 1 --machine-type g1-small
$ gcloud compute instances list
NAME            ZONE          MACHINE_TYPE INTERNAL_IP    EXTERNAL_IP    STATUS
k8s-cook-master us-central1-a g1-small     10.240.40.103 104.154.39.165 RUNNING
k8s-cook-node-1 us-central1-a g1-small     10.240.82.238 130.211.137.25 RUNNING
```

Your cluster IP addresses will differ from what shown above.

You could also create a cluster through the Google cloud web console.

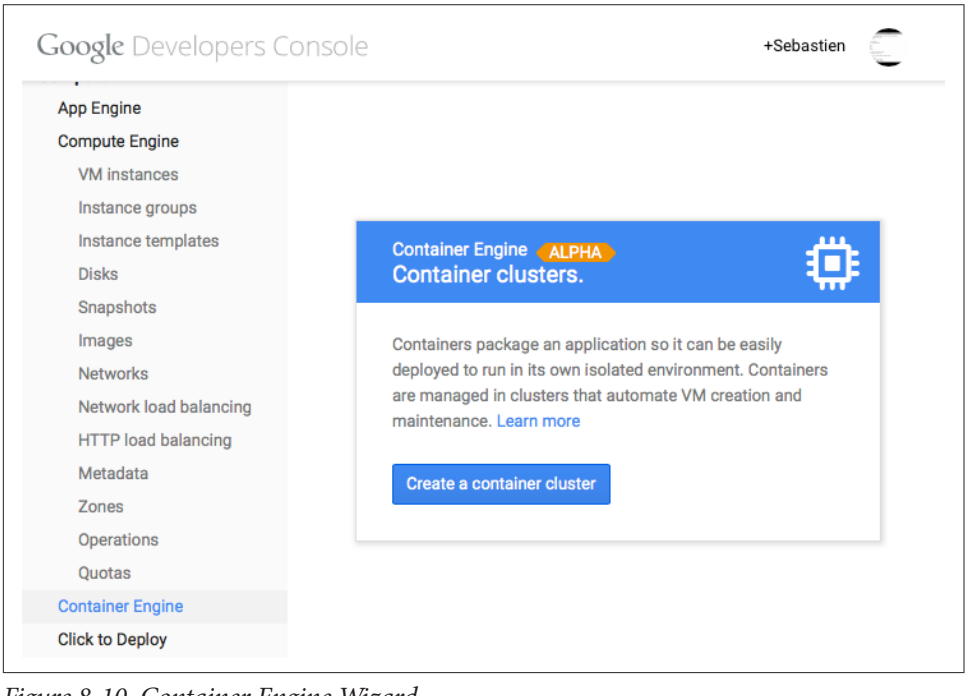


Figure 8-10. Container Engine Wizard

Once your cluster is up you can submit containers to it. Meaning that you can interact with the underlying Kubernetes master node to launch group of containers on the set of nodes in your cluster. Groups of containers are defined as *pods*. This is the same concept introduced in Recipe 8.8. The *gcloud* CLI gives you a convenient way to define simple pods and submit them to the cluster. Below we are going to launch a container using the `tutum/wordpress` image which contains a MySQL database in it.

```
$ gcloud preview container pods create --name wordpress \
                                       --image=tutum/wordpress \
                                       --port=80 wordpress
$ gcloud preview container pods describe wordpress
```

```
POD       IP          CONTAINER(S)    IMAGE(S)            HOST                    LABELS          S
wordpress 10.188.0.5  wordpress       tutum/wordpress     k8s-cook-node-1.c...    name=wordpress  R
```

Once the container is scheduled on one of the cluster nodes, we will need to open the firewall using tags.

```
$ gcloud compute firewall-rules create wordpress-80 --allow tcp:80 \
                                                    --target-tags k8s-cook-node
...
NAME         NETWORK SRC_RANGES RULES  SRC_TAGS TARGET_TAGS
wordpress-80 default 0.0.0.0/0  tcp:80          k8s-cook-node
```

You will then be able to enjoy Wordpress.

Discussion

While we can launch simple pods consisting of a single container, we can also specify a more advanced pod defined in a json or YAML file using the `--config-file` option.

```
$ gcloud preview container pods create --config-file /path/to/pod/pod.json
```

See Also

- Cluster operations
- Pods operations
- Services operations
- Replication controllers operations

8.11 Managing Google Container Engine Resources Using `kubecfg`

Problem

You have created a Kubernetes cluster through the Google container engine service. You know how to create pods, services and replication controllers using the `gcloud` cli but would like to use the default `kubecfg` CLI to do it.

Solution

With a Kubernetes cluster started (see Recipe 8.10), you connect to the master node via `ssh` using the `gcloud` CLI, you then have access to `kubecfg`. The IP addresses listed below will differ.

```
$ gcloud preview container clusters create cookbook --num-nodes 2 \
                                                    --machine-type g1-small
$ $ gcloud compute instances list
NAME                ZONE          MACHINE_TYPE INTERNAL_IP    EXTERNAL_IP    STATUS
k8s-cookbook-node-2 us-central1-a g1-small     10.240.222.210 104.154.39.165 RUNNING
k8s-cookbook-node-1 us-central1-a g1-small     10.240.240.234 130.211.170.38 RUNNING
k8s-cookbook-master us-central1-a g1-small     10.240.154.207 130.211.115.17 RUNNING
$ gcloud compute ssh k8s-cookbook-master --zone us-central1-a
sebastiengoasguen@k8s-cookbook-master:~$ which kubecfg
/usr/local/bin/kubecfg
```

The Kubernetes cluster may not have been created in your default zone. Specify the zone with the `--zone` option to `ssh` to the master node.

The power of `kubecfg` is now at your fingertips:

```
$ kubecfg list minions
Minion identifier                       Labels
----------                              ----------
k8s-cookbook-node-1.c.<your_google_project>.internal
k8s-cookbook-node-2.c.<your_google_project>.internal
```

Discussion

The `kubecfg` CLI can be used to manage all resources in a Kubernetes cluster.

```
$ kubecfg -h
....
Usage: kubecfg -h [-c config/file.json|url|-] <method>

Kubernetes REST API:

  kubecfg [OPTIONS] get|list|create|delete|update \
                    <events|minions|nodes|pods|replicationControllers|services>[/<id>]
```

To start a pod you need to define it in a YAML or json file. In Recipe 8.8, we saw an example in YAML. Here we write our pod in a json file, using the newly released Kubernetes *v1beta2* API version. This pod will simply start nginx.

```
{
    "id": "nginx",
    "kind": "Pod",
    "apiVersion": "v1beta2",
    "desiredState": {
        "manifest": {
            "id": "nginx",
            "version": "v1beta2",
            "containers": [{
                "name": "nginx",
```

```
                "image": "nginx",
                "imagePullPolicy": "PullIfNotPresent",
                "ports": [{"containerPort": 80, "hostPort": 80}],
            }]
        }
    },
}
```

Start the pod and check its status. Once it is running and that you have a firewall with port 80 open for the cluster nodes, you will be able to see the nginx welcome page. Additional examples are available on the Kubernetes GitHub page. For more information on the `skydns` pod that you see running below check the Chapter 5 chapter.

```
$ kubecfg -c nginx.json create pods
$ kubecfg list pods
$ kubecfg list pods
Name            Image(s)     Host                                                  Labels
----------      ----------   ----------                                            ----------
nginx           nginx        k8s-cookbook-node-2.c.runseb.internal/104.154.39.165
skydns-fplln    quay.io...   k8s-cookbook-node-1.c.runseb.internal/130.211.170.38   k8s-app=skydns
```

To clean things up, remove your pod, exit the master node and delete your cluster.

```
$ kubecfg delete pods/nginx
$ exit # On the kubernetes master
$ gcloud preview container clusters delete cookbook
```

8.12 Getting Setup to Use the EC2 Container Service

Problem

You want to try the new Amazon AWS EC2 container service (ECS).

Solution

ECS is currently a preview and only available in the AWS Northern Virginia region. Getting setup to test ECS involves several steps which are well documented on AWS documentation. In this recipe we summarize the main steps but you should refer to the official documentation for all details.

- Sign up for AWS if you have not done so.
- Log into the AWS console. Review Recipe 8.1 and Recipe 8.2 if you have not read those recipes. You will launch ECS instances within a security group associated to a VPC. Create a VPC and a security group or ensure that you have default ones present.

- Go to the Identity and Account Management (IAM) console and create a role for ECS. If you are not familiar with IAM, this step is a bit advanced and can be followed step by step on the AWS documentation for ECS.
- For the role that you just created create an inline policy. If successfull when you select the `Show Poliyc` link you should see the screenshot below. See the discussion section of this recipe for an automated way of creating this policy using Boto.

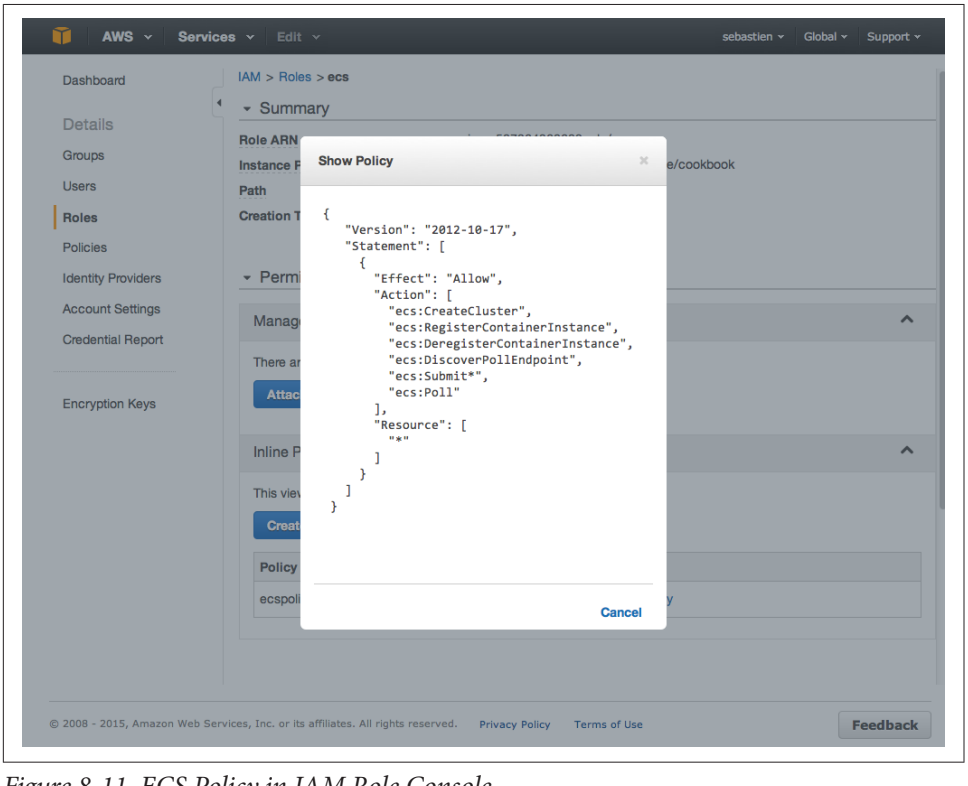


Figure 8-11. ECS Policy in IAM Role Console

- Install the latest AWS CLI. The ECS API is available in version 1.7.0 or greater. You can verify that the `aws ecs` commands are now available.

```
$ sudo pip install awscli
$ aws --version
aws-cli/1.7.8 Python/2.7.9 Darwin/12.6.0
$ aws ecs help

ECS()                                                    ECS()

NAME
```

```
       ecs -

DESCRIPTION
       Amazon  EC2  Container Service (Amazon ECS) is a highly scalable, fast,
       container management service that makes it easy to run, stop, and  man-
       age  Docker containers on a cluster of Amazon EC2 instances. Amazon ECS
       lets you launch and stop container-enabled applications with simple API
       calls,  allows  you to get the state of your cluster from a centralized
       service, and gives you access to many familiar Amazon EC2 features like
       security groups, Amazon EBS volumes, and IAM roles.
...
```

- Create a AWS CLI configuration file which contains the API keys of the IAM user your created above. Note the region being set to is `us-east-1`, which is the Northern Virginia region where ECS is currently available.

```
$ cat ~/.aws/config
[default]
output = table
region = us-east-1
aws_access_key_id = <your AWS access key>
aws_secret_access_key = <your AWS secret key>
```

Once you have completed all these steps you will be ready to use ECS. You will need to create a cluster (see Recipe 8.13), define tasks corresponding to containers and run those tasks to start the containers on the cluster (see Recipe 8.14)

Discussion

Creating the IAM profile and the ECS policy for the instances that will be started to form the cluster can be overwhelming if you have not used AWS before. To ease this step. You can use the online material accompanying this book. It contains a small code that uses the Python Boto client to create the policy.

Install Boto, copy `~/.aws/config` to `~/.aws/credentials`, clone the repository and execute the script.

```
$ git clone https://github.com/how2dock/docbook.git
$ sudo pip install boto
$ cp ~/.aws/config ~/.aws/credentials
$ cd ch08/ecs
$ ./ecs-policy.py
```

This script will create a `ecs` role, an `ecspolicy` policy and a `cookbook` instance profile. You can edit the script to change these names. After completion, you should see the role and the policy in the IAM console.

See Also

- Video of an ECS demo
- ECS documentation

8.13 Creating a ECS Cluster

Problem

You are setup to use ECS (see Recipe 8.12), you want to create a cluster and some instances in it to run containers.

Solution

Use the AWS CLI that you installed in Recipe 8.12 and explore the new ECS API. In this recipe we will learn to use:

- `aws ecs list-clusters`
- `aws ecs create-cluster`
- `aws ecs describe-clusters`
- `aws ecs list-container-instances`
- `aws ecs delete-cluster`

By default you have one cluster in ECS but until you have launched an instance in that cluster it is not active. Try to describe the default cluster.

```
$ aws ecs describe-clusters
-------------------------------------------------------------------
|                        DescribeClusters                         |
+-----------------------------------------------------------------+
||                           failures                            ||
|+-------------------------------------------------+----------+|
||                          arn                    | reason   ||
|+-------------------------------------------------+----------+|
||  arn:aws:ecs:us-east-1:587264368683:cluster/default  |  MISSING ||
|+-------------------------------------------------+----------+
```

Currently you are limited to two ECS clusters.

To activate this cluster, launch an instance using Boto. The AMI used is specific to ECS and contains the ECS agent. You will need to have created a SSH key pair to SSH into the instance if you want to and you will need an instance profile associated with a role that has the ECS policy (see Recipe 8.12).

```
$ python
...
>>> import boto
>>> c = boto.connect_ec2()
>>> c.run_instances('ami-34ddbe5c', \
                    key_name='ecs', \
                    instance_type='t2.micro', \
                    instance_profile_name='cookbook')
```

With one instance started, wait that it runs and registers in the cluster. Then if you *describe* the cluster again you will see that the default cluster has switched to active state. You can also list container instances.

```
$ aws ecs describe-clusters
-------------------------------------------------------------------------------
|                            DescribeClusters                                 |
+-----------------------------------------------------------------------------+
||                                clusters                                   ||
|+-----------------------------------------------------+-------------+-----------+|
||                    clusterArn                       | clusterName  | status   ||
|+-----------------------------------------------------+-------------+-----------+|
||  arn:aws:ecs:us-east-1:587264368683:cluster/default |  default     |  ACTIVE  ||
|+-----------------------------------------------------+-------------+-----------+|
sebimac:ecs sebgoa$ aws ecs list-container-instances
-------------------------------------------------------------------------------------------------------------
|                                  ListContainerInstances                                                   |
+-----------------------------------------------------------------------------------------------------------+
||                                   containerInstanceArns                                                 ||
|+---------------------------------------------------------------------------------------------------------+|
||  arn:aws:ecs:us-east-1:587264368683:container-instance/e62d3d79-c88f-48f9-99e4-28bc19b2a665  ||
|+---------------------------------------------------------------------------------------------------------+|
```

Starting additional instances will just increase the size of the cluster.

```
$ aws ecs list-container-instances
-------------------------------------------------------------------------------------------------------------
|                                  ListContainerInstances                                                   |
+-----------------------------------------------------------------------------------------------------------+
||                                   containerInstanceArns                                                 ||
|+---------------------------------------------------------------------------------------------------------+|
||  arn:aws:ecs:us-east-1:587264368683:container-instance/75738343-6fad-46fd-ba62-d4070b270207  ||
||  arn:aws:ecs:us-east-1:587264368683:container-instance/b457e535-feb6-4a14-9a1b-44f9612239d2  ||
||  arn:aws:ecs:us-east-1:587264368683:container-instance/e5c0be59-ce1d-40d8-8b72-e845f2d87efa  ||
||  arn:aws:ecs:us-east-1:587264368683:container-instance/e62d3d79-c88f-48f9-99e4-28bc19b2a665  ||
|+---------------------------------------------------------------------------------------------------------+|
```

Since these container instances are regular EC2 instances, you will see them in your EC2 console. If you have setup a SSH key properly and opened port 22 on the security group used you can also `ssh` to them:

The `aws ecs describe-container-instances` call does not seem to be working properly. So check the IP addresses of your instances in the AWS console.

```
$ ssh -i ~/.ssh/id_rsa_ecs ec2-user@52.1.224.245
...

    __|  __|  __|
    _|  (   \__ \  Amazon ECS-Optimized Amazon Linux AMI
  ____|\___|____/


  Image created: Thu Dec 18 01:39:14 UTC 2014
  PREVIEW AMI

9 package(s) needed for security, out of 10 available
Run "sudo yum update" to apply all updates.
[ec2-user@ip-172-31-33-78 ~]$ docker ps
CONTAINER ID    IMAGE                              ...
4bc4d480a362    amazon/amazon-ecs-agent:latest     ...
[ec2-user@ip-172-31-33-78 ~]$ docker version
Client version: 1.3.3
Client API version: 1.15
Go version (client): go1.3.3
Git commit (client): c78088f/1.3.3
OS/Arch (client): linux/amd64
Server version: 1.3.3
Server API version: 1.15
Go version (server): go1.3.3
Git commit (server): c78088f/1.3.3
```

You see that the container instance is running Docker 1.3.3 and that the ECS agent is actually a container.

Discussion

While you can use the default cluster, you can also create your own.

```
$ aws ecs create-cluster --cluster-name cookbook
-------------------------------------------------------------------------------------
|                                CreateCluster                                      |
+-----------------------------------------------------------------------------------+
||                                  cluster                                        ||
|+-----------------------------------------------------+--------------+----------+|
||                    clusterArn                       | clusterName  | status   ||
```

```
|+-----------------------------------------------------+-------------+----------+|
||  arn:aws:ecs:us-east-1:587264368683:cluster/cookbook |  cookbook   |  ACTIVE  ||
|+-----------------------------------------------------+-------------+----------+|
$ aws ecs list-clusters
-----------------------------------------------------
|                   ListClusters                    |
+---------------------------------------------------+
||                    clusterArns                  ||
|+-------------------------------------------------+|
||  arn:aws:ecs:us-east-1:587264368683:cluster/cookbook  ||
||  arn:aws:ecs:us-east-1:587264368683:cluster/default   ||
|+-------------------------------------------------+|
```

To launch instances in that freshly created cluster instead of the default one you will just need to pass some user data during the instance creation step. Via Boto this can be achieved with the following script:

```
#!/usr/bin/env python

import boto
import base64

userdata="""
#!/bin/bash
echo ECS_CLUSTER=cookbook >> /etc/ecs/ecs.config
"""

c = boto.connect_ec2()
c.run_instances('ami-34ddbe5c', \
                key_name='ecs', \
                instance_type='t2.micro', \
                instance_profile_name='cookbook', \
                user_data=base64.b64encode(userdata))
```

Once you are done with the cluster you can delete it entirely with the `aws ecs delete-cluster --cluster cookbook` command.

See Also

- The ECS agent on GitHub.

8.14 Starting Docker Containers on a ECS Cluster

Problem

You know how to create a ECS cluster on AWS (see Recipe 8.13), you are ready to start containers on the instances forming the cluster.

Solution

Define your containers or group of containers in a definition file in json format. This will be called a task. You will register this task and then run it. It is a two step process. Once the task is running in the cluster you can *list*, *stop* and *start* it.

For example to run `nginx` in a container based on the `nginx` image from Docker hub, you create the following task definition in json format:

```
[
  {
    "environment": [],
    "name": "nginx",
    "image": "nginx",
    "cpu": 10,
    "portMappings": [
      {
       "containerPort": 80,
       "hostPort": 80
      }
    ],
    "memory": 10,
    "essential": true
  }
]
```

You can notice the similarities between this task definition, a Kubernetes pod (Recipe 8.11) and a fig file (???). To register this task use the ECS `register-task-definition` call, specify a *family* which groups the tasks and helps you keep revision history which can be handy for roll back purposes.

```
$ aws ecs register-task-definition --family nginx
                                   --container-definitions file://$PWD/nginx.json
$ aws ecs list-task-definitions
-------------------------------------------------------------------
|                      ListTaskDefinitions                        |
+-----------------------------------------------------------------+
||                      taskDefinitionArns                       ||
|+---------------------------------------------------------------+|
||  arn:aws:ecs:us-east-1:587264368683:task-definition/nginx:1   ||
|+---------------------------------------------------------------+|
```

To start the container defined in this task definition you simply use the `run-task` command and specify the number of containers you want running. To stop the container, you stop the task specifying it via its task UUID obtained from `list-tasks`, as shown below:

```
$ aws ecs run-task --task-definition nginx:1 --count 1
$ aws ecs stop-task --task 6223f2d3-3689-4b3b-a110-ea128350adb2
```

ECS will schedule the task on one of the container instances in your cluster. The image will be pulled from Docker hub and the container started using the options specified in the task definion. At this preview stage of ECS, it is not straightforward to find the instance where the task is running and find the associated IP address. If you have multiple instances running, you will have to do a bit of a guess work. There does not seem to be a proxy service like in Kubernetes either.

Since ECS is in preview, the API might change. The returned objects from API calls might also be altered. Expect changes and improvements.

Discussion

While the Nginx example above represents a task with a single container running, you can also define a task with linked containers. The task definition reference describes all possible keys that can be used to define a task. to continue with our example of running Wordpress with two containers (a `wordpress` one and a `mysql` one), we can define a *wordpress* task. It is a translation of the fig (see ???) file to AWS ECS task defintion format. It will not go unnoticed that a standardization effort among *fig*, *pod* and *task* would benefit the community.

```
[
  {
    "image": "wordpress",
    "name": "wordpress",
    "cpu": 10,
    "memory": 200,
    "essential": true,
    "links": [
      "mysql"
    ],
    "portMappings": [
      {
        "containerPort": 80,
        "hostPort": 80
      }
    ],
    "environment": [
      {
        "name": "WORDPRESS_DB_NAME",
        "value": "wordpress"
      },
      {
        "name": "WORDPRESS_DB_USER",
        "value": "wordpress"
      },
      {
```

```
        "name": "WORDPRESS_DB_PASSWORD",
        "value": "wordpresspwd"
      }
    ]
  },
  {
    "image": "mysql",
    "name": "mysql",
    "cpu": 10,
    "memory": 200,
    "essential": true,
    "environment": [
      {
        "name": "MYSQL_ROOT_PASSWORD",
        "value": "wordpressdocker"
      },
      {
        "name": "MYSQL_DATABASE",
        "value": "wordpress"
      },
      {
        "name": "MYSQL_USER",
        "value": "wordpress"
      },
      {
        "name": "MYSQL_PASSWORD",
        "value": "wordpresspwd"
      }
    ]
  }
]
```

The task is registered the same way as done previously with Nginx, but we specify a new *family*. When the task is run, it could however fail due to constraints not being met. In this example, my container instances where of type `t2.micro` with 1GB of memory. Since the task defintion is asking for 500 MB for wordpress and 500 MB for Mysql, there was not enough memory for the cluster scheduler to find an instance that matched the constraints and running the task failed.

```
$ aws ecs register-task-definition --family wordpress
                                   --container-definitions file://$PWD/wordpress.json
$ aws ecs run-task --task-definition wordpress:1 --count 1
-------------------------------------------------------------------------------------------------------------
|                                                  RunTask                                                  |
+-----------------------------------------------------------------------------------------------------------+
||                                                 failures                                                  |
|+---------------------------------------------------------------------------------------------------------+|
||                                              arn                                                        ||
|+---------------------------------------------------------------------------------------------------------+|
||  arn:aws:ecs:us-east-1:587264368683:container-instance/f2acd785-5c55-4736-8cfe-3ae79e741e39               ||
||  arn:aws:ecs:us-east-1:587264368683:container-instance/9ff17f44-df6f-4cff-875a-68e80eb51fb3               ||
```

```
||  arn:aws:ecs:us-east-1:587264368683:container-instance/038fc90b-6bca-4383-a052-4579cf4cc848  |
|+-----------------------------------------------------------------------------------------------+
```

You can edit the task definition, relax the memory constraint and register a new task in the same family (revision 2). It will successfully run. If you log into the instance running this task you will see the containers running alongside the ECS agent.

```
$ aws ecs run-task --task-definition wordpress:2 --count 1
$ ssh -i ~/.ssh/id_rsa_ecs ec2-user@54.152.108.134
...

    __|  __|  __|
    _|  (   \__ \  Amazon ECS-Optimized Amazon Linux AMI
  ____|\___|____/

...
[ec2-user@ip-172-31-36-83 ~]$ docker ps
CONTAINER ID    IMAGE              ...   PORTS                          NAMES
36d590a206df    wordpress:4        ...   0.0.0.0:80->80/tcp             ecs-wordpress-2-wordpress-baa
893d1bd24421    mysql:5            ...   3306/tcp                       ecs-wordpress-2-mysql-a4eedcf
81023576f81e    amazon/amazon-ecs  ...   127.0.0.1:51678->51678/tcp     ecs-agent
```

Enjoy ECS and keep an eye on improvements and general availability.

See Also

- Task definition reference

8.15 Starting an Application in the Cloud Using Docker Support in AWS Beanstalk

Problem

You would like to deploy a Docker based application in the Cloud by just *pushing* your Dockerfile. You want the Cloud service to automatically spin up instances and configure possible load-balancers.

Solution

Use AWS Elastic Beanstalk. Beanstalk use AWS EC2 instances, it can automatically create an elastic load-balancer, create a security group and monitor the health of your application and resources. Docker support in Beanstalk was announced in April 2014. Originally, Beanstalk only supported single container applications, however recently AWS announced a coupling between AWS ECS and Beanstalk. This coupling allows you to let Beanstalk use an ECS cluster as an environment for your application and run multiple containers per instances (see Recipe 8.16).

To illustrate Docker support in Beanstalk we are going to setup a Beanstalk environment using AWS CLI tools, and deploy the 2048 game using a single Dockerfile. This is a variant of the official Beanstalk documentation.

To get started you will need couple things:

- An AWS account (see Figure 8-1).
- The AWS CLI (see Recipe 8.2).
- Register for Beanstalk by accessing the console and follow the on-screen instructions.

The application deployment will consist of four steps: * Create a Beanstalk application with `awscli` * Create a Beanstalk environment based on a Docker software stack (called *solution stack* in Beanstalk). * Create your Dockerfile and deploy it using the `eb` CLI.

All these steps can be done via the AWS console. In this recipe we chose to show a complete CLI based deployment. The output of the `awscli` calls however are truncated.

With the AWS CLI, create an application `foobar`, list the solution stacks and pick the Docker environment you need. Create a *configuration template* using the solution stack of your choice, and finally create an *environment*.

```
$ aws elasticbeanstalk create-application --application-name foobar
...
$ aws elasticbeanstalk list-available-solution-stacks
...
$ aws elasticbeanstalk create-configuration-template
                       --application-name foobar
                       --solution-stack-name="64bit Amazon Linux 2014.09 v1.2.1 running Docker 1.5
                       --template-name foo
...
$ aws elasticbeanstalk create-environment
                       --application-name foobar
                       --environment-name cookbook
                       --template-name foo
```

At this point, if you head over to the AWS Beanstalk console, you will see a *foobar* application and a *cookbook* environment being created. Once Beanstalk has finished creating the environment you can use the `describe-environments` API call and see that the environment is ready. In the console, you will also see that an EC2 instance, a security group and an elastic load-balancer has been created. You can configure the load-balancers through the Beanstalk console.

Going back to our CLI steps. Check the the environment is ready:

```
$ aws elasticbeanstalk describe-environments
-------------------------------------------------------------------------------------------------------------
|                                    DescribeEnvironments                                                   |
+-----------------------------------------------------------------------------------------------------------+
||                                        Environments                                                     ||
|+-------------------+-----------------------------------------------------------------------------------+|
||  ApplicationName  |  foobar                                                                           ||
||  CNAME            |  cookbook-pmpgzmx2e6.elasticbeanstalk.com                                         ||
||  DateCreated      |  2015-03-30T15:32:47.814Z                                                         ||
||  DateUpdated      |  2015-03-30T15:38:14.291Z                                                         ||
||  EndpointURL      |  awseb-e-7-AWSEBLoa-CUXDVD6RL9R7-992275618.eu-west-1.elb.amazonaws.com            ||
||  EnvironmentId    |  e-7hamntqqnw                                                                     ||
||  EnvironmentName  |  cookbook                                                                         ||
||  Health           |  Green                                                                            ||
||  SolutionStackName|  64bit Amazon Linux 2014.09 v1.2.1 running Docker 1.5.0                           ||
||  Status           |  Ready                                                                            ||
|+-------------------+-----------------------------------------------------------------------------------+|
|||                                          Tier                                                       |||
||+---------------------------------------+-------------------------------------------------+||
|||  Name                                 |  WebServer                                      |||
|||  Type                                 |  Standard                                       |||
|||  Version                              |                                                 |||
||+---------------------------------------+-------------------------------------------------+||
```

Once it is ready, you can push your Docker application to it. This is done the most easily by using the `eb` CLI which unfortunately is not included in the `awscli`. To finish the deployment, we will do the following steps:

- Install the `awsebcli`
- Create your Dockerfile
- Initialize the application *foobar* that you created earlier.
- List the environment to make sure you are using the *cookbook* environment created above.
- Deploy the application

Let's do this, install `awsebcli`, and create our application directory with our Dockerfile in it.

```
$ sudo pip install awsebcli
$ mkdir beanstalk
$ cd beanstalk
$ cat > Dockerfile
FROM ubuntu:12.04

RUN apt-get update
RUN apt-get install -y nginx zip curl
```

```
RUN echo "daemon off;" >> /etc/nginx/nginx.conf
RUN curl -o /usr/share/nginx/www/master.zip -L https://codeload.github.com/gabrielecirulli/2048/zi
RUN cd /usr/share/nginx/www/ && unzip master.zip && mv 2048-master/* . && rm -rf 2048-master maste

EXPOSE 80

CMD ["/usr/sbin/nginx", "-c", "/etc/nginx/nginx.conf"]
```

We can then use the `eb` CLI to initialize the application (using the application name used in the steps above, i.e *foobar*) and deploy it with `eb deploy`.

```
$ eb init foobar
$ eb list
* cookbook
$ eb deploy
Creating application version archive "app-150331_181300".
Uploading foobar/app-150331_181300.zip to S3. This may take a while.
Upload Complete.
INFO: Environment update is starting.
...
INFO: Successfully built aws_beanstalk/staging-app
INFO: Docker container ba7e79c37c43 is running aws_beanstalk/current-app.
INFO: New application version was deployed to running EC2 instances.
INFO: Environment update completed successfully.
```

You application is now deployed. Head over to the Beanstalk console and you will find the URL of the application. Click on the URL and it will open the *2048 game*. This is fronted by an elastic load-balancer, which means that increased load on the game will trigger the creation of additional instances serving the game behind the load-balancer.

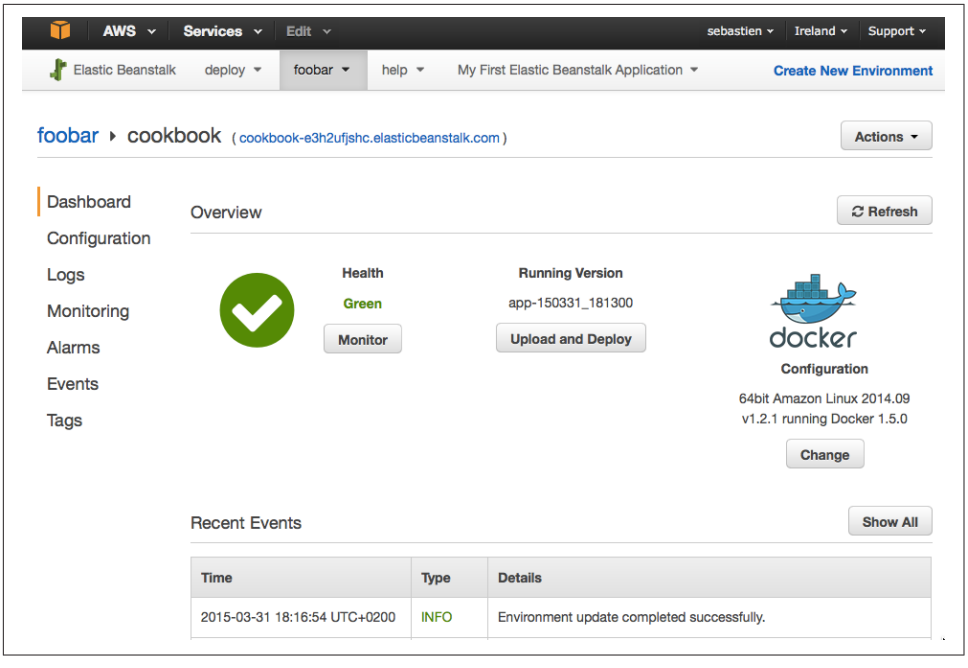


Figure 8-12. AWS Beanstalk Console

Discussion

In the example above our application was encapsulated in a single Dockerfile with no additional dependencies.

See Also

8.16 Using AWS Elastic Container Service as a Beanstalk Environment

Problem

Solution

Discussion

See Also

CHAPTER 9

Monitoring containers

This chapter will consist of several recipes focused on application and container monitoring as well as log management. The four recipes listed currently are only stubs, more will be added as the book nears completion. You can send me suggestions at *how2dock@gmail.com*

9.1 Getting Detailed Information About a Container With `docker inspect`

Problem

You want to get detailed information about a container. Details like when it was created, what command was passed to the container, what port mappings exists, what IP address the container has etc.

Solution

Use the `docker inspect` command. For example start a `nginx` container and use `inspect`:

```
$ docker run -d -p 80:80 nginx
$ docker inspect kickass_babbage
[{
    "AppArmorProfile": "",
    "Args": [
        "-g",
        "daemon off;"
    ],
...
        "ExposedPorts": {
```

```
            "443/tcp": {},
            "80/tcp": {}
        },
...
    "NetworkSettings": {
...
        "IPAddress": "172.17.0.3",
...
```

The inspect command also works on an image:

```
$ docker inspect nginx
[{
    "Architecture": "amd64",
    "Author": "NGINX Docker Maintainers \"docker-maint@nginx.com\"",
    "Comment": "",
    "Config": {
        "AttachStderr": false,
        "AttachStdin": false,
        "AttachStdout": false,
        "Cmd": [
            "nginx",
            "-g",
            "daemon off;"
        ],
...
```

Discussion

Docker `inspect` command takes a format option. You can use it to specify a go template and extract specific information about a container or image instead of getting the full json output.

```
$ docker inspect --help

Usage: docker inspect [OPTIONS] CONTAINER|IMAGE [CONTAINER|IMAGE...]

Return low-level information on a container or image

  -f, --format=""    Format the output using the given go template.
  --help=false       Print usage
```

For example to get the IP address of a running container and check its state.

```
$ docker inspect -f '' kickass_babbage
172.17.0.3
$ docker inspect -f '' kickass_babbage
true
```

If you prefer to use another Docker client like Docker-py (see Recipe 4.11) you can also access the detailed information about containers and images using standard Python dictionary notation:

```
$ python
...
>>> from docker import Client
>>> c=Client(base_url="unix://var/run/docker.sock")
>>> c.inspect_container('kickass_babbage')['State']['Running']
True
>>> c.inspect_container('kickass_babbage')['NetworkSettings']['IPAddress']
u'172.17.0.3'
```

9.2 Obtaining Usage Statistics of a Running Container

Problem

You have a running container on one of your Docker hosts and would like to monitor its resource usage (e.g Memory, CPU, Network).

Solution

Using the `docker stats` command. This new API endpoint was introduced on February 10th 2015 and is accessible in Docker 1.5. The usage syntax is quite simple, you just pass the container name (or container ID) to it and you will receive a stream of statistics. Below we start a Flask application container and run `stats` on it:

```
$ docker run -d -p 5000:5000 runseb/flask
$ docker stats dreamy_mccarthy
CONTAINER           CPU %               MEM USAGE/LIMIT       MEM %               NET I/O
dreamy_mccarthy     0.03%               24.01 MiB/1.955 GiB   1.20%               648 B/648 B
```

Since you are receiving a stream, you will not to Ctrl-C to kill the stream.

Discussion

Getting quick stats from the command line is quite useful for interactive debugging. However you will most likely want to collect all these statistics and aggregate them with a log collector solution like Logstash for futher visualiztion and analysis.

To prepare for such a monitoring framework you can try to use the `stats` API via `curl`, by issuing TCP requests to the Docker daemon. First you will need to configure your local Docker daemon to listen on port 2375 over TCP. On Ubuntu systems edit `/etc/default/docker` to include:

```
DOCKER_OPTS="-H tcp://127.0.0.1:2375"
```

Restart your Docker daemon with `sudo service docker restart`. You are now ready to use `curl` and target the Docker remote API. The syntax is again quite simple. It is a HTTP GET request to the `/containers/(id)/stats` URI. Try it like so:

```
$ $ docker -H tcp://127.0.0.1:2375 run -d -p 5001:5000 runseb/flask
$ curl http://127.0.0.1:2375/containers/agitated_albattani/stats
{"read":"2015-04-01T11:48:40.609469913Z","network":{"rx_bytes":648,"rx_packets":8,"...
```

DO not forget to replace `agitated_albattani` with the name of your container. You will start receiving a stream of statistics that you can interrupt with Ctrl-C. For practical purposes I truncated most of the results from the previous command. This is very handy to try things out, however if like me you like Python, you might want to access these statistics from a Python program. To do this you can use `docker-py` (see Recipe 4.11). A Python script like the one below will put you on the right track.

```
#!/usr/bin/env python

import json
import docker
import sys

cli=docker.Client(base_url='tcp://127.0.0.1:2375')
stats=cli.stats(sys.argv[1])
print json.dumps(json.loads(next(stats).rstrip('\n')),indent=4)
```

The `stats` object in the Python script above is a *generator* which yields results instead of the standard return behavior of functions. It uses to capture the statistics stream and pick up where it left off. `next(stats)` in the script is the way to yield the lates result from the stream.

See Also

- Original GitHub pull request for `stats`.
- API documentation

9.3 Listening to Docker Events on Your Docker Hosts

Problem

You want to monitor Docker events on your host. You are interested in image untagging and deletion and container life cycle events (e.g create, destroy, kill)

Solution

Use the `docker events` command. It will return a stream of events as they happen on your Docker host. The command takes a few optional arguments if you want to select events for a specific time range:

```
$ docker events --help

Usage: docker events [OPTIONS]

Get real time events from the server

  -f, --filter=[]    Provide filter values (i.e., 'event=stop')
  --help=false       Print usage
  --since=""         Show all events created since timestamp
  --until=""         Stream events until this timestamp
```

Hence while, `docker events` will work and block until your Ctrl-C the stream, you can use the `--since` or `--until` options like so:

```
$ docker events --since 1427890602
2015-04-01T12:17:04....9393146cb55e5bc9f04e20eb5a0622b3e26aae7: untag
2015-04-01T12:17:09....d5266f8777bfba4974ac56e3270e7760f6f0a81: untag
2015-04-01T12:17:22....d5266f8777bfba4974ac56e3270e7760f6f0a85: untag
2015-04-01T12:17:23....66f8777bfba4974ac56e3270e7760f6f0a81253: delete
2015-04-01T12:17:23....e9b5a793523481a0a18645fc77ad53c4eadsfa2: delete
2015-04-01T12:17:23....878585defcc1bc6f79d2728a13957871b345345: delete
```

Just as a reminder, you can get the current timestamp in epoch with `date +"%s"`

Discussion

This `events` command is also available as an API call and you can use `curl` to access it (see Recipe 9.2). Let's leave this as an exercise and give an example of using `docker-py` to get the list of events.

In Recipe 9.2, we reconfigured the Docker daemon to access the remote API over TCP. We can also use `docker-py` to access the unix docker socket. A sample Python script that would do this and save you some time to reconfigure the Docker daemon looks like this:

```
#!/usr/bin/env python
import json
import docker
import sys

cli=docker.Client(base_url='unix://var/run/docker.sock')
events=cli.events(since=sys.argv[1],until=sys.argv[2])
for e in events:
    print e
```

This scrip takes two timestamps as arguments and returns the events between these two. An example output would be:

```
$ ./events.py 1427890602 1427891476
{"status":"untag","id":"967a84db1eff36cab6e77fe9c9393146c...","time":1427890624}
{"status":"untag","id":"4986bf8c15363d1c5d15512d5266f8777...","time":1427890629}
{"status":"untag","id":"4986bf8c15363d1c5d15512d5266f8777...","time":1427890642}
{"status":"delete","id":"4986bf8c15363d1c5d15512d5266f877...","time":1427890643}
{"status":"delete","id":"ea13149945cb6b1e746bf28032f02e9b...","time":1427890643}
{"status":"delete","id":"df7546f9f060a2268024c8a230d86398...","time":1427890643}
```

Event based tools likehttp://stackstorm.com[StackStorm] take advantage of this to orchestrate various parts of a Docker base infrastructure.

See Also

- API documentation

9.4 Getting The Logs of a Container With `docker logs`

Problem

You have a running container, it runs a process in the foreground within the container, you would like to access the process logs from the host.

Solution

Use the `docker logs` command.

For example start an `nginx` container and open your browser on port 80 of the Docker host:

```
$ docker run -d -p 80:80 nginx
$ docker ps
CONTAINER ID    IMAGE           ...  PORTS                          NAMES
dd0e926c4015    nginx:latest    ...  443/tcp, 0.0.0.0:80->80/tcp    gloomy_mclean
$ docker logs gloomy_mclean
192.168.34.1 - - [10/Mar/2015:10:12:35 +0000] "GET / HTTP/1.1" 200 612 "-" ...
...
```

Discussion

You can get a continuous log stream using the `-f` option.

```
$ docker logs -f gloomy_mclean
192.168.34.1 - - [10/Mar/2015:10:12:35 +0000] "GET / HTTP/1.1" 200 612 "-" ...
...
```

In addition you can also monitor the process running in the container with `docker top`

```
$ docker top gloomy_mclean
UID         PID         PPID      ...    CMD
root        5605        4732      ...    nginx: master process nginx -g daemon off;
syslog      5632        5605      ...    nginx: worker process
```

9.5 Using Logspout to Collect Container Logs

Problem

Container logs can be obtained from `docker logs` as seen in Recipe 9.4, but you would like to collect these logs from containers running in multiple Docker hosts and aggregate them.

Solution

Use `logspout`. Logspout can collect logs from all containers running on a host and route them to another host. It runs as a container and is purely stateless. You can use it to route logs to a syslog server or send it to logstash for processing.

Let's install `logspout` on one Docker host to collect logs from a `nginx` container. We run `nginx` on port 80 of the host. Start `logspout`, mount the Docker unix socket `/var/run/docker.sock` in `/tmp/docker.sock` and specify a syslog endpoint (here we use another Docker host with the IP address of 192.168.34.11)

```
$ docker pull nginx
$ docker pull gliderlabs/logspout
$ docker run -d --name webserver -p 80:80 nginx
$ docker run -d --name logspout -v /var/run/docker.sock:/tmp/docker.sock \
            gliderlabs/logspout syslog://192.168.34.11:5000
```

To collect the logs we are going to use a `logstash` container running at 192.168.34.11. To simplify things, it will listen for syslog input on UDP port 5000 and output everything to stdout on the same host. Start by pulling the logstash image (We use the image `ehazlett/logstash` but there are many logstash images that you might want to consider). After pulling the image we are going to build our own and specify a custom logstash configuration file (this is based on the `/etc/logstash.conf.sample` from the +ehazlett/logstash image).

```
$ docker pull ehazlett/logstash
$ cat logstash.conf
input {
  tcp {
    port => 5000
    type => syslog
  }
}

filter {
```

```
  if [type] == "syslog" {
    grok {
      match => { "message" => "%{SYSLOGTIMESTAMP:syslog_timestamp} \
          %{SYSLOGHOST:syslog_hostname} %{DATA:syslog_program}(?:\[%{POSINT:syslog_pid}\])?: \
          %{GREEDYDATA:syslog_message}" }
      add_field => [ "received_at", "%{@timestamp}" ]
      add_field => [ "received_from", "%{host}" ]
    }
    syslog_pri { }
    date {
      match => [ "syslog_timestamp", "MMM  d HH:mm:ss", "MMM dd HH:mm:ss" ]
    }
  }
}

output {
  stdout { codec => rubydebug }
}
$ cat Dockerfile
FROM ehazlett/logstash

COPY logstash.conf /etc/logstash.conf
ENTRYPOINT ["/opt/logstash/bin/logstash"]
$ docker build -t logstash .
```

You are now ready to run the logstash container, and bind port 5000 of the container to port 5000 of the host listening for UDP traffic.

```
$ docker run -d --name logstash -p 5000:5000/udp log -f /etc/logstash.conf
```

Once you open your browser to access Nginx running on the first Docker host you used, logs will appear in the logstash container:

```
$ docker logs logstash
...
{
  "message" => "<14>2015-03-10T13:00:39Z 889bbf0753a8 nginx[1]: 192.168.34.1 - \
               - [10/Mar/2015:13:00:39 +0000] \"GET / HTTP/1.1\" 200 612 \"-\" \"Mozilla/5.0 \
                           (Macintosh; Intel Mac OS X 10_8_5) AppleWebKit/600.3.18 (KHTML, like Ge
               Version/6.2.3 Safari/537.85.12\" \"-\"\n",
               "@version" => "1",
               "@timestamp" => "2015-03-10T13:00:36.241Z",
                    "type" => "syslog",
                    "host" => "192.168.34.10",
                    "tags" => [
...
```

Discussion

To simplify testing `logspout` with `logstash` you can clone the repository accompanying this book and go to the `ch09/logspout` directory. A Vagrantfile will start two Docker hosts and pull the required Docker images on each host.

```
$ git clone https://github.com/how2dock/docbook.git
$ vagrant up
$ vagrant status
Current machine states:

w                         running (virtualbox)
elk                       running (virtualbox)
...
```

On the *web server* node, you can run `nginx` and the `logspout` container. On the *elk* node you can run the `logstash` container.

```
$ vagrant ssh w
$ docker run --name nginx -d -p 80:80 nginx
$ docker run -d --name logspout -v /var/run/docker.sock:/tmp/docker.sock \
            gliderlabs/logspout syslog://192.168.34.11:5000
```

```
$ vagrant ssh elk
$ cd /vagrant
$ docker build -t logstash .
$ docker run -d --name logstash -p 5000:5000/udp log -f /etc/logstash.conf
```

You should see your `nginx` logs in the `logstash` container. Experiment with more hosts, different containers and play with the logstash plugins to store your logs in different formats.

See Also

- logstash website.
- Configuration of logstash.
- Plugins for logstash *inputs*, *outputs*, *codecs* and *filters*

9.6 Managing `logspout` Routes to Store Container Logs

Problem

You are using `logspout` to stream your logs to a remote server, but you would like to modify this endpoint. Specifically, you want to debug your containers by looking directly at logspout, change the endpoint it uses or add more endpoints.

Solution

In Recipe 9.5, you might have noticed that the logspout container has port 8000 exposed. You can use this port to manage routes via a straightfoward HTTP API.

You can bind port 8000 to the host to access this API remotely, but as an exercise we are going to use a linked container to do it locally. Pull an image that contains `curl`

and start a container interactively. Verify that you can ping the `logspout` container (Here I assume that we have the same setup that in Recipe 9.5). Then use curl to access the `logspout` API at `http://logspout:8000`.

```
$ docker pull tutum/curl
$ docker run -ti --link logspout:logspout tutum/curl /bin/bash
root@c94a4eacb7cc:/# ping logspout
PING logspout (172.17.0.10) 56(84) bytes of data.
64 bytes from logspout (172.17.0.10): icmp_seq=1 ttl=64 time=0.075 ms
...
root@c94a4eacb7cc:/# curl http://logspout:8000/logs
        logspout|[martini] Started GET /logs for 172.17.0.12:38353
        nginx|192.168.34.1 - - [10/Mar/2015:13:57:38 +0000] "GET / HTTP/1.1" 200  ...
        nginx|192.168.34.1 - - [10/Mar/2015:13:57:43 +0000] "GET / HTTP/1.1" 200  ...
```

Discussion

To manage the log streams the API exposes a `/routes` route. The standard HTTP verbs GET, DELETE and POST can be used to list, delete and update the streaming endpoints.

```
root@1fbb2f9636a8:/# curl http://logspout:8000/routes
[
  {
    "id": "e508de0c9689",
    "target": {
      "type": "syslog",
      "addr": "192.168.34.11:5000"
    }
  }
]
root@1fbb2f9636a8:/# curl http://logspout:8000/routes/e508de0c9689
{
  "id": "e508de0c9689",
  "target": {
    "type": "syslog",
    "addr": "192.168.34.11:5000"
  }
}
root@1fbb2f9636a8:/# curl -X DELETE http://logspout:8000/routes/e508de0c9689
root@1fbb2f9636a8:/# curl http://logspout:8000/routes
[]
root@1fbb2f9636a8:/# curl -X POST \
                          -d '{"target": {"type": "syslog", \
                                          "addr": "192.168.34.11:5000"}}' \
                          http://logspout:8000/routes
{
  "id": "f60d30502654",
  "target": {
    "type": "syslog",
    "addr": "192.168.34.11:5000"
```

```
  }
}
root@1fbb2f9636a8:/# curl http://logspout:8000/routes
[
  {
    "id": "f60d30502654",
    "target": {
      "type": "syslog",
      "addr": "192.168.34.11:5000"
    }
  }
]
```

You can create a route to Papertrail which provides automatic-backup to Amazon S3.

9.7 Using Elasticsearch and Kibana to Store and Visualize Container Logs

Problem

In Recipe 9.5 we only used logstash to receive logs and we sent them to stdout. However logstash many plugins allow you to do much more. You would like to go further and use elasticsearch to store your container logs.

Solution

Start an elasticsearch and a Kibana container. Kibana is a dashboard that allows you to easily visualize and query your elasticsearch indexes. Start a logstash container using the default configuration from the `ehazlett/logstash` image.

```
$ docker run --name es -d -p 9200:9200 -p 9300:9300 ehazlett/elasticsearch
$ docker run --name kibana -d -p 80:80 ehazlett/kibana
$ docker run -d --name logstash -p 5000:5000/udp \
             --link es:elasticsearch ehazlett/logstash \
             -f /etc/logstash.conf.sample
```

Notice that the logstash container is linked to the elasticsearch container. If you do not link it, logstash will not be able to find the elasticsearch server.

With the container running, you can open your browser on port 80 of the Docker host where you are running the Kibana container. You will see the Kibana default dashboard. Select *1. Sample Dashboard*, this will extract some information from your index and build a basic dashboard. You should see the logs obtained from hitting the Nginx server, below is a sample snapshot:

Figure 9-1. Snapshot of a Kibana dashboard obtained with this recipe.

Discussion

In the solution above, elasticsearch is running on a single container. The index created when storing your logs streamed by logspout will not persist if you kill and remove the elasticsearch container. Consider mounting a volume and backing it up to persist your elasticsearch data. In addition, if you need more storage and an efficient index, you should create an elasticsearch cluster across multiple Docker hosts.

9.8 Using Collectd to Visualize Container Metrics

Problem

In addition to visualizing application logs (see Recipe 9.7), you would like to monitor container metrics such as CPU.

Solution

Use Collectd. Run it in a container on all hosts where you have running containers that you want to monitor. By mounting the `/var/run/docker.sock` socket in a `collectd` container you can use a `collectd` plugin that uses the Docker stats API (see Recipe 9.2) and sends metrics to a Graphite dashboard running in a different host.

This is an advanced recipe that uses several concepts covered earlier. Make sure to do Recipe 7.1 and Recipe 9.7 before doing this recipe.

To test this, we are going to use the following setup, with two Docker hosts. One runs four containers. A `nginx` container used to generate dummy logs to `stdout`, a `logspout` container that will route all `stdout` logs to a logstash instance, one that generates synthetic load (i.e `borja/unixbench`) and one `collectd` container. These four containers can be started using Docker compose.

The other host runs four containers as well. A `logstash` container to collect the logs coming from `logspout`, an elasticsearch container to store the logs, a Kibana container to visualize those logs and a `graphite` container. The `graphite` container also runs `carbon` to store the metrics.

The following diagram illustrates this two hosts, eight containers setup:

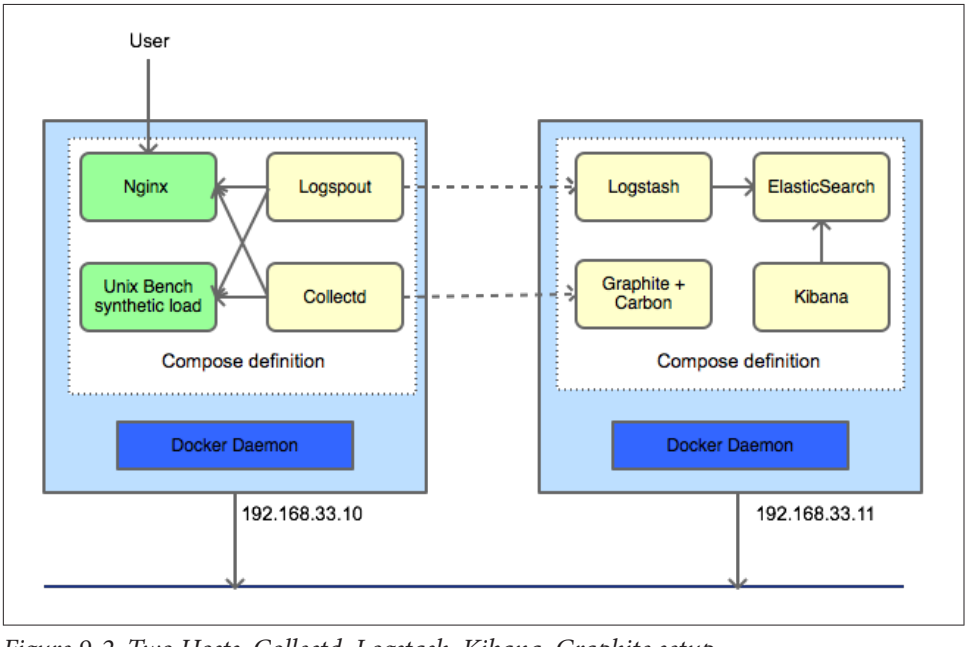


Figure 9-2. Two Hosts, Collectd, Logstash, Kibana, Graphite setup.

On the first host (the worker), you can start all the containers with Docker compose (see Recipe 7.1) using a YAML file like this one:

```
nginx:
  image: nginx
  ports:
   - 80:80
logspout:
  image: gliderlabs/logspout
  volumes:
   - /var/run/docker.sock:/tmp/docker.sock
  command: syslog://192.168.33.11:5000
```

```
collectd:
  build: .
  volumes:
   - /var/run/docker.sock:/var/run/docker.sock
load:
  image: borja/unixbench
```

The `logspout` container uses a command that specifies your `logstash` enpoint. Change the IP above if you are running in a different networking environment. The `collectd` container is built by Docker compose and based on the following Dockerfile:

```
FROM debian:jessie

RUN apt-get update && apt-get -y install \
    collectd \
    python \
    python-pip
RUN apt-get clean
RUN pip install docker-py

RUN groupadd -r docker && useradd -r -g docker docker

ADD docker-stats.py /opt/collectd/bin/docker-stats.py
ADD docker-report.py /opt/collectd/bin/docker-report.py
ADD collectd.conf /etc/collectd/collectd.conf

RUN chown -R docker /opt/collectd/bin

CMD ["/usr/sbin/collectd","-f"]
```

In the discussion section of this recipe, we will go over the scripts used in this Dockerfile.

On the second host (the monitor), you can start all containers with Docker compose (see Recipe 7.1) using a YAML file like this one:

```
es:
  image: ehazlett/elasticsearch
  ports:
   - 9300:9300
   - 9200:9200
kibana:
  image: ehazlett/kibana
  ports:
   - 8080:80
graphite:
  image: hopsoft/graphite-statsd
  ports:
   - 80:80
   - 2003:2003
   - 8125:8125/udp
```

```
logstash:
  image: ehazlett/logstash
  ports:
   - 5000:5000
   - 5000:5000/udp
  volumes:
   - /root/docbook/ch09/collectd/logstash.conf:/etc/logstash.conf
  links:
   - es:elasticsearch
  command: -f /etc/logstash.conf
```

Several non-official images are being used in this setup. `glider labs/logspout`, `borja/unixbench`, `ehazlett/elasticsearch`, `ehazlett/kibana`, `ehazlett/logstash` and `hopsoft/graphite-statsd`. Check the Dockerfile of these images on Docker Hub or build your own images if you do not trust them.

Once all the containers are up on the two hosts and assuming that you setup the networking and any firewall that may exists properly (open ports on security groups if you are using cloud instances), you will be able to access the `nginx` container on port 80 of the worker host, the Kibana dashboard on port 8080 of the monitor host and the graphite dashboard on port 80 of the monitor host.

The graphite dashboard will show you basic CPU metric coming from all the containers running on the worker host, See the snapshot below for what you should see:

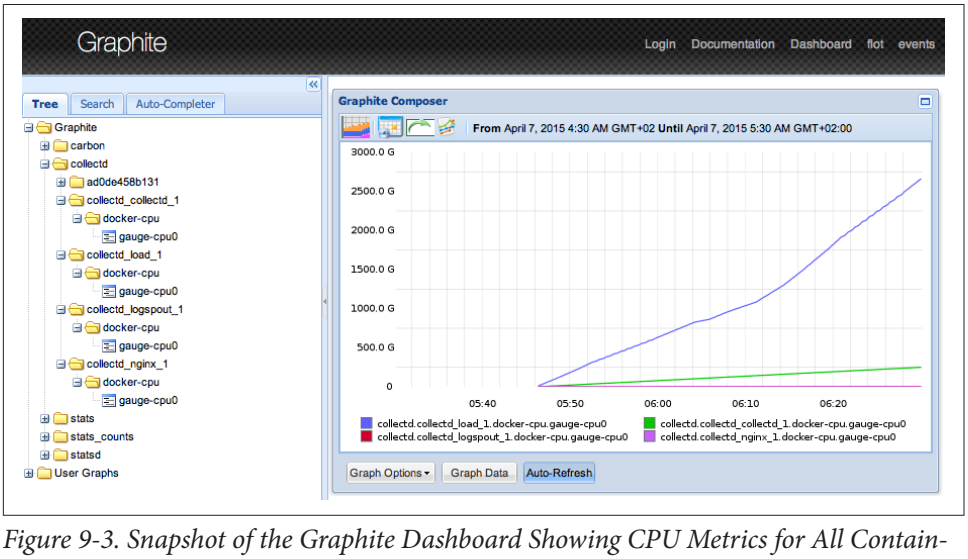


Figure 9-3. Snapshot of the Graphite Dashboard Showing CPU Metrics for All Containers.

Discussion

You can get all the scripts used in this recipe using the on-line material coming with this book. Clone the repository if you have not done so already and head over to the `docbook/ch09/collectd` directory.

```
$ git clone https://github.com/how2dock/docbook.git
$ cd docbook/ch09/collectd
$ tree
.
├── Dockerfile
├── README.md
├── Vagrantfile
├── collectd.conf
├── docker-report.py
├── docker-stats.py
├── logstash.conf
├── monitor.yml
└── worker.yml
```

The Vagrantfile will allow you to start two Docker hosts on your local machine to experiment with this setup. However you can clone this repository in two cloud instances that have Docker and Docker compose installed and then start all the containers. If you use Vagrant do:

```
$ vagrant up
$ vagrant ssh monitor
$ vagrant ssh worker
```

While using Vagrant for this recipe I encountered several intermittent errors as well as delays when downloading the images. Using cloud instances with better network connectivity might be more enjoyable.

The two YAML files are used to easily start all containers on the two hosts. Do not run them on the same host.

```
$ docker-compose -f monitor.yml up -d
$ docker-compose -f worker.yml up -d
```

The `logstash.conf` file was discussed in Recipe 9.5. Go back to this recipe if you do not understand this configuration file.

The Dockerfile is used to build a `collectd` image and was shown in the solution section earlier. It is based on a Debian Jessie image and installs Docker-py (see Recipe 4.11) and a few other scripts.

Collectd makes uses of plugins to collect metrics and send them to a data store (e.g Carbon with Graphite). In this setup we use the simplest form of `collectd` plugin

which is called an `exec` plugin. This is defined in the `collectd.conf` file in the section:

```
<Plugin exec>
   Exec "docker" "/opt/collectd/bin/docker-stats.py"
   NotificationExec "docker" "/opt/collectd/bin/docker-report.py"
</Plugin>
```

The collectd process running in the foreground in the `collectd` container will routinely execute the two Python scripts defined in the configuration file. This is also why we copy them in the Dockerfile. The `docker-report.py` script, simply outputs values to syslog. This has the benefit that we will also collect them via our `logspout` container and see them in our Kibana dashboard. The `docker-stats.py` script makes use of the Docker stats API (see Recipe 9.2) and the `docker-py` Python package. This script lists all the running containers, for obtains the statistics for them. For the stats called `cpu_stats` it writes a so-called *PUTVAL* string to stdout. This string is understood by collectd and sent to the Graphite data store (a.k.a Carbon) for storage and visualization. The *PUTVAL* string follows the Collectd `exec` plugin syntax.

```
#!/usr/bin/env python

import random
import json
import docker
import sys

cli=docker.Client(base_url='unix://var/run/docker.sock')

types = ["gauge-cpu0"]

for h in cli.containers():
    if not h["Status"].startswith("Up"):
        continue
    stats = json.loads(cli.stats(h["Id"]).next())
    for k, v in stats.items():
        if k == "cpu_stats":
            print("PUTVAL %s/%s/%s N:%s" % (h['Names'][0].lstrip('/'), \
                                'docker-cpu', types[0], v['cpu_usage']['total_usage']))
```

The example plugin in this recipe is very minimal and the statistics need to be processed further. You might want to consider using this Python based plugin instead.

See Also

- Collectd website

- Collectd Exec plugin
- Graphite website
- Logstash website
- Collectd Docker plugin

9.9 Accessing Container Logs Through Mounted Volumes

Problem

Solution

Discussion

9.10 Using cAdvisor to Monitor Resource Usage in Containers

Problem

While `logspout` (see Recipe 9.5) allows you to stream application logs to remote endpoints, you need a resource utilization monitoring system.

Solution

Use *cAdvisor*, it was created by Google to monitor resource usage and performance of their lcmtfy containers. cAdvisor runs as a container on your Docker hosts, by mounting local volumes it can monitor the performance of all other running containers on that same host. It provides a local web UI, exposes an API and can stream data to InfluxDB. Streaming data from running containers to a remote InfluxDB cluster allows you to aggregate performance metrics for all your containers running in a cluster.

To get started, let's use a single host. Download the *cAdvisor* image as well as *borja/unixbench* an image that will allows us to simulate a workload inside a container.

```
$ docker pull google/cadvisor:latest
$ docker pull borja/unixbench
$ docker run -v /var/run:/var/run:rw\
             -v /sys:/sys:ro \
             -v /var/lib/docker/:/var/lib/docker:ro \
             -p 8080:8080 \
             -d \
             --name cadvisor \
```

```
                google/cadvisor:latest
$ docker run -d borja/unixbench
```

With the two containers running, you can open your browser at `http://<IP_DOCKER_HOST>:8080` and you will enjoy the cAdvisor UI. You will be able to browse the running containers and access metrics for each of them.

Figure 9-4. Snapshot of the cAdvisor UI.

Discussion

See Also

- cAdvisor API documentation

9.11 Monitoring Container Metrics With InfluxDB, Grafana and cAdvisor

Problem

Solution

```
$ docker run -d -p 8083:8083 -p 8086:8086 \
             -e PRE_CREATE_DB="db" \
                         --name influxdb \
                         tutum/influxdb:latest
$ docker run -d -p 80:80 \
             --link=influxdb:influxdb \
                         -e HTTP_USER=admin \
                         -e HTTP_PASS=root \
                         -e INFLUXDB_HOST=influxdb \
                         -e INFLUXDB_NAME=db \
                         --name=grafana \
                         tutum/grafana
$ docker run -v /var/run:/var/run:rw \
             -v /sys:/sys:ro \
                         -v /var/lib/docker/:/var/lib/docker:ro \
                         -p 8080:8080 \
                         --link=influxdb:influxdb \
                         -d --name=cadvisor \
                         google/cadvisor:latest \
                         -storage_driver=influxdb \
                         -storage_driver_host=influxdb:8086 \
                         -storage_driver_db=db
$ docker run -d borja/unixbench
```

Discussion

9.12 Gaining Visibility Into Your Containers Layout with Weavescope

Problem

Building a distributed application based on a microservices architecture will lead to hundreds and potentially more containers running in your data center. Visibility into that application and all the containers that it will be made of will be crucial and a key part of your overall infrastructure.

Solution

Weavescope from Weaveworks provides you with a simple yet powerful way of probing your infrastructure and dynamically create a map of all your containers. It gives you multiple views: per container, per image, per host and per application, allows you to do grouping of containers and drill down on their characteristics.

It is open source and available on GitHub.

To ease testing I prepared a Vagrant box similar to many other recipes in this book. Clone the repository with Git and launch the Vagrant box like so:

```
$ git clone https://github.com/how2dock/docbook.git
$ cd how2dock/ch09/weavescope
$ vagrant up
```

The Vagrant box installs the latest Docker version (i.e 1.6.2 as of this writing) and also installs Docker compose (see <<>>). In the `/vagrant` folder you will find a *compose* file that gives you a synthetic three tiered application made of two load balancers, two application containers and three databases containers. This is a toy application meant to illustrate Weavescope. Once the VM has booted, `ssh` into it, go to the `/vagrant` folder, launch `compose` and the weavescope script (i.e `scope`) like so:

```
$ vagrant ssh
$ cd /vagrant
$ docker-compose up -d
$ ./scope launch
```

You will end up with eight containers running. Seven for the tiered toy application and one for weavescope. The toy application is accessible at `http://192.168.33.10:8001` or `http://192.168.33.10:8002`. Of course the most interesting part is the weavescope dashboard. Open your browser at `http://192.168.33.10:4040` and you will see something similar to the snapshot below.

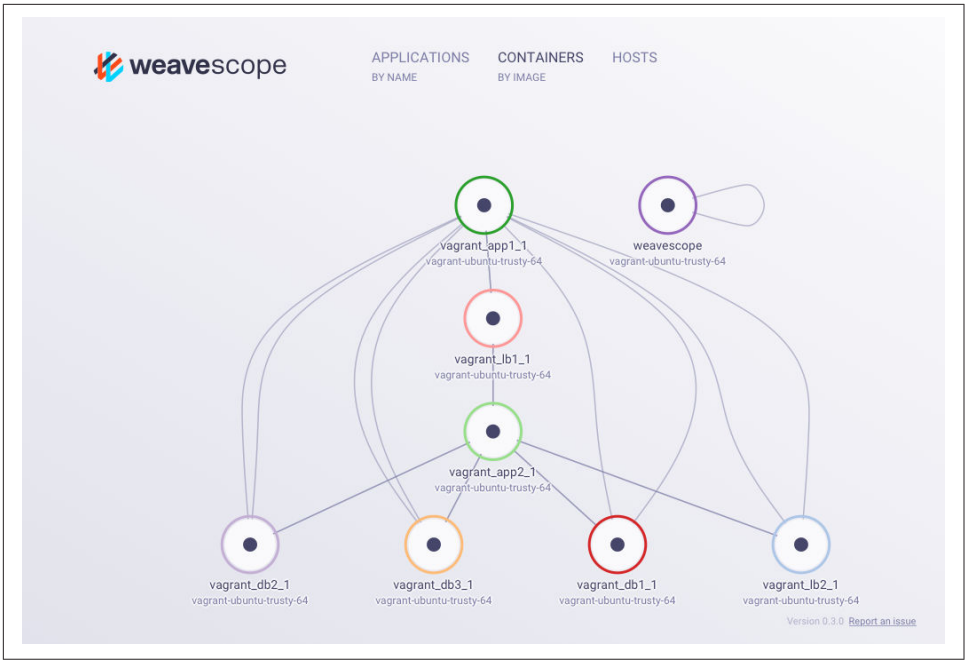


Figure 9-5. Snapshot of the Weavescope Dashboard.

Navigate through the UI, explore the various groupings capabilities and explore the information of each container.

Discussion

Weavescope is still in early development and considered pre-alpha as of this writing. You should expect more features to be added to this open source product. Definitely worth keeping an eye on this visibility solution for Docker containers.

Building from source is straightforward with a Makefile that builds a Docker image.

See Also

- Detect, Map and Monitor Docker containers with weavescope

9.13 Monitoring a Kubernetes Cluster with Heapster

Problem

Solution

Discussion