Community Experience Distilled

# Orchestrating Docker

Manage and deploy Docker services to containerize applications efficiently

Shrikrishna Holla

[PACKT] open source* PUBLISHING
community experience distilled

# Orchestrating Docker

Manage and deploy Docker services to containerize applications efficiently

**Shrikrishna Holla**

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

# Orchestrating Docker

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: January 2015

Production reference: 1190115

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

ISBN 978-1-78398-478-7

www.packtpub.com

# Credits

**Author**
Shrikrishna Holla

**Reviewers**
Amit Mund
Taichi Nakashima
Tommaso Patrizi

**Acquisition Editor**
Larissa Pinto

**Content Development Editor**
Parita Khedekar

**Technical Editor**
Tanmayee Patil

**Copy Editor**
Vikrant Phadke

**Project Coordinator**
Neha Thakur

**Proofreaders**
Simran Bhogal
Maria Gould
Ameesha Green
Paul Hindle

**Indexer**
Mariammal Chettiyar

**Graphics**
Abhinash Sahu

**Production Coordinator**
Komal Ramchandani

**Cover Work**
Komal Ramchandani

# About the Author

**Shrikrishna Holla** is a full-stack developer based in Bangalore and Chennai, India. He loves biking, listening to music, and occasionally, sketching. You can find him frequently in hackathons, wearing a hoodie and sipping Red Bull, preparing for an all-nighter.

He currently works as a product developer for Freshdesk, a cloud-based customer support platform.

You can get in touch with him on Twitter (`@srikrishnaholla`) or find him at the Docker IRC channel (`#docker` on `Freenode`) with the `shrikrishna` handle.

---

I would like to thank the creators of Docker, without whom this book wouldn't have seen the light of the day. To my editors, Parita and Larissa, it has been a long journey, but you have been extremely supportive and helpful week after week. To my parents, you have been, are, and will always be my inspiration—the final ray of light in the darkest dungeon. To my sisters, for the soothing words of advice whenever I've had the blues. To all my teachers, who helped me to choose my path. To my friends, who help me forget all my worries. To the countless people who have given me encouragement, suggestions, and feedback, I couldn't have done this without you. To my readers, thank you for trusting me with your learning.

---

# About the Reviewers

**Amit Mund** has been working on Linux and other technologies for automation and infrastructure monitoring since 2004. He is currently associated with Akamai Technologies. He has previously worked for website-hosting teams at Amazon and Yahoo!

---

I would like to thank my family, my mentors from Bhawanipatna, and my friends and colleagues for helping me in my learning and development throughout my professional career.

---

**Taichi Nakashima** is a Tokyo-based web developer and software engineer. He is also a blogger and he loves Docker, Golang, and DevOps. Taichi is also an OSS contributor. You can find his contributions at `https://github.com/tcnksm`.

**Tommaso Patrizi** is Docker fan who used the technology since its first release and had machines in production with Docker since Version 0.6.0. He has planned and deployed a basic private PaaS with Docker and Open vSwitch.

Tommaso is an enthusiastic Ruby and Ruby on Rails programmer. He strives for simplicity, which he considers to be the perfect synthesis between code effectiveness, maintainability, and beauty. He is currently learning the Go language.

Tommaso is a system administrator. He has broad knowledge of operating systems (Microsoft, Linux, OSX, SQL Server, MySql, PostgreSQL, and PostGIS), virtualization, and the cloud (vSphere, VirtualBox, and Docker).

# www.PacktPub.com

## Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.


`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

### Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

### Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

# Table of Contents

# Preface

Get started with Docker, the Linux containerizing technology that has revolutionized application sandboxing. With this book, you will be able to learn how to use Docker to make your development faster and your deployment of applications simpler.

This guide will show you how to build your application in sandboxed Docker containers and make them run everywhere—your development machine, your private server, or even on the cloud, with a fraction of the cost of a virtual machine. Build a PaaS, deploy a cluster, and so on, all on your development setup.

## What this book covers

*Chapter 1*, *Unboxing Docker*, teaches you how to get Docker running in your environment.

*Chapter 2*, *Docker CLI and Dockerfile*, helps you to acclimatize to the Docker command-line tool and start building your own containers by writing Dockerfiles.

*Chapter 3*, *Configuring Docker Containers*, shows you how to control your containers and configure them to achieve fine-grained resource management.

*Chapter 4*, *Automation and Best Practices*, covers various techniques that help manage containers—co-ordinating multiple services using supervisor, service discovery, and knowledge about Docker's security.

*Chapter 5*, *Friends of Docker*, shows you the world surrounding Docker. You will be introduced to open source projects that use Docker. Then you can build your own PaaS and deploy a cluster using CoreOS.

# What you need for this book

This book expects you to have used Linux and Git before, but a novice user will find no difficulty in running the commands provided in the examples. You need to have an administrative privilege in the user account of your operating system in order to install Docker. Windows and OSX users will need to install VirtualBox.

# Who this book is for

Whether you are a developer or a sysadmin, or anything in between, this book will give you the guidance you need to use Docker to build, test, and deploy your applications and make them easier, even enjoyable.

Starting from the installation, this book will take you through the different commands you need to know to start Docker containers. Then it will show you how to build your own application and take you through instructions on how to fine-tune the resource allocations to those containers, before ending with notes on managing a cluster of Docker containers.

By sequentially working through the steps in each chapter, you will quickly master Docker and be ready to ship your applications without needing to spend sleepless nights for deployment.

# Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can set environment variables with the `ENV` directive."

A block of code is set as follows:

```
WORKDIR code.it
RUN     git submodule update --init --recursive
RUN     npm install
```

Any command-line input or output is written as follows:

```
$ docker run --d -p '8000:8000' -e 'NODE_PORT=8000' -v
'/var/log/code.it:/var/log/code.it' shrikrishna/code.it .
```

**New terms** and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Go to **Settings** in your repository."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

# Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

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

# Customer support

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

## Downloading the example code

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

# Downloading the color images of this book

We also provide you a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from: `https://www.packtpub.com/sites/default/files/downloads/B02634_4787OS_Graphics.pdf`

# Errata

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

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

# Piracy

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

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

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

# Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

# 1
# Unboxing Docker

**Docker** is a lightweight containerization technology that has gained widespread popularity in recent years. It uses a host of the Linux kernel's features such as namespaces, cgroups, AppArmor profiles, and so on, to sandbox processes into configurable virtual environments.

In this chapter, you will learn how to install Docker on various systems, both in development and in production. For Linux-based systems, since a kernel is already available, installation is as simple as the `apt-get install` or `yum install` commands. However, to run Docker on non-Linux operating systems such as OSX and Windows, you will need to install a helper application developed by Docker Inc., called **Boot2Docker**. This will install a lightweight Linux VM on **VirtualBox**, which will make Docker available through port 2375, assigned by the **Internet Assigned Numbers Authority** (**IANA**).

At the end of this chapter, you will have installed Docker on your system, be it in development or production, and verified it.

This chapter will cover the following points:

- Introducing Docker
- Installing Docker
- Ubuntu (14.04 and 12.04)
- Mac OSX and Windows
- OpenStack
- Inception: building Docker in Docker
- Verifying installation: `Hello World` output
- Introducing Docker

Docker was developed by DotCloud Inc. (Currently Docker Inc.), as the framework they built their **Platform as a Service** (**PaaS**) upon. When they found increasing developer interest in the technology, they released it as open source and have since announced that they will completely focus on the Docker technology's development, which is good news as it means continual support and improvement for the platform.

There have been many tools and technologies aimed at making distributed applications possible, even easy to set up, but none of them have as wide an appeal as Docker does, which is primarily because of its cross-platform nature and friendliness towards both system administrators and developers. It is possible to set up Docker in any OS, be it Windows, OSX, or Linux, and Docker containers work the same way everywhere. This is extremely powerful, as it enables a write-once-run-anywhere workflow. Docker containers are guaranteed to run the same way, be it on your development desktop, a bare-metal server, virtual machine, data center, or cloud. No longer do you have the situation where a program runs on the developer's laptop but not on the server.

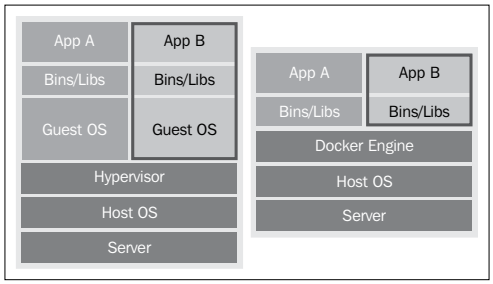
The nature of the workflow that comes with Docker is such that developers can completely concentrate on building applications and getting them running inside the containers, whereas sysadmins can work on running the containers in deployment. This separation of roles and the presence of a single underlying tool to enable it simplifies the management of code and the deployment process.

But don't virtual machines already provide all of these features?

**Virtual Machines** (**VMs**) are fully virtualized. This means that they share minimal resources amongst themselves and each VM has its own set of resources allocated to it. While this allows fine-grained configuration of the individual VMs, minimal sharing also translates into greater resource usage, redundant running processes (an entire operating system needs to run!), and hence a performance overhead.

Docker, on the other hand, builds on a container technology that isolates a process and makes it believe that it is running on a standalone operating system. The process still runs in the same operating system as its host, sharing its kernel. It uses a layered copy-on-write filesystem called **Another Unionfs** (**AUFS**), which shares common portions of the operating system between containers. Greater sharing, of course, can only mean less isolation, but vast improvements in Linux process's resource management solutions such as namespaces and cgroups have allowed Docker to achieve VM-like sandboxing of processes and yet maintain a very small resource footprint.

Let's take a look at the following image:




This a Docker vs VM comparison. Containers share the host's resources with other containers and processes, and virtual machines have to run an entire operating system for every instance.

# Installing Docker

Docker is available in the standard repositories of most major Linux distributions. We will be looking at the installation procedures for Docker in Ubuntu 14.04 and 12.04 (Trusty and Precise), Mac OSX, and Windows. If you are currently using an operating system not listed above, you can look up the instructions for your operating system at `https://docs.docker.com/installation/#installation`.

# Installing Docker in Ubuntu

Docker is supported by Ubuntu from Ubuntu 12.04 onwards. Remember that you still need a 64-bit operating system to run Docker. Let's take a look at the installation instructions for Ubuntu 14.04.

## Installing Docker in Ubuntu Trusty 14.04 LTS

Docker is available as a package in the Ubuntu Trusty release's software repositories under the name of `docker.io`:

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

That's it! You have now installed Docker onto your system. However, since the command has been renamed `docker.io`, you will have to run all Docker commands with `docker.io` instead of `docker`.

The package is named `docker.io` because it conflicts with another KDE3/GNOME2 package called `docker`. If you rather want to run commands as `docker`, you can create a symbolic link to the `/usr/local/bin` directory. The second command adds autocomplete rules to bash:

```
$ sudo ln -s /usr/bin/docker.io /usr/local/bin/docker
$ sudo sed -i '$acomplete -F _docker docker' \
> /etc/bash_completion.d/docker.io
```

## Installing Docker in Ubuntu Precise 12.04 LTS

Ubuntu 12.04 comes with an older kernel (3.2), which is incompatible with some of the dependencies of Docker. So we will have to upgrade it:

```
$ sudo apt-get update
$ sudo apt-get -y install linux-image-generic-lts-raring linux-headers-generic-lts-raring
$ sudo reboot
```

The kernel that we just installed comes with AUFS built in, which is also a Docker requirement.

Now let's wrap up the installation:

```
$ curl -s https://get.docker.io/ubuntu/ | sudo sh
```

This is a `curl` script for easy installation. Looking at the individual pieces of this script will allow us to understand the process better:

1. First, the script checks whether our **Advanced Package Tool** (**APT**) system can deal with `https` URLs, and installs `apt-transport-https` if it cannot:

   ```
   # Check that HTTPS transport is available to APT
   if [ ! -e /usr/lib/apt/methods/https ]; then  apt-get update  apt-get install -y apt-transport-https

   fi
   ```

2. Then it will add the Docker repository to our local key chain:

   ```
   $ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys 36A1D7869245C8950F966E92D8576A8BA88D21E9
   ```

You may receive a warning that the package isn't trusted. Answer `yes` to continue the installation.

3. Finally, it adds the Docker repository to the APT sources list, and updates and installs the `lxc-docker` package:

```
$ sudo sh -c "echo deb https://get.docker.io/ubuntu docker
main\
> /etc/apt/sources.list.d/docker.list"
$ sudo apt-get update
$ sudo apt-get install lxc-docker
```

Docker versions before 0.9 had a hard dependency on LXC (Linux Containers) and hence couldn't be installed on VMs hosted on OpenVZ. But since 0.9, the execution driver has been decoupled from the Docker core, which allows us to use one of numerous isolation tools such as LXC, OpenVZ, systemd-nspawn, libvirt-lxc, libvirt-sandbox, qemu/kvm, BSD Jails, Solaris Zones, and even chroot! However, it comes by default with an execution driver for Docker's own containerization engine, called l**ibcontainer**, which is a pure Go library that can access the kernel's container APIs directly, without any other dependencies.

To use any other containerization engine, say LXC, you can use the-e flag, like so: `$ docker -d -e lxc`.

Now that we have Docker installed, we can get going at full steam! There is one problem though: software repositories like **APT** are usually behind times and often have older versions. Docker is a fast-moving project and a lot has changed in the last few versions. So it is always recommended to have the latest version installed.

# Upgrading Docker

You can upgrade Docker as and when it is updated in the APT repositories. An alternative (and better) method is to build from source. The tutorial for this method is in the section titled *Inception: Docker in Docker*. It is recommended to upgrade to the newest stable version as the newer versions might contain critical security updates and bug fixes. Also, the examples in this book assume a Docker version greater than 1.0, whereas Ubuntu's standard repositories package a much older version.

# Mac OSX and Windows

Docker depends on the Linux kernel, so we need to run Linux in a VM and install and use Docker through it. Boot2Docker is a helper application built by Docker Inc. that installs a VM containing a lightweight Linux distribution made specifically to run Docker containers. It also comes with a client that provides the same **Application Program Interface** (**API**) as that of Docker, but interfaces with the `docker` daemon running in the VM, allowing us to run commands from within the OSX/Windows terminal. To install Boot2Docker, carry out the following steps:

1. Download the latest release of Boot2Docker for your operating system from `http://boot2docker.io/`.
2. The installation image is shown as follows:


3. Run the installer, which will install VirtualBox and the Boot2Docker management tool.

Run Boot2docker. The first run will ask you for a **Secure Shell** (**SSH**) key passphrase. Subsequent runs of the script will connect you to a shell session in the virtual machine. If needed, the subsequent runs will initialize a new VM and start it.

Alternately, to run Boot2Docker, you can also use the terminal command `boot2docker`:

```
$ boot2docker init # First run
```

```
$ boot2docker start
$ export DOCKER_HOST=tcp://$(boot2docker ip 2>/dev/null):2375
```

You will have to run `boot2docker init` only once. It will ask you for an SSH key passphrase. This passphrase is subsequently used by `boot2docker ssh` to authenticate SSH access.

Once you have initialized Boot2Docker, you can subsequently use it with the `boot2docker start` and `boot2docker stop` commands.

`DOCKER_HOST` is an environment variable that, when set, indicates to the Docker client the location of the `docker` daemon. A port forwarding rule is set to the boot2Docker VM's port 2375 (where the `docker` daemon runs). You will have to set this variable in every terminal shell you want to use Docker in.

Bash allows you to insert commands by enclosing subcommands within `` ` ` `` or `$()`. These will be evaluated first and the result will be substituted in the outer commands.

If you are the kind that loves to poke around, the Boot2Docker default user is `docker` and the password is `tcuser`.

The boot2Docker management tool provides several commands:

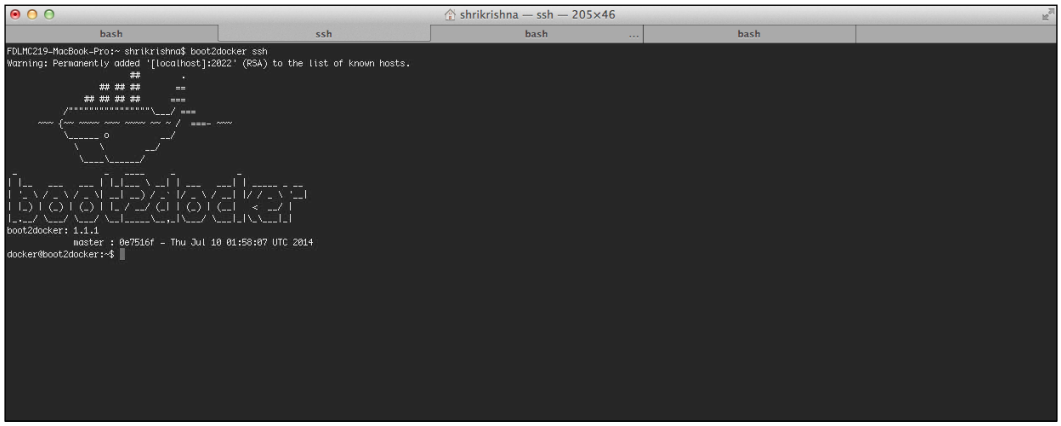
```
$ boot2docker
Usage: boot2docker [<options>] {help|init|up|ssh|save|down|poweroff|reset
|restart|config|status|info
|ip|delete|download|version} [<args>]
```

When using boot2Docker, the `DOCKER_HOST` environment variable has to be available in the terminal session for Docker commands to work. So, if you are getting the `Post http:///var/run/docker.sock/v1.12/containers/create: dial unix /var/run/docker.sock: no such file or directory` error, it means that the environment variable is not assigned. It is easy to forget to set this environment variable when you open a new terminal. For OSX users, to make things easy, add the following line to your `.bashrc` or `.bash_profile` shells:

```
alias setdockerhost='export DOCKER_HOST=tcp://$(boot2docker ip
2>/dev/null):2375'
```

Now, whenever you open a new terminal or get the above error, just run the following command:

```
$ setdockerhost
```




This image shows how the terminal screen will look like when you have logged into the Boot2Docker VM.

## Upgrading Boot2Docker

1. Download the latest release of the Boot2Docker Installer for OSX from `http://boot2docker.io/`.
2. Run the installer, which will update VirtualBox and the Boot2Docker management tool.

To upgrade your existing virtual machine, open a terminal and run the following commands:

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

# OpenStack

**OpenStack** is a piece of free and open source software that allows you to set up a cloud. It is primarily used to deploy public and private **Infrastructure as a Service** (**IaaS**) solutions. It consists of a pool of interrelated projects for the different components of a cloud setup such as compute schedulers, keychain managers, network managers, storage managers, dashboards, and so on.

Docker can act as a hypervisor driver for OpenStack Nova Compute. Docker support for OpenStack was introduced with the **Havana** release.

But... how?

Nova's Docker driver embeds a tiny HTTP server that talks to the Docker Engine's internal **Representational State Transfer** (**REST**) API (you will learn more on this later) through a **UNIX TCP** socket.

Docker has its own image repository system called Docker-Registry, which can be embedded into Glance (OpenStack's image repository) to push and pull Docker images. Docker-Registry can be run either as a `docker` container or in a standalone mode.

# Installation with DevStack

If you are just setting up OpenStack and taking up the DevStack route, configuring the setup to use Docker is pretty easy.

Before running the DevStack route's `stack.sh` script, configure the **virtual driver** option in the `localrc` file to use Docker:

```
VIRT_DRIVER=docker
```

Then run the Docker installation script from the `devstack` directory. The `socat` utility is needed for this script (usually installed by the `stack.sh` script). If you don't have the `socat` utility installed, run the following:

```
$ apt-get install socat
$ ./tools/docker/install_docker.sh
```

Finally, run the `stack.sh` script from the `devstack` directory:

```
$ ./stack.sh
```

# Installing Docker for OpenStack manually

Docker can also be installed manually if you already have OpenStack set up or in case the DevStack method doesn't work out:

1. Firstly, install Docker according to one of the Docker installation procedures.

   If you are co-locating the `docker` registry alongside the Glance service, run the following command:

   ```
   $ sudo yum -y install docker-registry
   ```

In the `/etc/sysconfig/docker-registry` folder, set the `REGISTRY_PORT` and `SETTINGS_FLAVOR` registries as follows:

```
$ export SETTINGS_FLAVOR=openstack
$ export REGISTRY_PORT=5042
```

In the `docker` registry file, you will also need to specify the OpenStack authentication variables. The following commands accomplish this:

```
$ source /root/keystonerc_admin
$ export OS_GLANCE_URL=http://localhost:9292
```

By default, `/etc/docker-registry.yml` sets the local or alternate `storage_path` path for the openstack configuration under `/tmp`. You may want to alter the path to a more permanent location:

```
openstack:
   storage: glance
   storage_alternate: local
   storage_path: /var/lib/docker-registry
```

2. In order for **Nova** to communicate with Docker over its local socket, add `nova` to the `docker` group and restart the `compute` service to pick up the change:

```
$ usermod -G docker nova
$ service openstack-nova-compute restart
```

3. Start Redis (used by the Docker Registry), if it wasn't started already:

```
$ sudo service redis start
$ sudo chkconfig redis on
```

4. Finally, start the registry:

```
$ sudo service docker-registry start
$ sudo chkconfig docker-registry on
```

# Nova configuration

Nova needs to be configured to use the `virt` Docker driver.

Edit the `/etc/nova/nova.conf` configuration file according to the following options:

```
[DEFAULT]
compute_driver = docker.DockerDriver
```

Alternatively, if you want to use your own Docker-Registry, which listens on a port different than 5042, you can override the following option:

```
docker_registry_default_port = 5042
```

# Glance configuration

Glance needs to be configured to support the Docker container format. Just add Docker to the list of container formats in the Glance configuration file:

```
[DEFAULT]
container_formats = ami,ari,aki,bare,ovf,docker
```

Leave the default formats in order to not break an existing glance installation.

# Docker-OpenStack flow

Once you configured Nova to use the `docker` driver, the flow is the same as that in any other driver:

```
$ docker search hipache
Found 3 results matching your query ("hipache")
NAME                            DESCRIPTION
samalba/hipache                 https://github.com/dotcloud/hipache
```

Then tag the image with the Docker-Registry location and push it:

```
$ docker pull samalba/hipache
$ docker tag samalba/hipache localhost:5042/hipache
$ docker push localhost:5042/hipache
```

The push refers to a repository:

```
[localhost:5042/hipache] (len: 1)
Sending image list
Pushing repository localhost:5042/hipache (1 tags)
Push 100% complete
```

In this case, the Docker-Registry (running in a docker container with a port mapped on 5042) will push the images to Glance. From there, Nova can reach them and you can verify the images with the Glance **Command-Line Interface** (**CLI**):

```
$ glance image-list
```

Only images with a docker container format will be bootable. The image basically contains a tarball of the container filesystem.

You can boot instances with the `nova boot` command:

```
$ nova boot --image "docker-busybox:latest" --flavor m1.tiny test
```

The command used will be the one configured in the image. Each container image can have a command configured for the run. The driver does not override this command.

Once the instance is booted, it will be listed in `nova list`:

```
$ nova list
```

You can also see the corresponding container in Docker:

```
$ docker ps
```

# Inception: Build Docker in Docker

Though installing from standard repositories is easier, they usually contain older versions, which means that you might miss critical updates or features. The best way to remain updated is to regularly get the latest version from the public `GitHub` repository. Traditionally, building software from a source has been painful and done only by people who actually work on the project. This is not so with Docker. From Docker 0.6, it has been possible to build Docker in Docker. This means that upgrading Docker is as simple as building a new version in Docker itself and replacing the binary. Let's see how this is done.

# Dependencies

You need to have the following tools installed in a 64-bit Linux machine (VM or bare-metal) to build Docker:

- **Git**
- **Make**

**Git** is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency. It is used here to clone the Docker public source code repository. Check out `git-scm.org` for more details.

The `make` utility is a software engineering tool used to manage and maintain computer programs. **Make** provides most help when the program consists of many component files. A `Makefile` file is used here to kick off the Docker containers in a repeatable and consistent way.

# Building Docker from source

To build Docker in Docker, we will first fetch the source code and then run a few `make` commands that will, in the end, create a `docker` binary, which will replace the current binary in the Docker installation path.

Run the following command in your terminal:

```
$ git clone https://git@github.com/dotcloud/docker
```

This command clones the official Docker source code repository from the `Github` repository into a directory named `docker`:

```
$ cd docker
$ sudo make build
```

This will prepare the development environment and install all the dependencies required to create the binary. This might take some time on the first run, so you can go and have a cup of coffee.

If you encounter any errors that you find difficult to debug, you can always go to `#docker` on freenode IRC. The developers and the Docker community are very helpful.

Now we are ready to compile that binary:

```
$ sudo make binary
```

This will compile a binary and place it in the `./bundles/<version>-dev/binary/` directory. And voila! You have a fresh version of Docker ready.

Before replacing your existing binary though, run the tests:

```
$ sudo make test
```

If the tests pass, then it is safe to replace your current binary with the one you've just compiled. Stop the `docker` service, create a backup of the existing binary, and then copy the freshly baked binary in its place:

```
$ sudo service docker stop
$ alias wd='which docker'
$ sudo cp $(wd) $(wd)_
$ sudo cp $(pwd)/bundles/<version>-dev/binary/docker-<version>-dev $(wd)
$ sudo service docker start
```

Congratulations! You now have the up-to-date version of Docker running.

OSX and Windows users can follow the same procedures as SSH in the boot2Docker VM.

# Verifying Installation

To verify that your installation is successful, run the following command in your terminal console:

```
$ docker run -i -t ubuntu echo Hello World!
```

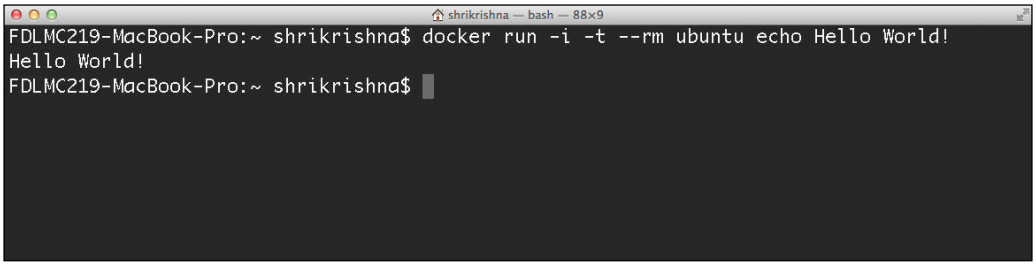
The `docker run` command starts a container with the `ubuntu` base image. Since this is the first time you are starting an `ubuntu` container, the output of the container will be something like this:

```
Unable to find image 'ubuntu' locally
Pulling repository ubuntu
e54ca5efa2e9: Download complete
511136ea3c5a: Download complete
d7ac5e4f1812: Download complete
2f4b4d6a4a06: Download complete
83ff768040a0: Download complete
6c37f792ddac: Download complete

Hello World!
```

When you issue the `docker run ubuntu` command, Docker looks for the `ubuntu` image locally, and it's not found, it will download the `ubuntu` image from the public `docker` registry. You will also see it say **Pulling dependent layers**.

This means that it is downloading filesystem layers. By default, Docker uses AUFS, a layered copy-on-write filesystem, which means that the container image's filesystem is a culmination of multiple read-only filesystem layers. And these layers are shared between running containers. If you initiate an action that will write to this filesystem, it will create a new layer that will be the difference of the underlying layers and the new data. Sharing of common layers means that only the first container will take up a considerable amount of memory and subsequent containers will take up an insignificant amount of memory as they will be sharing the read-only layers. This means that you can run hundreds of containers even on a relatively low-powered laptop.




Once the image has been completely downloaded, it will start the container and echo `Hello World!` in your console. This is another salient feature of the Docker containers. Every container is associated with a command and it should run that command. Remember that the Docker containers are unlike VMs in that they do not virtualize the entire operating system. Each `docker` container accepts only a single command and runs it in a sandboxed process that lives in an isolated environment.

# Useful tips

The following are two useful tips that might save you a lot of trouble later on. The first shows how to give the docker client non-root access, and the second shows how to configure the Ubuntu firewall rules to enable forwarding network traffic.


You do not need to follow these if you are using Boot2Docker.

# Giving non-root access

Create a group called `docker` and add your user to that group to avoid having to add the `sudo` prefix to every `docker` command. The reason you need to run a `docker` command with the `sudo` prefix by default is that the `docker` daemon needs to run with `root` privileges, but the docker client (the commands you run) doesn't. So, by creating a `docker` group, you can run all the client commands without using the `sudo` prefix, whereas the daemon runs with the `root` privileges:

```
$ sudo groupadd docker # Adds the docker group
$ sudo gpasswd -a $(whoami) docker # Adds the current user to the
group
$ sudo service docker restart
```

You might need to log out and log in again for the changes to take effect.

# UFW settings

Docker uses a bridge to manage network in the container. **Uncomplicated Firewall** (**UFW**) is the default firewall tool in Ubuntu. It drops all forwarding traffic. You will need to enable forwarding like this:

```
$ sudo vim /etc/default/ufw
# Change:
# DEFAULT_FORWARD_POLICY="DROP"
# to
DEFAULT_FORWARD_POLICY="ACCEPT"
```

Reload the firewall by running the following command:

```
$ sudo ufw reload
```

Alternatively, if you want to be able to reach your containers from other hosts, then you should enable incoming connections on the docker port (`default 2375`):

```
$ sudo ufw allow 2375/tcp
```

**Downloading the example code**

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

# Summary

I hope this introductory chapter got you hooked to Docker. The upcoming chapters will take you into the Docker world and try to dazzle you with its awesomeness.

In this chapter, you learned some history and some basics on Docker and how it works. We saw how it is different from and advantageous over VM.

Then we proceeded to install Docker on our development setup, be it Ubuntu, Mac, or Windows. Then we saw how to replace OpenStack's hypervisor with Docker. Later, we built Docker from source, within Docker! Talk about eating your own dog food!

Finally, we downloaded our first image and ran our first container. Now you can pat your self on the back and proceed to the next chapter, where we will cover the primary Docker commands in depth and see how we can create our own images.

# 2
# Docker CLI and Dockerfile

In the last chapter, we set up Docker in our development setup and ran our first container. In this chapter, we will explore the Docker command-line interface. Later in the chapter, we will see how to create our own Docker images using Dockerfiles and how to automate this process.

In this chapter, we will cover the following topics:

- Docker terminologies
- Docker commands
- Dockerfiles
- Docker workflow—pull-use-modify-commit-push workflow
- Automated builds

## Docker terminologies

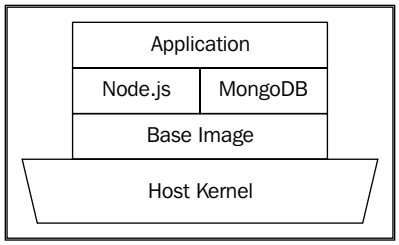
Before we begin our exciting journey into the Docker sphere, let's understand the Docker terminologies that will be used in this book a little better. Very similar in concept to VM images, a Docker image is a snapshot of a system. The difference between a VM image and a Docker image is that a VM image can have running services, whereas a Docker image is just a filesystem snapshot, which means that while you can configure the image to have your favorite packages, you can run only one command in the container. Don't fret though, since the limitation is one command, not one process, so there are ways to get a Docker container to do almost anything a VM instance can.

Docker has also implemented a Git-like distributed version management system for Docker images. Images can be stored in repositories (called a registry), both locally and remotely. The functionalities and terminologies borrow heavily from Git—snapshots are called commits, you pull an image repository, you push your local image to a repository, and so on.

# Docker container

A Docker container can be correlated to an instance of a VM. It runs sandboxed processes that share the same kernel as the host. The term **container** comes from the concept of shipping containers. The idea is that you can ship containers from your development environment to the deployment environment and the applications running in the containers will behave the same way no matter where you run them.

The following image shows the layers of AUFS:




This is similar in context to a shipping container, which stays sealed until delivery but can be loaded, unloaded, stacked, and transported in between.

The visible filesystem of the processes in the container is based on AUFS (although you can configure the container to run with a different filesystem too). AUFS is a layered filesystem. These layers are all read-only and the merger of these layers is what is visible to the processes. However, if a process makes a change in the filesystem, a new layer is created, which represents the difference between the original state and the new state. When you create an image out of this container, the layers are preserved. Thus, it is possible to build new images out of existing images, creating a very convenient hierarchical model of images.

# The docker daemon

The `docker` daemon is the process that manages containers. It is easy to get this confused with the Docker client because the same binary is used to run both the processes. The `docker` daemon, though, needs the `root` privileges, whereas the client doesn't.

Unfortunately, since the `docker` daemon runs with root privileges, it also introduces an attack vector. Read `https://docs.Docker.com/articles/security/` for more details.

# Docker client

The Docker client is what interacts with the `docker` daemon to start or manage containers. Docker uses a RESTful API to communicate between the client and the daemon.

REST is an architectural style consisting of a coordinated set of architectural constraints applied to components, connectors, and data elements within a distributed hypermedia system. In plain words, a RESTful service works over standard HTTP methods such as the `GET`, `POST`, `PUT`, and `DELETE` methods.

# Dockerfile

A Dockerfile is a file written in a **Domain Specific Language** (**DSL**) that contains instructions on setting up a Docker image. Think of it as a Makefile equivalent of Docker.

# Docker registry

This is the public repository of all Docker images published by the Docker community. You can pull images from this registry freely, but to push images, you will have to register at `http://hub.docker.com`. Docker registry and Docker hub are services operated and maintained by Docker Inc., and they provide unlimited free repositories. You can also buy private repositories for a fee.

# Docker commands

Now let's get our hands dirty on the Docker CLI. We will look at the most common commands and their use cases. The Docker commands are modeled after Linux and Git, so if you have used either of these, you will find yourself at home with Docker.

Only the most commonly used options are mentioned here. For the complete reference, you can check out the official documentation at `https://docs.docker.com/reference/commandline/cli/`.

# The daemon command

If you have installed the `docker` daemon through standard repositories, the command to start the `docker` daemon would have been added to the `init` script to automatically start as a service on startup. Otherwise, you will have to first run the `docker` daemon yourself for the client commands to work.

Now, while starting the daemon, you can run it with arguments that control the **Domain Name System** (**DNS**) configurations, storage drivers, and execution drivers for the containers:

```
$ export DOCKER_HOST="tcp://0.0.0.0:2375"
$ Docker -d -D -e lxc -s btrfs --dns 8.8.8.8 --dns-search example.com
```

You'll need these only if you want to start the daemon yourself. Otherwise, you can start the `docker` daemon with `$ sudo service Docker start`. For OSX and Windows, you need to run the commands mentioned in *Chapter 1*, *Installing Docker*.

The following table describes the various flags:

| Flag | Explanation |
|---|---|
| `-d` | This runs Docker as a daemon. |
| `-D` | This runs Docker in debug mode. |
| `-e [option]` | This is the execution driver to be used. The default execution driver is native, which uses `libcontainer`. |
| `-s [option]` | This forces Docker to use a different storage driver. The default value is "", for which Docker uses AUFS. |
| `--dns [option(s)]` | This sets the DNS server (or servers) for all Docker containers. |
| `--dns-search [option(s)]` | This sets the DNS search domain (or domains) for all Docker containers. |
| `-H [option(s)]` | This is the socket (or sockets) to bind to. It can be one or more of `tcp://host:port, unix:///path/to/socket, fd://* or fd://socketfd`. |

If multiple `docker` daemons are being simultaneously run, the client honors the value set by the `DOCKER_HOST` parameter. You can also make it connect to a specific daemon with the `-H` flag.

Consider this command:

```
$ docker -H tcp://0.0.0.0:2375 run -it ubuntu /bin/bash
```

The preceding command is the same as the following command:

```
$ DOCKER_HOST="tcp://0.0.0.0:2375" docker run -it ubuntu /bin/bash
```

# The version command

The `version` command prints out the version information:

```
$ docker -v
Docker version 1.1.1, build bd609d2
```

# The info command

The `info` command prints the details of the `docker` daemon configuration such as the execution driver, the storage driver being used, and so on:

```
$ docker info # The author is running it in boot2docker on OSX
Containers: 0
Images: 0
Storage Driver: aufs
 Root Dir: /mnt/sda1/var/lib/docker/aufs
 Dirs: 0
Execution Driver: native-0.2
Kernel Version: 3.15.3-tinycore64
Debug mode (server): true
Debug mode (client): false
Fds: 10
Goroutines: 10
EventsListeners: 0
Init Path: /usr/local/bin/docker
Sockets: [unix:///var/run/docker.sock tcp://0.0.0.0:2375]
```

# The run command

The run command is the command that we will be using most frequently. It is used to run Docker containers:

```
$ docker run [options] IMAGE [command] [args]
```

| Flags | Explanation |
|---|---|
| `-a, --attach=[]` | Attach to the `stdin`, `stdout`, or `stderr` files (standard input, output, and error files.). |
| `-d, --detach` | This runs the container in the background. |
| `-i, --interactive` | This runs the container in interactive mode (keeps the `stdin` file open). |
| `-t, --tty` | This allocates a pseudo `tty` flag (which is required if you want to attach to the container's terminal). |
| `-p, --publish=[]` | This publishes a container's port to the host (`ip:hostport:containerport`). |
| `--rm` | This automatically removes the container when exited (it cannot be used with the `-d` flag). |
| `--privileged` | This gives additional privileges to this container. |
| `-v, --volume=[]` | This bind mounts a volume (from host => `/host:/container`; from docker => `/container`). |
| `--volumes-from=[]` | This mounts volumes from specified containers. |
| `-w, --workdir=""` | This is the working directory inside the container. |
| `--name=""` | This assigns a name to the container. |
| `-h, --hostname=""` | This assigns a hostname to the container. |
| `-u, --user=""` | This is the username or UID the container should run on. |
| `-e, --env=[]` | This sets the environment variables. |
| `--env-file=[]` | This reads environment variables from a new line-delimited file. |
| `--dns=[]` | This sets custom DNS servers. |
| `--dns-search=[]` | This sets custom DNS search domains. |
| `--link=[]` | This adds link to another container (`name:alias`). |
| `-c, --cpu-shares=0` | This is the relative CPU share for this container. |
| `--cpuset=""` | These are the CPUs in which to allow execution; starts with 0. (For example, 0 to 3). |
| `-m, --memory=""` | This is the memory limit for this container (`<number><b\|k\|m\|g>`). |
| `--restart=""` | (v1.2+) This specifies a restart policy in case the container crashes. |

| Flags | Explanation |
|---|---|
| `--cap-add=""` | (v1.2+) This grants a capability to a container (refer to *Chapter 4, Security Best Practices*). |
| `--cap-drop=""` | (v1.2+) This blacklists a capability to a container (refer to *Chapter 4, Security Best Practices*). |
| `--device=""` | (v1.2+) This mounts a device on a container. |

While running a container, it is important to keep in mind that the container's lifetime is associated with the lifetime of the command you run when you start the container. Now try to run this:

```
$ docker run -dt ubuntu ps
b1d037dfcff6b076bde360070d3af0d019269e44929df61c93dfcdfaf29492c9
$ docker attach b1d037
2014/07/16 16:01:29 You cannot attach to a stopped container, start
it first
```

What happened here? When we ran the simple command, `ps`, the container ran the command and exited. Therefore, we got an error.

The `attach` command attaches the standard input and output to a running container.

Another important piece of information here is that you don't need to use the whole 64-character ID for all the commands that require the container ID. The first couple of characters are sufficient. With the same example as shown in the following code:

```
$ docker attach b1d03
2014/07/16 16:09:39 You cannot attach to a stopped container, start
it first
$ docker attach b1d0
2014/07/16 16:09:40 You cannot attach to a stopped container, start
it first
$ docker attach b1d
2014/07/16 16:09:42 You cannot attach to a stopped container, start
it first
$ docker attach b1
2014/07/16 16:09:44 You cannot attach to a stopped container, start
it first
$ docker attach b
2014/07/16 16:09:45 Error: No such container: b
```

A more convenient method though would be to name your containers yourself:

```
$ docker run -dit --name OD-name-example ubuntu /bin/bash
1b21af96c38836df8a809049fb3a040db571cc0cef000a54ebce978c1b5567ea
$ docker attach OD-name-example
root@1b21af96c388:/#
```

The `-i` flag is necessary to have any kind of interaction in the container, and the `-t` flag is necessary to create a pseudo-terminal.

The previous example also made us aware of the fact that even after we exit a container, it is still in a `stopped` state. That is, we will be able to start the container again, with its filesystem layer preserved. You can see this by running the following command:

```
$ docker ps -a
CONTAINER ID IMAGE          COMMAND CREATED    STATUS    NAMES
eb424f5a9d3f ubuntu:latest ps      1 hour ago Exited OD-name-example
```

While this can be convenient, you may pretty soon have your host's disk space drying up as more and more containers are saved. So, if you are going to run a disposable container, you can run it with the `--rm` flag, which will remove the container when the process exits:

```
$ docker run --rm -it --name OD-rm-example ubuntu /bin/bash
root@0fc99b2e35fb:/# exit
exit
$ docker ps -a
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS    PORTS    NAMES
```

## Running a server

Now, for our next example, we'll try running a web server. This example is chosen because the most common practical use case of Docker containers is the shipping of web applications:

```
$ docker run -it --name OD-pythonserver-1 --rm python:2.7 \
python -m SimpleHTTPServer 8000;
Serving HTTP on 0.0.0.0 port 8000
```

Now we know the problem; we have a server running in a container, but since the container's IP is assigned by Docker dynamically, it makes things difficult. However, we can bind the container's ports to the host's ports and Docker will take care of forwarding the networking traffic. Now let's try this command again with the `-p` flag:

```
$ docker run -p 0.0.0.0:8000:8000 -it --rm --name OD-pythonserver-2 \
python:2.7 python -m SimpleHTTPServer 8000;
Serving HTTP on 0.0.0.0 port 8000 ...
172.17.42.1 - - [18/Jul/2014 14:25:46] "GET / HTTP/1.1" 200 -
```

Now open your browser and go to `http://localhost:8000`. Voilà!

If you are an OS X user and you realize that you are not able to access `http://localhost:8000`, it is because VirtualBox hasn't been configured to respond to **Network Address Translation** (**NAT**) requests to the boot2Docker VM. Adding the following function to your aliases file (`bash_profile` or `.bashrc`) will save a lot of trouble:

```
natboot2docker () {
   VBoxManage controlvm boot2docker-vm natpf1 \
   "$1,tcp,127.0.0.1,$2,,$3";
}


removeDockerNat() {
    VBoxManage modifyvm boot2docker-vm \
    --natpf1 delete $1;
}
```

After this, you should be able to use the `$ natboot2docker mypythonserver 8000 8000` command to be able to access the Python server. But remember to run the `$ removeDockerDockerNat mypythonserver` command when you are done. Otherwise, when you run the boot2Docker VM next time, you will be faced with a bug that won't allow you to get the IP address or the `ssh` script into it:

```
$ boot2docker ssh
ssh_exchange_identification: Connection closed by remote host
2014/07/19 11:55:09 exit status 255
```

Your browser now shows the `/root` path of the container. What if you wanted to serve your host's directories? Let's try mounting a device:

```
root@eb53f7ec79fd:/# mount -t tmpfs /dev/random /mnt
mount: permission denied
```

As you can see, the `mount` command doesn't work. In fact, most kernel capabilities that are potentially dangerous are dropped, unless you include the `--privileged` flag.

However, you should never use this flag unless you know what you are doing. Docker provides a much easier way to bind mount host volumes and bind mount host volumes with the `-v` and `–volumes` options. Let's try this example again in the directory we are currently in:

```
$ docker run -v $(pwd):$(pwd) -p 0.0.0.0:8000:8000 -it –rm \
--name OD-pythonserver-3 python:2.7 python -m SimpleHTTPServer 8000;
Serving HTTP on 0.0.0.0 port 8000 ...
10.0.2.2 - - [18/Jul/2014 14:40:35] "GET / HTTP/1.1" 200 -
```

You have now bound the directory you are running the commands from to the container. However, when you access the container, you still get the directory listing of the root of the container. To serve the directory that has been bound to the container, let's set it as the working directory of the container (the directory the containerized process runs in) using the `-w` flag:

```
$ docker run -v $(pwd):$(pwd) -w $(pwd) -p 0.0.0.0:8000:8000 -it \ --name
OD-pythonserver-4 python:2.7 python -m SimpleHTTPServer 8000;
Serving HTTP on 0.0.0.0 port 8000 ...
10.0.2.2 - - [18/Jul/2014 14:51:35] "GET / HTTP/1.1" 200 -
```

Boot2Docker users will not be able to utilize this yet, unless you use guest additions and set up shared folders, the guide to which can be found at `https://medium.com/boot2docker-lightweight-linux-for-docker/boot2docker-together-with-virtualbox-guest-additions-da1e3ab2465c`. Though this solution works, it is a hack and is not recommended. Meanwhile, the Docker community is actively trying to find a solution (check out issue `#64` in the boot2Docker GitHub repository and `#4023` in the Docker repository).

Now `http://localhost:8000` will serve the directory you are currently running in, but from a Docker container. Take care though, because any changes you make are written into the host's filesystem as well.

Since v1.1.1, you can bind mount the root of the host to a container using `$ docker run -v /:/my_host:ro ubuntu ls /my_host`, but mounting on the / path of the container is forbidden.

The volume can be optionally suffixed with the `:ro` or `:rw` commands to mount the volumes in read-only or read-write mode, respectively. By default, the volumes are mounted in the same mode (read-write or read-only) as they are in the host.

This option is mostly used to mount static assets and to write logs.

But what if I want to mount an external device?

Before v1.2, you had to mount the device in the host and bind mount using the `-v` flag in a privileged container, but v1.2 has added a `--device` flag that you can use to mount a device without needing to use the `--privileged` flag.

For example, to use the webcam in your container, run this command:

```
$ docker run --device=/dev/video0:/dev/video0
```

Docker v1.2 also added a `--restart` flag to specify a restart policy for containers. Currently, there are three restart policies:

- `no`: Do not restart the container if it dies (default).
- `on-failure`: Restart the container if it exits with a non-zero exit code. It can also accept an optional maximum restart count (for example, `on-failure:5`).
- `always`: Always restart the container no matter what exit code is returned.

The following is an example to restart endlessly:

```
$ docker run --restart=always code.it
```

The next line is used to try five times before giving up:

```
$ docker run --restart=on-failure:5 code.it
```

# The search command

The `search` command allows us to search for Docker images in the public registry. Let's search for all images related to Python:

```
$ docker search python | less
```

# The pull command

The `pull` command is used to pull images or repositories from a registry. By default, it pulls them from the public Docker registry, but if you are running your own registry, you can pull them from it too:

```
$ docker pull python # pulls repository from Docker Hub
$ docker pull python:2.7 # pulls the image tagged 2.7
$ docker pull <path_to_registry>/<image_or_repository>
```

# The start command

We saw when we discussed `docker run` that the container state is preserved on exit unless it is explicitly removed. The `docker start` command starts a stopped container:

```
$ docker start [-i] [-a] <container(s)>
```

Consider the following example of the `start` command:

```
$ docker ps -a
CONTAINER ID IMAGE          COMMAND   CREATED STATUS    NAMES
e3c4b6b39cff ubuntu:latest python -m 1h ago  Exited OD-pythonserver-4
81bb2a92ab0c ubuntu:latest /bin/bash 1h ago  Exited evil_rosalind
d52fef570d6e ubuntu:latest /bin/bash 1h ago  Exited prickly_morse
eb424f5a9d3f ubuntu:latest /bin/bash 20h ago Exited OD-name-example
$ docker start -ai OD-pythonserver-4
Serving HTTP on 0.0.0.0 port 8000
```

The options have the same meaning as with the `docker run` command.

# The stop command

The `stop` command stops a running container by sending the `SIGTERM` signal and then the `SIGKILL` signal after a grace period:

`SIGTERM` and `SIGKILL` are Unix signals. A signal is a form of interprocess communication used in Unix, Unix-like, and other POSIX-compliant operating systems. `SIGTERM` signals the process to terminate. The `SIGKILL` signal is used to forcibly kill a process.

```
docker run -dit --name OD-stop-example ubuntu /bin/bash
$ docker ps
CONTAINER ID IMAGE         COMMAND   CREATED  STATUS    NAMES
679ece6f2a11 ubuntu:latest /bin/bash 5h ago   Up 3s   OD-stop-example
$ docker stop OD-stop-example
OD-stop-example
$ docker ps
CONTAINER ID IMAGE         COMMAND   CREATED  STATUS    NAMES
```

You can also specify the `-t` flag or `--time` flag, which allows you to set the wait time.

# The restart command

The `restart` command restarts a running container:

```
$ docker run -dit --name OD-restart-example ubuntu /bin/bash
$ sleep 15s # Suspends execution for 15 seconds
$ docker ps
CONTAINER ID IMAGE         COMMAND   STATUS    NAMES
cc5d0ae0b599 ubuntu:latest /bin/bash Up 20s    OD-restart-example

$ docker restart OD-restart-example
$ docker ps
CONTAINER ID IMAGE         COMMAND   STATUS    NAMES
cc5d0ae0b599 ubuntu:latest /bin/bash Up 2s    OD-restart-example
```

If you observe the status, you will notice that the container was rebooted.

# The rm command

The `rm` command removes Docker containers:

```
$ Docker ps -a # Lists containers including stopped ones
CONTAINER ID  IMAGE  COMMAND   CREATED  STATUS NAMES
cc5d0ae0b599  ubuntu /bin/bash 6h ago   Exited OD-restart-example
679ece6f2a11  ubuntu /bin/bash 7h ago   Exited OD-stop-example
e3c4b6b39cff  ubuntu /bin/bash 9h ago   Exited OD-name-example
```

We seem to be having a lot of containers left over after our adventures. Let's remove one of them:

```
$ dockerDocker rm OD-restart-example
cc5d0ae0b599
```

We can also combine two Docker commands. Let's combine the `docker ps -a -q` command, which prints the ID parameters of the containers in the `docker ps -a`, and `docker rm` commands, to remove all containers in one go:

```
$ docker rm $(docker ps -a -q)
679ece6f2a11
e3c4b6b39cff
$ docker ps -a
CONTAINER ID    IMAGE    COMMAND     CREATED    STATUS      NAMES
```

This evaluates the `docker ps -a -q` command first, and the output is used by the `docker rm` command.

# The ps command

The `ps` command is used to list containers. It is used in the following way:

```
$ docker ps [option(s)]
```

| Flag | Explanation |
|---|---|
| `-a, --all` | This shows all containers, including stopped ones. |
| `-q, --quiet` | This shows only container ID parameters. |
| `-s, --size` | This prints the sizes of the containers. |
| `-l, --latest` | This shows only the latest container (including stopped containers). |
| `-n=""` | This shows the last *n* containers (including stopped containers). Its default value is -1. |
| `--before=""` | This shows the containers created before the specified ID or name. It includes stopped containers. |
| `--after=""` | This shows the containers created after the specified ID or name. It includes stopped containers. |

The `docker ps` command will show only running containers by default. To see all containers, run the `docker ps -a` command. To see only container ID parameters, run it with the `-q` flag.

# The logs command

The `logs` command shows the logs of the container:

```
Let us look at the logs of the python server we have been running
$ docker logs OD-pythonserver-4
Serving HTTP on 0.0.0.0 port 8000 ...
10.0.2.2 - - [18/Jul/2014 15:06:39] "GET / HTTP/1.1" 200 -
^CTraceback (most recent call last):
  File ...
  ...
KeyboardInterrupt
```

You can also provide a `--tail` argument to follow the output as the container is running.

# The inspect command

The `inspect` command allows you to get the details of a container or an image. It returns those details as a JSON array:

```
$ Docker inspect ubuntu # Running on an image
[{
    "Architecture": "amd64",
    "Author": "",
    "Comment": "",
     .......
     .......
     .......
    "DockerVersion": "0.10.0",
    "Id":
"e54ca5efa2e962582a223ca9810f7f1b62ea9b5c3975d14a5da79d3bf6020f37",
    "Os": "linux",
    "Parent":
"6c37f792ddacad573016e6aea7fc9fb377127b4767ce6104c9f869314a12041e",
    "Size": 178365
}]
```

Similarly, for a container we run the following command:

```
$ Docker inspect OD-pythonserver-4 # Running on a container
[{
    "Args": [
        "-m",
        "SimpleHTTPServer",
        "8000"
    ],
    ......
    ......
    "Name": "/OD-pythonserver-4",
    "NetworkSettings": {
        "Bridge": "Docker0",
        "Gateway": "172.17.42.1",
        "IPAddress": "172.17.0.11",
        "IPPrefixLen": 16,
        "PortMapping": null,
        "Ports": {
            "8000/tcp": [
                {
                    "HostIp": "0.0.0.0",
                    "HostPort": "8000"
                }
            ]
        }
    },
    ......
    ......
    "Volumes": {
        "/home/Docker": "/home/Docker"
    },
    "VolumesRW": {
        "/home/Docker": true
    }
}]
```

Docker inspect provides all of the low-level information about a container or image. In the preceding example, find out the IP address of the container and the exposed port and make a request to the `IP:port`. You will see that you are directly accessing the server running in the container.

However, manually looking through the entire JSON array is not optimal. So the `inspect` command provides a flag, `-f` (or the `--follow` flag), which allows you to specify exactly what you want using `Go` templates. For example, if you just want to get the container's IP address, run the following command:

```
$ docker inspect -f  '{{.NetworkSettings.IPAddress}}' \
OD-pythonserver-4;
172.17.0.11
```

The `{{.NetworkSettings.IPAddress}}` is a `Go` template that was executed over the JSON result. `Go` templates are very powerful, and some of the things that you can do with them have been listed at `http://golang.org/pkg/text/template/`.

# The top command

The `top` command shows the running processes in a container and their statistics, mimicking the Unix `top` command.

Let's download and run the `ghost` blogging platform and check out what processes are running in it:

```
$ docker run -d -p 4000:2368 --name OD-ghost dockerfile/ghost
ece88c79b0793b0a49e3d23e2b0b8e75d89c519e5987172951ea8d30d96a2936

$ docker top OD-ghost-1
PID                 USER                COMMAND
1162                root                bash /ghost-start
1180                root                npm
1186                root                sh -c node index
1187                root                node index
```

Yes! We just set up our very own `ghost` blog, with just one command. This brings forth another subtle advantage and shows something that could be a future trend. Every tool that exposes its services through a TCP port can now be containerized and run in its own sandboxed world. All you need to do is expose its port and bind it to your host port. You don't need to worry about installations, dependencies, incompatibilities, and so on, and the uninstallation will be clean because all you need to do is stop all the containers and remove the image.

Ghost is an open source publishing platform that is beautifully designed, easy to use, and free for everyone. It is coded in Node.js, a server-side JavaScript execution engine.

# The attach command

The `attach` command attaches to a running container.

Let's start a container with Node.js, running the node interactive shell as a daemon, and later attach to it.

Node.js is an event-driven, asynchronous I/O web framework that runs applications written in JavaScript on Google's V8 runtime environment.

The container with Node.js is as follows:

```
$ docker run -dit --name OD-nodejs shykes/nodejs node
8e0da647200efe33a9dd53d45ea38e3af3892b04aa8b7a6e167b3c093e522754

$ docker attach OD-nodejs
console.log('Docker rocks!');Docker rocks!
```

# The kill command

The `kill` command kills a container and sends the `SIGTERM` signal to the process running in the container:

```
Let us kill the container running the ghost blog.
$ docker kill OD-ghost-1
OD-ghost-1

$ docker attach OD-ghost-1 # Verification
2014/07/19 18:12:51 You cannot attach to a stopped container, start
it first
```

# The cp command

The `cp` command copies a file or folder from a container's filesystem to the host path. Paths are relative to the root of the filesystem.

It's time to have some fun. First, let's run an Ubuntu container with the `/bin/bash` command:

```
$ docker run -it –name OD-cp-bell ubuntu /bin/bash
```

Now, inside the container, let's create a file with a special name:

```
# touch $(echo -e '\007')
```

The `\007` character is an ASCII `BEL` character that rings the system bell when printed on a terminal. You might have already guessed what we're about to do. So let's open a new terminal and execute the following command to copy this newly created file to the host:

```
$ docker cp OD-cp-bell:/$(echo -e '\007') $(pwd)
```

For the `docker cp` command to work, both the container path and the host path must be complete, so do not use shortcuts such as `.`, `,`, `*`, and so on.

So we created an empty file whose filename is the `BEL` character, in a container. Then we copied the file to the current directory in the host container. Just one last step is remaining. In the host tab where you executed the `docker cp` command, run the following command:

```
$ echo *
```

You will hear the system bell ring! We could have copied any file or directory from the container to the host. But it doesn't hurt to have some fun!

If you found this interesting, you might like to read `http://www.dwheeler.com/essays/fixing-unix-linux-filenames.html`. This is a great essay that discusses the edge cases in filenames, which can cause simple to complicated issues in a program.

# The port command

The `port` command looks up the public-facing port that is bound to an exposed port in the container:

```
$ docker port CONTAINER PRIVATE_PORT
$ docker port OD-ghost 2368

4000
```

Ghost runs a server at the `2368` port that allows you to write and publish a blog post. We bound a host port to the `OD-ghost` container's port `2368` in the example for the `top` command.

# Running your own project

By now, we are considerably familiar with the basic Docker commands. Let's up the ante. For the next couple of commands, I am going to use one of my side projects. Feel free to use a project of your own.

Let's start by listing out our requirements to determine the arguments we must pass to the `docker run` command.

Our application is going to run on Node.js, so we will choose the well-maintained `dockerfile/nodejs` image to start our base container:

- We know that our application is going to bind to port `8000`, so we will expose the port to `8000` of the host.
- We need to give a descriptive name to the container so that we can reference it in future commands. In this case, let's choose the name of the application:

```
$ docker run -it --name code.it dockerfile/nodejs /bin/bash
[ root@3b0d5a04cdcd:/data ]$ cd /home
[ root@3b0d5a04cdcd:/home ]$
```

Once you have started your container, you need to check whether the dependencies for your application are already available. In our case, we only need Git (apart from Node.js), which is already installed in the `dockerfile/nodejs` image.

Now that our container is ready to run our application, all that is remaining is for us to fetch the source code and do the necessary setup to run the application:

```
$ git clone https://github.com/shrikrishnaholla/code.it.git
$ cd code.it && git submodule update --init --recursive
```

This downloads the source code for a plugin used in the application.

Then run the following command:

```
$ npm install
```

Now all the node modules required to run the application are installed.

Next, run this command:

```
$ node app.js
```

Now you can go to `localhost:8000` to use the application.

# The diff command

The `diff` command shows the difference between the container and the image it is based on. In this example, we are running a container with `code.it`. In a separate tab, run this command:

```
$ docker diff code.it
C /home
A /home/code.it
...
```

# The commit command

The `commit` command creates a new image with the filesystem of the container. Just as with Git's `commit` command, you can set a commit message that describes the image:

```
$ docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]
```

| Flag | Explanation |
|---|---|
| `-p, --pause` | This pause the container during commit (availabe from v1.1.1+ onwards). |
| `-m, --message=""` | This is a commit message. It can be a description of what the image does. |
| `-a, --author=""` | This displays the author details. |

For example, let's use this command to commit the container we have set up:

```
$ docker commit -m "Code.it – A browser based text editor and
interpreter" -a "Shrikrishna Holla <s**a@gmail.com>" code.it
shrikrishna/code.it:v1
```

Replace the author details and the username portion of the image name in this example if you are copying these examples.

The output will be a lengthy image ID. If you look at the command closely, we have named the image `shrikrishna/code.it:v1`. This is a convention. The first part of an image/repository's name (before the forward slash) is the Docker Hub username of the author. The second part is the intended application or image name. The third part is a tag (usually a version description) separated from the second part by a colon.

`Docker Hub` is a public registry maintained by Docker, Inc. It hosts public Docker images and provides services to help you build and manage your Docker environment. More details about it can be found at `https://hub.docker.com`.

A collection of images tagged with different versions is a repository. The image you create by running the `docker commit` command will be a local one, which means that you will be able to run containers from it but it won't be available publicly. To make it public or to push to your private Docker registry, use the `docker push` command.

# The images command

The `images` command lists all the images in the system:

```
$ docker images [OPTIONS] [NAME]
```

| Flag | Explanation |
|---|---|
| `-a, --all` | This shows all images, including intermediate layers. |
| `-f, --filter=[]` | This provides filter values. |
| `--no-trunc` | This doesn't truncate output (shows complete ID). |
| `-q, --quiet` | This shows only the image IDs. |

Now let's look at a few examples of the usage of the `image` command:

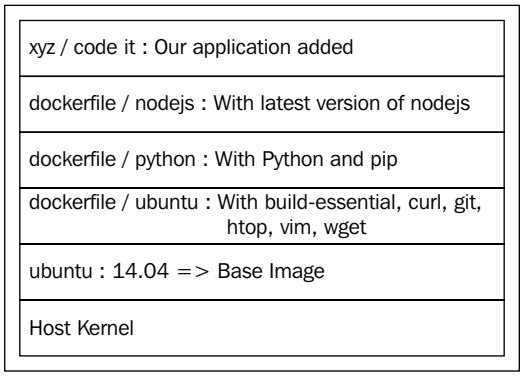
```
$ docker images
REPOSITORY           TAG    IMAGE ID       CREATED     VIRTUAL SIZE
shrikrishna/code.it  v1     a7cb6737a2f6   6m ago      704.4 MB
```

This lists all top-level images, their repository and tags, and their virtual size.

Docker images are nothing but a stack of read-only filesystem layers. A union filesystem, such as AUFS, then merges these layers and they appear to be one filesystem.

In Docker-speak, a read-only layer is an image. It never changes. When running a container, the processes think that the entire filesystem is read-write. But the changes go only at the topmost writeable layer, which is created when a container is started. The read-only layers of the image remain unchanged. When you commit a container, it freezes the top layer (the underlying layers are already frozen) and turns it into an image. Now, when a container is started this image, all the layers of the image (including the previously writeable layer) are read-only. All the changes are now made to a new writeable layer on top of all the underlying layers. However, because of how union filesystems (such as AUFS) work, the processes believe that the filesystem is read-write.

A rough schematic of the layers involved in our `code.it` example is as follows:




At this point, it might be wise to think just how much effort is to be made by the union filesystems to merge all of these layers and provide a consistent performance. After some point, things inevitably break. AUFS, for instance, has a 42-layer limit. When the number of layers goes beyond this, it just doesn't allow the creation of any more layers and the build fails. Read `https://github.com/docker/docker/issues/1171` for more information on this issue.

The following command lists the most recently created images:

```
$ docker images | head
```

The `-f` flag can be given arguments of the `key=value` type. It is frequently used to get the list of dangling images:

```
$ docker images -f "dangling=true"
```

This will display untagged images, that is, images that have been committed or built without a tag.

# The rmi command

The `rmi` command removes images. Removing an image also removes all the underlying images that it depends on and were downloaded when it was pulled:

```
$ docker rmi [OPTION] {IMAGE(s)]
```

| Flag | Explanation |
|---|---|
| `-f, --force` | This forcibly removes the image (or images). |
| `--no-prune` | This command does not delete untagged parents. |

This command removes one of the images from your machine:

```
$ docker rmi test
```

# The save command

The `save` command saves an image or repository in a tarball and this streams to the `stdout` file, preserving the parent layers and metadata about the image:

```
$ docker save -o codeit.tar code.it
```

The `-o` flag allows us to specify a file instead of streaming to the `stdout` file. It is used to create a backup that can then be used with the `docker load` command.

# The load command

The `load` command loads an image from a tarball, restoring the filesystem layers and the metadata associated with the image:

```
$ docker load -i codeit.tar
```

The `-i` flag allows us to specify a file instead of trying to get a stream from the `stdin` file.

# The export command

The `export` command saves the filesystem of a container as a tarball and streams to the `stdout` file. It flattens filesystem layers. In other words, it merges all the filesystem layers. All of the metadata of the image history is lost in this process:

```
$ sudo Docker export red_panda > latest.tar
```

Here, `red_panda` is the name of one of my containers.

# The import command

The `import` command creates an empty filesystem image and imports the contents of the tarball to it. You have the option of tagging it the image:

```
$ docker import URL|- [REPOSITORY[:TAG]]
```

URLs must start with `http`.

```
$ docker import http://example.com/test.tar.gz # Sample url
```

If you would like to import from a local directory or archive, you can use the - parameter to take the data from the `stdin` file:

```
$ cat sample.tgz | docker import – testimage:imported
```

# The tag command

You can add a `tag` command to an image. It helps identify a specific version of an image.

For example, the `python` image name represents `python:latest`, the latest version of Python available, which can change from time to time. But whenever it is updated, the older versions are tagged with the respective Python versions. So the `python:2.7` command will have Python 2.7 installed. Thus, the `tag` command can be used to represent versions of the images, or for any other purposes that need identification of the different versions of the image:

```
$ docker tag IMAGE [REGISTRYHOST/][USERNAME/]NAME[:TAG]
```

The `REGISTRYHOST` command is only needed if you are using a private registry of your own. The same image can have multiple tags:

```
$ docker tag shrikrishna/code.it:v1 shrikrishna/code.it:latest
```

Whenever you are tagging an image, follow the `username/repository:tag` convention.

Now, running the `docker images` command again will show that the same image has been tagged with both the `v1` and `latest` commands:

```
$ docker images
REPOSITORY            TAG     IMAGE ID      CREATED     VIRTUAL SIZE
shrikrishna/code.it   v1      a7cb6737a2f6  8 days ago  704.4 MB
shrikrishna/code.it   latest  a7cb6737a2f6  8 days ago  704.4 MB
```

# The login command

The `login` command is used to register or log in to a Docker registry server. If no server is specified, `https://index.docker.io/v1/` is the default:

```
$ Docker login [OPTIONS] [SERVER]
```

| Flag | Explanation |
|---|---|
| `-e, --email=""` | Email |
| `-p, --password=""` | Password |
| `-u, --username=""` | Username |

If the flags haven't been provided, the server will prompt you to provide the details. After the first login, the details will be stored in the `$HOME/.dockercfg` path.

# The push command

The `push` command is used to push an image to the public image registry or a private Docker registry:

```
$ docker push NAME[:TAG]
```

# The history command

The `history` command shows the history of the image:

```
$ docker history shykes/nodejs
IMAGE         CREATED        CREATED BY                      SIZE
6592508b0790  15 months ago  /bin/sh -c wget http://nodejs.  15.07 MB
0a2ff988ae20  15 months ago  /bin/sh -c apt-get install ...  25.49 MB
43c5d81f45de  15 months ago  /bin/sh -c apt-get update       96.48 MB
b750fe79269d  16 months ago  /bin/bash                       77 B
27cf78414709  16 months ago                                  175.3 MB
```

# The events command

Once started, the `events` command prints all the events that are handled by the `docker` daemon, in real time:

```
$ docker events [OPTIONS]
```

| Flag | Explanation |
|---|---|
| `--since=""` | This shows all events created since timestamp (in Unix). |
| `--until=""` | This stream events until timestamp. |

For example the `events` command is used as follows:

```
$ docker events
```

Now, in a different tab, run this command:

```
$ docker start code.it
```

Then run the following command:

```
$ docker stop code.it
```

Now go back to the tab running Docker events and see the output. It will be along these lines:

```
[2014-07-21 21:31:50 +0530 IST]
c7f2485863b2c7d0071477e6cb8c8301021ef9036afd4620702a0de08a4b3f7b: (from
dockerfile/nodejs:latest) start


[2014-07-21 21:31:57 +0530 IST]
c7f2485863b2c7d0071477e6cb8c8301021ef9036afd4620702a0de08a4b3f7b: (from
dockerfile/nodejs:latest) stop


[2014-07-21 21:31:57 +0530 IST]
c7f2485863b2c7d0071477e6cb8c8301021ef9036afd4620702a0de08a4b3f7b: (from
dockerfile/nodejs:latest) die
```

You can use flags such as `--since` and `--until` to get the event logs of specific timeframes.

# The wait command

The `wait` command blocks until a container stops, then prints its exit code:

```
$ docker wait CONTAINER(s)
```

# The build command

The build command builds an image from the source files at a specified path:

```
$ Docker build [OPTIONS] PATH | URL | -
```

| Flag | Explanation |
|---|---|
| `-t, --tag=""` | This is the repository name (and an optional tag) to be applied to the resulting image in case of success. |
| `-q, --quiet` | This suppresses the output, which by default is verbose. |
| `--rm=true` | This removes intermediate containers after a successful build. |
| `--force-rm` | This always removes intermediate containers, even after unsuccessful builds. |
| `--no-cache` | This command does not use the cache while building the image. |

This command uses a Dockerfile and a context to build a Docker image.

A Dockerfile is like a Makefile. It contains instructions on the various configurations and commands that need to be run in order to create an image. We will look at writing Dockerfiles in the next section.

It would be a good idea to read the section about Dockerfiles first and then come back here to get a better understanding of this command and how it works.

The files at the `PATH` or `URL` paths are called **context** of the build. The context is used to refer to the files or folders in the Dockerfile, for instance in the `ADD` instruction (and that is the reason an instruction such as `ADD ../file.txt` won't work. It's not in the context!).

When a GitHub URL or a URL with the `git://` protocol is given, the repository is used as the context. The repository and its submodules are recursively cloned in your local machine, and then uploaded to the `docker` daemon as the context. This allows you to have Dockerfiles in your private Git repositories, which you can access from your local user credentials or from the **Virtual Private Network** (**VPN**).

# Uploading to Docker daemon

Remember that Docker engine has both the `docker` daemon and the Docker client. The commands that you give as a user are through the Docker client, which then talks to the `docker` daemon (either through a TCP or a Unix socket), which does the necessary work. The `docker` daemon and Docker host can be in different hosts (which is the premise with which boot2Docker works), with the `DOCKER_HOST` environment variable set to the location of the remote `docker` daemon.

When you give a context to the `docker build` command, all the files in the local directory get tared and are sent to the `docker` daemon. The `PATH` variable specifies where to find the files for the context of the build in the `docker` daemon. So when you run `docker build .`, all the files in the current folder get uploaded, not just the ones listed to be added in the Dockerfile.

Since this can be a bit of a problem (as some systems such as Git and some IDEs such as Eclipse create hidden folders to store metadata), Docker provides a mechanism to ignore certain files or folders by creating a file called `.dockerignore` in the `PATH` variable with the necessary exclusion patterns. For an example, look up `https://github.com/docker/docker/blob/master/.dockerignore`.

If a plain URL is given or if the Dockerfile is streamed through the `stdin` file, then no context is set. In these cases, the `ADD` instruction works only if it refers to a remote URL.

Now let's build the `code.it` example image through a Dockerfile. The instructions on how to create this Dockerfile are provided in the *Dockerfile* section.

At this point, you would have created a directory and placed the Dockerfile inside it. Now, on your terminal, go to that directory and execute the `docker build` command:

```
$ docker build -t shrikrishna/code.it:docker Dockerfile .
Sending build context to Docker daemon  2.56 kB
Sending build context to Docker daemon
Step 0 : FROM Dockerfile/nodejs
 ---> 1535da87b710
```

```
Step 1 : MAINTAINER Shrikrishna Holla <s**a@gmail.com>
 ---> Running in e4be61c08592
 ---> 4c0eabc44a95
Removing intermediate container e4be61c08592
Step 2 : WORKDIR /home
 ---> Running in 067e8951cb22
 ---> 81ead6b62246
Removing intermediate container 067e8951cb22
. . . . .
. . . . .
Step 7 : EXPOSE  8000
 ---> Running in 201e07ec35d3
 ---> 1db6830431cd
Removing intermediate container 201e07ec35d3
Step 8 : WORKDIR /home
 ---> Running in cd128a6f090c
 ---> ba05b89b9cc1
Removing intermediate container cd128a6f090c
Step 9 : CMD     ["/usr/bin/node", "/home/code.it/app.js"]
 ---> Running in 6da5d364e3e1
 ---> 031e9ed9352c
Removing intermediate container 6da5d364e3e1
Successfully built 031e9ed9352c
```

Now, you will be able to look at your newly built image in the output of Docker images

```
REPOSITORY          TAG        IMAGE ID     CREATED      VIRTUAL SIZE
shrikrishna/code.it Dockerfile 031e9ed9352c 21 hours ago 1.02 GB
```

To see the caching in action, run the same command again

```
$ docker build -t shrikrishna/code.it:dockerfile .
Sending build context to Docker daemon  2.56 kB
Sending build context to Docker daemon
Step 0 : FROM dockerfile/nodejs
 ---> 1535da87b710
Step 1 : MAINTAINER Shrikrishna Holla <s**a@gmail.com>
 ---> Using cache
 ---> 4c0eabc44a95
```

```
Step 2 : WORKDIR /home
 ---> Using cache
 ---> 81ead6b62246
Step 3 : RUN     git clone https://github.com/shrikrishnaholla/code.
it.git
 ---> Using cache
 ---> adb4843236d4
Step 4 : WORKDIR code.it
 ---> Using cache
 ---> 755d248840bb
Step 5 : RUN     git submodule update --init --recursive
 ---> Using cache
 ---> 2204a519efd3
Step 6 : RUN     npm install
 ---> Using cache
 ---> 501e028d7945
Step 7 : EXPOSE  8000
 ---> Using cache
 ---> 1db6830431cd
Step 8 : WORKDIR /home
 ---> Using cache
 ---> ba05b89b9cc1
Step 9 : CMD     ["/usr/bin/node", "/home/code.it/app.js"]
 ---> Using cache
 ---> 031e9ed9352c
Successfully built 031e9ed9352c
```

Now experiment with this caching. Change one of the lines in the middle (the port number for example), or add a `RUN echo "testing cache"` line somewhere in the middle and see what happens.

An example of building an image using a repository URL is as follows:

```
$ docker build -t shrikrishna/optimus:git_url \ git://github.com/
shrikrishnaholla/optimus
Sending build context to Docker daemon 1.305 MB
Sending build context to Docker daemon
Step 0 : FROM        dockerfile/nodejs
```

```
 ---> 1535da87b710
Step 1 : MAINTAINER  Shrikrishna Holla
 ---> Running in d2aae3dba68c
 ---> 0e8636eac25b
Removing intermediate container d2aae3dba68c
Step 2 : RUN         git clone https://github.com/pesos/optimus.git
/home/optimus
 ---> Running in 0b46e254e90a
. . . . .
. . . . .
. . . . .
Step 5 : CMD         ["/usr/local/bin/npm", "start"]
 ---> Running in 0e01c71faa0b
 ---> 0f0dd3deae65
Removing intermediate container 0e01c71faa0b
Successfully built 0f0dd3deae65
```

# Dockerfile

We have seen how to create images by committing containers. What if you want to update the image with new versions of dependencies or new versions of your own application? It soon becomes impractical to do the steps of starting, setting up, and committing over and over again. We need a repeatable method to build images. In comes Dockerfile, which is nothing more than a text file that contains instructions to automate the steps you would otherwise have taken to build an image. `docker build` will read these instructions sequentially, committing them along the way, and build an image.

The `docker build` command takes this Dockerfile and a context to execute the instructions, and builds a Docker image. Context refers to the path or source code repository URL given to the `docker build` command.

A Dockerfile contains instructions in this format:

```
# Comment
INSTRUCTION arguments
```

Any line beginning with # will be considered as a comment. If a # sign is present anywhere else, it will be considered a part of arguments. The instruction is not case-sensitive, although it is an accepted convention for instructions to be uppercase so as to distinguish them from the arguments.

Let's look at the instructions that we can use in a Dockerfile.

# The FROM instruction

The `FROM` instruction sets the base image for the subsequent instructions. A valid Dockerfile's first non-comment line will be a `FROM` instruction:

```
FROM <image>:<tag>
```

The image can be any valid local or public image. If it is not found locally,the `Docker build` command will try to pull it from the public registry. The `tag` command is optional here. If it is not given, the `latest` command is assumed. If the incorrect `tag` command is given, it returns an error.

# The MAINTAINER instruction

The `MAINTAINER` instruction allows you to set the author for the generated images:

```
MAINTAINER <name>
```

# The RUN instruction

The `RUN` instruction will execute any command in a new layer on top of the current image, and commit this image. The image thus committed will be used for the next instruction in the Dockerfile.

The `RUN` instruction has two forms:

- The `RUN <command>` form
- The `RUN ["executable", "arg1", "arg2"...]` form

In the first form, the command is run in a shell, specifically the `/bin/sh -c <command>` shell. The second form is useful in instances where the base image doesn't have a `/bin/sh` shell. Docker uses a cache for these image builds. So in case your image build fails somewhere in the middle, the next run will reuse the previously successful partial builds and continue from the point where it failed.

The cache will be invalidated in these situations:

- When the `docker build` command is run with the `--no-cache` flag.
- When a non-cacheable command such as `apt-get update` is given. All the following `RUN` instructions will be run again.
- When the first encountered `ADD` instruction will invalidate the cache for all the following instructions from the Dockerfile if the contents of the context have changed. This will also invalidate the cache for the `RUN` instructions.

# The CMD instruction

The `CMD` instruction provides the default command for a container to execute. It has the following forms:

- The `CMD ["executable", "arg1", "arg2"...]` form
- The `CMD ["arg1", "arg2"...]` form
- The `CMD command arg1 arg2` ... form

The first form is like an exec and it is the preferred form, where the first value is the path to the executable and is followed by the arguments to it.

The second form omits the executable but requires the `ENTRYPOINT` instruction to specify the executable.

If you use the shell form of the `CMD` instruction, then the `<command>` command will execute in the `/bin/sh -c` shell.

If the user provides a command in `docker run`, it overrides the `CMD` command.

The difference between the `RUN` and `CMD` instructions is that a `RUN` instruction actually runs the command and commits it, whereas the `CMD` instruction is not executed during build time. It is a default command to be run when the user starts a container, unless the user provides a command to start it with.

For example, let's write a `Dockerfile` that brings a `Star Wars` output to your terminal:

```
FROM ubuntu:14.04
MAINTAINER shrikrishna
RUN apt-get -y install telnet
CMD ["/usr/bin/telnet", "towel.blinkenlights.nl"]
```

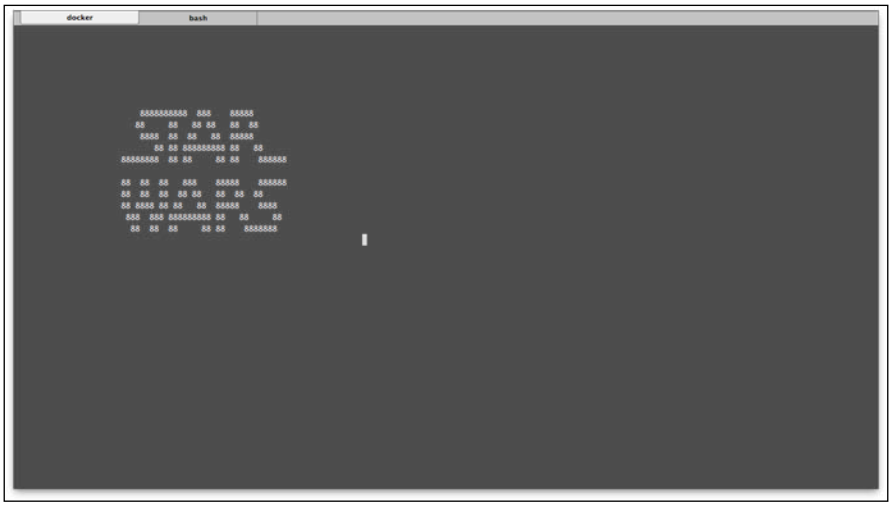
Save this in a folder named `star_wars` and open your terminal at this location. Then run this command:

```
$ docker build -t starwars .
```

Now you can run it using the following command:

```
$ docker run -it starwars
```

The following screenshot shows the `starwars` output:




Thus, you can watch **Star Wars** in your terminal!

This *Star Wars* tribute was created by Simon Jansen, Sten Spans, and Mike Edwards. When you've had enough, hold *Ctrl* + *]*. You will be given a prompt where you can type `close` to exit.

# The ENTRYPOINT instruction

The `ENTRYPOINT` instruction allows you to turn your Docker image into an executable. In other words, when you specify an executable in an `ENTRYPOINT`, containers will run as if it was just that executable.

The `ENTRYPOINT` instruction has two forms:

1. The `ENTRYPOINT ["executable", "arg1", "arg2"...]` form.
2. The `ENTRYPOINT command arg1 arg2` ... form.

This instruction adds an entry command that will not be overridden when arguments are passed to the `docker run` command, unlike the behavior of the `CMD` instruction. This allows arguments to be passed to the `ENTRYPOINT` instruction. The `docker run <image> -arg` command will pass the `-arg` argument to the command specified in the `ENTRYPOINT` instruction.

Parameters, if specified in the `ENTRYPOINT` instruction, will not be overridden by the `docker run` arguments, but parameters specified via the `CMD` instruction will be overridden.

As an example, let's write a Dockerfile with `cowsay` as the `ENTRYPOINT` instruction:

The `cowsay` is a program that generates ASCII pictures of a cow with a message. It can also generate pictures using premade images of other animals, such as Tux the Penguin, the Linux mascot.

```
FROM ubuntu:14.04
RUN apt-get -y install cowsay
ENTRYPOINT ["/usr/games/cowsay"]
CMD ["Docker is so awesomooooooooo!"]
```

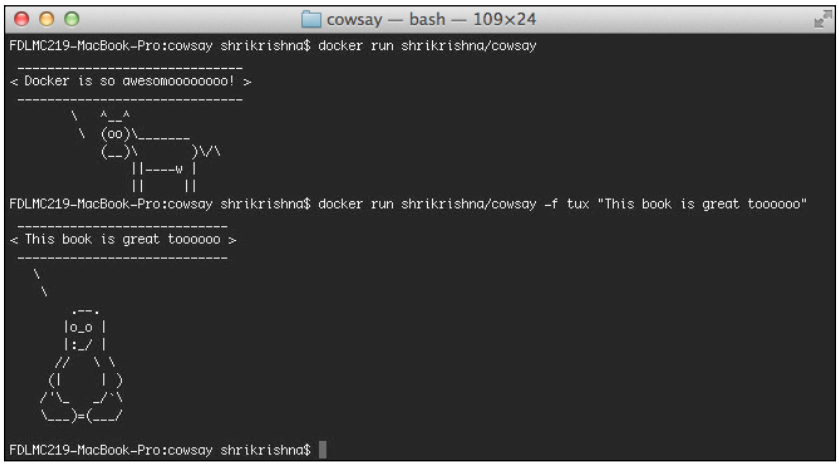
Save this with the name `Dockerfile` in a folder named `cowsay`. Then through terminal, go to that directory, and run this command:

```
$ docker build -t cowsay .
```

Once the image is built, run the following command:

```
$ docker run cowsay
```

The following screenshot shows the output of the preceding command:



If you look at the screenshot closely, the first run has no arguments and it used the argument we configured in the Dockerfile. However, when we gave our own arguments in the second run, it overrode the default and passed all the arguments (The `-f` flag and the sentence) to the `cowsay` folder.

If you are the kind who likes to troll others, here's a tip: apply the instructions given at `http://superuser.com/a/175802` to set up a pre-exec script (a function that is called whenever a command is executed) that passes every command to this Docker container, and place it in the `.bashrc` file. Now cowsay will print every command that it execute in a text balloon, being said by an ASCII cow!

# The WORKDIR instruction

The `WORKDIR` instruction sets the working directory for the `RUN`, `CMD`, and `ENTRYPOINT` Dockerfile commands that follow it:

```
WORKDIR /path/to/working/directory
```

This instruction can be used multiple times in the same Dockerfile. If a relative path is provided, the `WORKDIR` instruction will be relative to the path of the previous `WORKDIR` instruction.

# The EXPOSE instruction

The `EXPOSE` instruction informs Docker that a certain port is to be exposed when a container is started:

```
EXPOSE port1 port2 …
```

Even after exposing ports, while starting a container, you still need to provide port mapping using the `-p` flag to `Docker run`. This instruction is useful when linking containers, which we will see in *Chapter 3*, *Linking Containers*.

# The ENV instruction

The ENV command is used to set environment variables:

```
ENV <key> <value>
```

This sets the `<key>` environment variable to `<value>`. This value will be passed to all future `RUN` instructions. This is equivalent to prefixing the command with `<key>=<value>`.

The environment variables set using the `ENV` command will persist. This means that when a container is run from the resulting image, the environment variable will be available to the running process as well. The `docker inspect` command shows the values that have been assigned during the creation of the image. However, these can be overridden using the `$ docker run –env <key>=<value>` command.

# The USER instruction

The USER instruction sets the username or UID to use when running the image and any following the `RUN` directives:

```
USER xyz
```

# The VOLUME instruction

The `VOLUME` instruction will create a mount point with the given name and mark it as holding externally mounted volumes from the host or from other containers:

```
VOLUME [path]
```

Here is an example of the `VOLUME` instruction:

```
VOLUME ["/data"]
```

Here is another example of this instruction:

```
VOLUME /var/log
```

Both formats are acceptable.

# The ADD instruction

The `ADD` instruction is used to copy files into the image:

```
ADD <src> <dest>
```

The `ADD` instruction will copy files from `<src>` into the path at `<dest>`.

The `<src>` path must be the path to a file or directory relative to the source directory being built (also called the context of the build) or a remote file URL.

The `<dest>` path is the absolute path to which the source will be copied inside the destination container.

If you build by passing a Dockerfile through the `stdin` file (`docker build - < somefile`), there is no build context, so the Dockerfile can only contain a URL-based `ADD` statement. You can also pass a compressed archive through the `stdin` file (`docker build - < archive.tar.gz`). Docker will look for a Dockerfile at the root of the archive and the rest of the archive will get used as the context of the build.

The `ADD` instruction obeys the following rules:

- The `<src>` path must be inside the context of the build. You cannot use `ADD ../file as ..` syntax, as it is beyond the context.
- If `<src>` is a URL and the `<dest>` path doesn't end with a trailing slash (it's a file), then the file at the URL is copied to the `<dest>` path.
- If `<src>` is a URL and the `<dest>` path ends with a trailing slash (it's a directory), then the content at the URL is fetched and a filename is inferred from the URL and saved into the `<dest>/filename` path. So, the URL cannot have a simple path such as `example.com` in this case.
- If `<src>` is a directory, the entire directory is copied, along with the filesystem metadata.
- If `<src>` is a local tar archive, then it is extracted into the `<dest>` path. The result at `<dest>` is union of:
    - Whatever existed at the path `<dest>`.
    - Contents of the extracted tar archive, with conflicts in favor of the path `<src>`, on a file-by-file basis.
- If `<dest>` path doesn't exist, it is created along with all the missing directories along its path.

# The COPY instruction

The COPY instruction copies a file into the image:

```
COPY <src> <dest>
```

The `Copy` instruction is similar to the `ADD` instruction. The difference is that the `COPY` instruction does not allow any file out of the context. So, if you are streaming Dockerfile via the `stdin` file or a URL (which doesn't point to a source code repository), the `COPY` instruction cannot be used.

# The ONBUILD instruction

The `ONBUILD` instruction adds to the image a trigger that will be executed when the image is used as a base image for another build:

```
ONBUILD [INSTRUCTION]
```

This is useful when the source application involves generators that need to compile before they can be used. Any build instruction apart from the `FROM`, `MAINTAINER`, and `ONBUILD` instructions can be registered.

Here's how this instruction works:

1. During a build, if the `ONBUILD` instruction is encountered, it registers a trigger and adds it to the metadata of the image. The current build is not otherwise affected in any way.
2. A list of all such triggers is added to the image manifest as a key named `OnBuild` at the end of the build (which can be seen through the `Docker inspect` command).
3. When this image is later used as a base image for a new build, as part of processing the `FROM` instruction, the `OnBuild key` triggers are read and executed in the order they were registered. If any of them fails, the `FROM` instruction aborts, causing the build to fail. Otherwise, the `FROM` instruction completes and the build continues as usual.
4. Triggers are cleared from the final image after being executed. In other words they are not inherited by *grand-child builds*.

Let's bring `cowsay` back! Here's a Dockerfile with the `ONBUILD` instruction:

```
FROM ubuntu:14.04
RUN apt-get -y install cowsay
RUN apt-get -y install fortune
ENTRYPOINT ["/usr/games/cowsay"]
CMD ["Docker is so awesomooooooooo!"]
ONBUILD RUN /usr/games/fortune | /usr/games/cowsay
```

Now save this file in a folder named `OnBuild`, open a terminal in that folder, and run this command:

```
$ Docker build -t shrikrishna/onbuild .
```

We need to write another Dockerfile that builds on this image. Let's write one:

```
FROM shrikrishna/onbuild
```

```
RUN  apt-get moo
CMD ['/usr/bin/apt-get', 'moo']
```

The `apt-get moo` command is an example of Easter eggs typically found in many open source tools, added just for the sake of fun!

Building this image will now execute the `ONBUILD` instruction we gave earlier:

```
$ docker build -t shrikrishna/apt-moo apt-moo/
Sending build context to Docker daemon  2.56 kB
Sending build context to Docker daemon
Step 0 : FROM shrikrishna/onbuild
# Executing 1 build triggers
Step onbuild-0 : RUN /usr/games/fortune | /usr/games/cowsay
 ---> Running in 887592730f3d
 _________________________________
/ It was all so different before \
\ everything changed.            /
 ---------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
 ---> df01e4ca1dc7
 ---> df01e4ca1dc7
Removing intermediate container 887592730f3d
Step 1 : RUN  apt-get moo
 ---> Running in fc596cb91c2a
                 (__)
                 (oo)
           /------\/
          / |    ||
         *  /\---/\
            ~~   ~~
```

```
..."Have you mooed today?"...
 ---> 623cd16a51a7
Removing intermediate container fc596cb91c2a
Step 2 : CMD ['/usr/bin/apt-get', 'moo']
 ---> Running in 22aa0b415af4
 ---> 7e03264fbb76
Removing intermediate container 22aa0b415af4
Successfully built 7e03264fbb76
```

Now let's use our newly gained knowledge to write a Dockerfile for the `code.it` application that we previously built by manually satisfying dependencies in a container and committing. The Dockerfile would look something like this:

```
# Version 1.0
FROM dockerfile/nodejs
MAINTAINER Shrikrishna Holla <s**a@gmail.com>

WORKDIR /home
RUN     git clone \ https://github.com/shrikrishnaholla/code.it.git

WORKDIR code.it
RUN     git submodule update --init --recursive
RUN     npm install

EXPOSE  8000

WORKDIR /home
CMD     ["/usr/bin/node", "/home/code.it/app.js"]
```

Create a folder named `code.it` and save this content as a file named `Dockerfile`.

It is good practice to create a separate folder for every Dockerfile even if there is no context needed. This allows you to separate concerns between different projects. You might notice as you go that many Dockerfile authors club `RUN` instructions (for example, check out the Dockerfiles in `dockerfile.github.io`). The reason is that AUFS limits the number of possible layers to 42. For more information, check out this issue at `https://github.com/docker/docker/issues/1171`.

You can go back to the section on *Docker build* to see how to build an image out of this Dockerfile.

# Docker workflow - pull-use-modify-commit-push

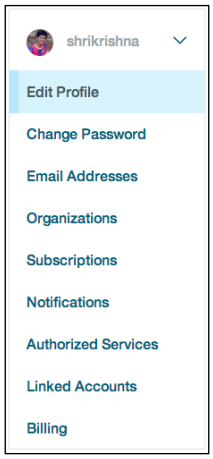
Now, as we are nearing the end of this chapter, we can guess what a typical Docker workflow is like:

1. Prepare a list of requirements to run your application.
2. Determine which public image (or one of your own) can satisfy most of these requirements, while also being well-maintained (this is important as you would need the image to be updated with newer versions whenever they are available).
3. Next, fulfill the remaining requirements either by running a container and executing the commands that fulfill the requirements (which can be installing dependencies, bind mounting volumes, or fetching your source code), or by writing a Dockerfile (which is preferable since you will be able to make the build repeatable).
4. Push your new image to the public Docker registry so that the community can use it too (or to a private registry or repository if needs be).

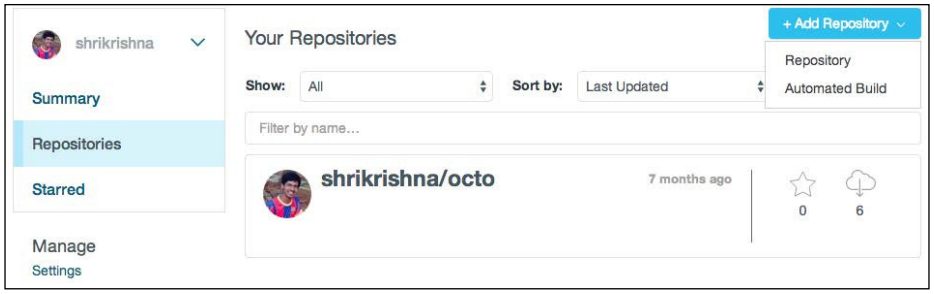
# Automated Builds

Automated Builds automate the building and updating of images from GitHub or BitBucket, directly on Docker Hub. They work by adding a `commit` hook to your selected GitHub or BitBucket repository, triggering a build and an update when you push a commit. So you need not manually build and push an image to Docker Hub every time you make an update. The following steps will show you how to do this:

1. To set up an Automated Build, log in to your Docker Hub account.

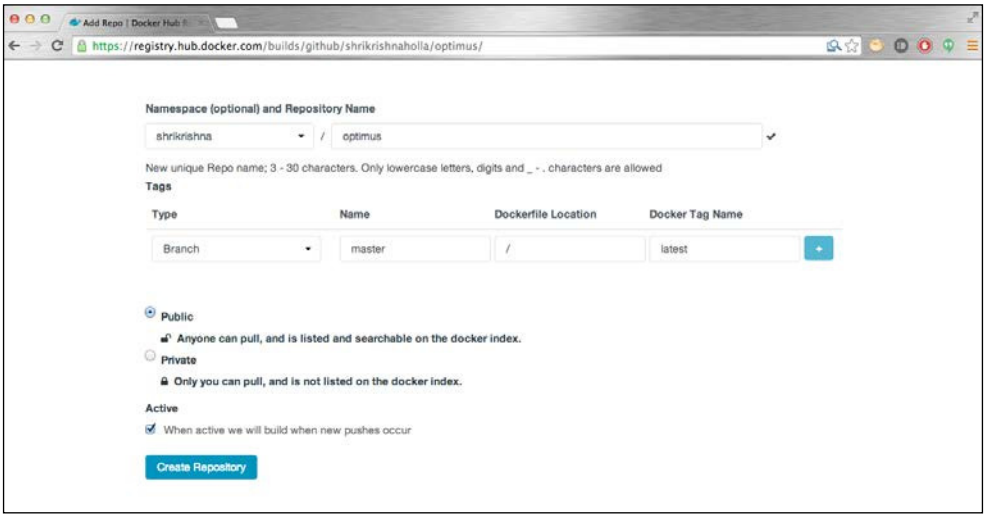



2. Link your GitHub or BitBucket account through the **Link Accounts** menu.
3. Select **Automated Build** in the **Add Repository** menu.



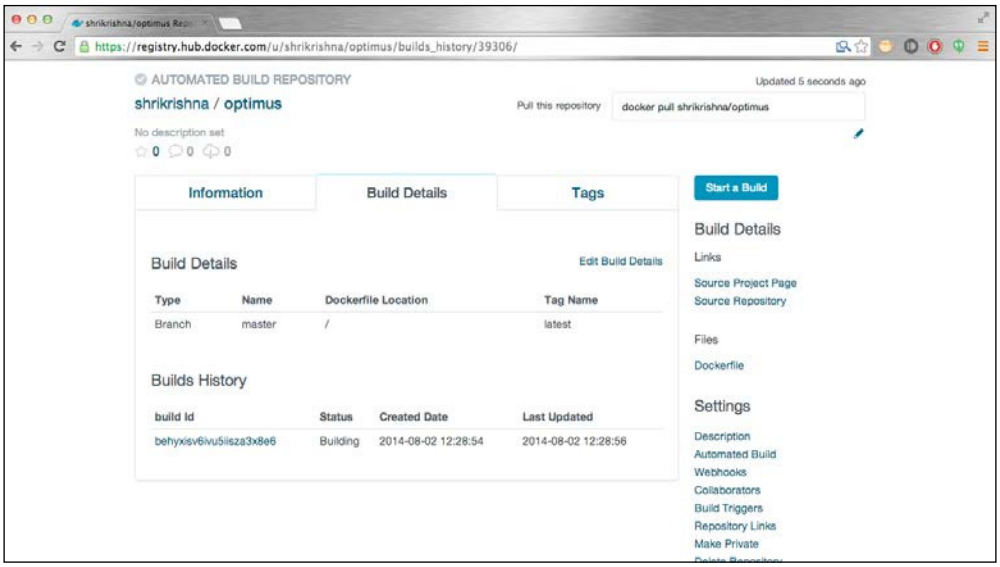

4. Select the GitHub or BitBucket project that has the Dockerfile you want to build. (You will need to authorize Docker Hub to access your repositories.)
5. Select the branch that contains the source code and the Dockerfile (the default is the master branch).
6. Give the Automated Build a name. This will be the name of the repository as well.

7. Assign an optional Docker tag to the Build. The default is the `lastest` tag.
8. Specify where the Dockerfile is located. The default is `/`.




Once configured, the automated build will trigger a build and you will be able to see it in the Docker Hub Registry in a few minutes. It will stay in sync with your GitHub and BitBucket repository until you deactivate the Automated Build yourself.

The build status and history can be seen in the Automated Builds page on your profile in Docker Hub.



Once you've created an Automated Build, you can deactivate or delete it.

You cannot, however, push to an Automated Build with the Docker `push` command. You can only manage it by committing code to your GitHub or BitBucket repository.

You can create multiple Automated Builds per repository and configure them to point to specific Dockerfile or Git branches.

# Build triggers

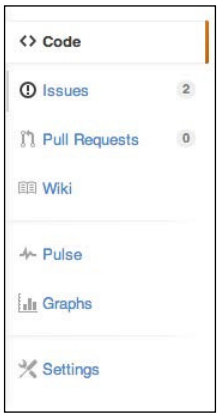
Automated Builds can also be triggered via a URL on Docker Hub. This allows you to rebuild an Automated Build image on demand.

# Webhooks

Webhooks are triggers that are called upon a successful build event. With a webhook, you can specify a target URL (such as a service that notifies you) and a JSON payload that will be delivered when the image is pushed. Webhooks are useful if you have a continuous-integration workflow.

To add a webhook to your Github repository, follow these steps:

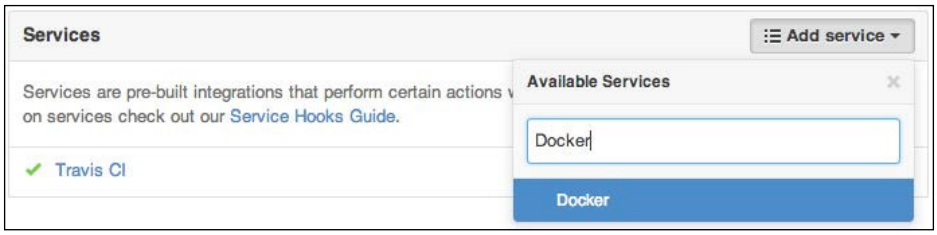
1. Go to **Settings** in your repository.



2. From the menu bar on the left, go to **Webhooks and Services**.


3. Click on **Add Service**.


4. In the text box that opens, enter **Docker** and select the service.




5. You're all set! Now a build will be triggered in Docker Hub whenever you commit to the repository.

# Summary

In this chapter, we looked at the **Docker** command-line tool and tried out the commands available. Then we figured out how to make builds repeatable using Dockerfile. Also, we automated this build process using Docker Hub's Automated Build service.

In the next chapter, we will try to gain more control over how our containers run by looking at the various commands that help us configure them. We will look at restraining the amount of resources (CPU, RAM, and storage) consumable by the container.

# 3
# Configuring Docker Containers

In the previous chapter, we saw all the different commands available in Docker. We took a look at examples covering how to pull images, run containers, attach images to containers, commit, and push an image to the repositories. We also learned how to write Dockerfiles to make building an image a repeatable process.

In this chapter, we will look closer at gaining control over how our containers run. Although Docker containers are sandboxed, this doesn't prevent a stray rogue process in one of the containers from hogging the resources available to other containers, including the host. For instance, beware of this command (don't run it):

```
$ docker run ubuntu /bin/bash -c ":(){ :|:& };:"
```

You would fork bomb the container as well as the host you run it on by running the preceding command.

The Wikipedia definition of a *fork bomb* is as follows:

> "*In computing, a fork bomb is a denial-of-service attack wherein a process continually replicates itself to deplete available system resources, causing resource starvation and slowing or crashing the system.*"

Since Docker is expected to be used in production, the possibility of one container stalling all others would be fatal. So there are mechanisms to limit the amount of resources that a container can take ownership of, which we will be looking at in this chapter.

In the previous chapter, we had a basic introduction to volumes when we talked about the `docker` run. We will now explore volumes in more detail and discuss why they are important and how to use them best. We will also try to change the storage driver being used by the `docker` daemon.

Another aspect is networking. While inspecting running containers, you might have noticed that Docker randomly chooses a subnet and allots an IP address (the default is usually the range 172.17.42.0/16). We will try to override this by setting our own subnet and explore other options available that help manage the networking aspects. In many scenarios, we will need to communicate between containers (imagine one container running your application and another running your database). Since IP addresses are not available at build time, we need a mechanism to dynamically discover the services running in other containers. We will be looking at ways to achieve this, both when the containers are running in the same host and when they are running in different hosts.

In short, in this chapter, we will be covering the following topics:

- Constraining resources
    - CPU
    - RAM
    - Storage
- Managing data in containers with volumes
- Configuring Docker to use a different storage driver
- Configuring networking
    - Port forwarding
    - A custom IP address range
- Linking containers
    - Linking within the same host using container links
    - Cross-host linking using ambassador containers

# Constraining resources

It is imperative for any tool that promises sandboxing capabilities to provide a mechanism to constrain resource allocation. Docker provides mechanisms to limit the amount of CPU memory and RAM that a container can use when it is being started.

# Setting CPU share

The amount of CPU share a container takes up can be controlled using the `-c` option in the `docker run` command:

```
$ docker run -c 10 -it ubuntu /bin/bash
```

The value, `10`, is the relative priority given to this container with respect to other containers. By default, all containers get the same priority, and hence the same ratio of CPU processing cycles, which you can check out by running `$ cat /sys/fs/cgroup/cpu/docker/cpu.shares` (add SSH to the boot2Docker VM before doing this if you are on OS X or Windows). However, you can give your own priority values when you run containers.

Is it possible to set CPU shares when a container is already running? Yes. Edit the file at `/sys/fs/cgroup/cpu/docker/<container-id>/cpu.shares` and enter the priority you want to give it.

If the location mentioned doesn't exist, find out where `cpu cgroup` is mounted by running the command `$ grep -w cgroup / proc/mounts | grep -w cpu`.

However, this is a hack, and might change in the future if Docker decides to change the way CPU sharing is implemented.More information about this can be found at `https://groups.google.com/forum/#!topic/docker-user/-pP8-KgJJGg`.

# Setting memory limit

Similarly, the amount of RAM that a container is allowed to consume can also be limited while starting the container:

```
$ docker run -m <value><optional unit>
```

Here, `unit` can be `b`, `k`, `m`, or `g`, representing bytes, kilobytes, megabytes, and gigabytes, respectively).

An example of a unit can be represented as follows:

```
$ docker run -m 1024m -dit ubuntu /bin/bash
```

This sets a memory limit of 1 GB for the container.

As in the case with limiting CPU shares, you can check the default memory limit by running this line of code:

```
$ cat /sys/fs/cgroup/memory/docker/memory.limit_in_bytes
18446744073709551615
```

As the filename states, the preceding code prints the limit in bytes. The value shown in the output corresponds to 1.8 x 1010 gigabytes, which practically means that there is no limit.

Is it possible to set a memory limit when a container is already running?

As with CPU shares, memory limit is enforced by the `cgroup` file, which means that we can change the limit on the fly by changing the value of the container's `cgroup` memory file:

```
$ echo 1073741824 > \
/sys/fs/cgroup/memory/docker/<container_id>/memory.limit_in_bytes
```

If the location of the `cgroup` file doesn't exist, find out where the file is mounted by running `$ grep -w cgroup /proc/mounts | grep -w memory`.

This is also a hack, and might change in the future if Docker decides to change the way memory limiting is internally implemented.

More information about this can be found at `https://groups.google.com/forum/#!topic/docker-user/-pP8-KgJJGg`.

# Setting a storage limit on the virtual filesystem (Devicemapper)

Limiting disk usage can be a bit tricky. There is no direct way to limit the amount of disk space a container can use. The default storage driver, AUFS, doesn't support disk quotas, at least not without hacks (the difficulty is because AUFS does not have its own block device. Visit `http://aufs.sourceforge.net/aufs.html` for in-depth information on how AUFS works). At the time of writing this book, Docker users who need disk quota opt for the `devicemapper` driver, which will allow each container to use up to a certain amount of disk space. But a more generic mechanism that works across storage drivers is under progress and may be introduced in future releases.

The `devicemapper` driver is a Linux kernel framework used to map block devices to higher-level virtual block devices.

The `devicemapper` driver creates a `thin` pool of storage blocks based on two block devices (think of them as virtual disks), one for data and another for metadata. By default, these block devices are created by mounting sparse files as loopback devices.

A **sparse file** is a file that contains mostly empty space. So a sparse file of 100 GB might actually just contain a few bytes in the beginning and the end (and occupy just these bytes on the disk), and yet be visible to an application as a 100 GB file. When reading sparse files, the filesystem transparently converts the empty blocks into real blocks filled with zero bytes at runtime. It tracks the location of the written and empty blocks through the file's metadata. In UNIX-like operating systems, a loopback device is a pseudo-device that makes a file accessible as a block device.

A `thin` pool is called so because it only marks storage blocks as used (from the pool) when you actually write to the blocks. Each container is provisioned a base thin device of a certain size, and the container is not allowed to accumulate data more than that size limit.

What are the default limits? The default limit for the `thin` pool is 100 GB. But since the loopback device used for this pool is a sparse file, it will initially not take up this much space.

The default size limit for the base device created for each container and image is 10 GB. Again, since this is sparse, it will not initially take up this much space on the physical disk. However, the amount of space it takes up increases with the increase in the size limit because, the larger the size of the block device, the greater is the (virtual) size of the sparse file, and the metadata it needs to store is more.

How can you change these default values? You can change these options using the `--storage-opts` option, which is available when running the `docker` daemon, with the `dm` (for `devicemapper`) prefix.

Before running any of the commands in this section, back up all your images with `docker save` and stop the `docker` daemon. It might also be wise to completely remove `/var/lib/docker` (the path where Docker stores image data).

## Devicemapper configurations

The various configurations available are as follows:

- `dm.basesize`: This specifies the size of the base device, which is used by containers and images. By default, this is set to 10 GB. The device created is sparse, so it will not initially occupy 10 GB. Instead, it will fill up as and when data is written into it, until it reaches the 10 GB limit:

  ```
  $ docker -d -s devicemapper --storage-opt dm.basesize=50G
  ```

- `dm.loopdatasize`: This is the size of the `thin` pool. The default size is 100 GB. It is to be noted that this file is sparse, so it will not initially take up this space; instead, it will fill up gradually as more and more data is written into it:

  ```
  $ docker -d -s devicemapper --storage-opt
  dm.loopdatasize=1024G
  ```

- `dm.loopmetadatasize`: As mentioned earlier, two block devices are created, one for data and another for metadata. This option specifies the size limit to use when creating this block device. The default size is 2 GB. This file is sparse too, so it will not initially take up the entire size. The recommended minimum size is 1 percent of the total pool size:

  ```
  $ docker -d -s devicemapper --storage-opt
  dm.loopmetadatasize=10G
  ```

- `dm.fs`: This is the filesystem type to use for the base device. The `ext4` and `xfs` filesystems are supported, although `ext4` is taken by default:

  ```
  $ docker -d -s devicemapper --storage-opt dm.fs=xfs
  ```

- `dm.datadev`: This specifies a custom block device to use (instead of loopback) for the `thin` pool. If you are using this option, it is recommended to specify block devices for both data and metadata to completely avoid using the loopback device:

  ```
  $ docker -d -s devicemapper --storage-opt dm.datadev=/dev/sdb1
  \-storage-opt dm.metadatadev=/dev/sdc1
  ```

There are more options available, along with a neat explanation of how all of this works at `https://github.com/docker/docker/tree/master/daemon/graphdriver/devmapper/README.md`.

Another great resource is a blog post on resizing containers by Docker contributor Jérôme Petazzoni at `http://jpetazzo.github.io/2014/01/29/docker-device-mapper-resize/`.

If you switch storage drivers, the older containers and images will no longer be visible.

At the beginning of this section, it was mentioned that there is a possibility to have quotas and still use AUFS through a hack. The hack involves creating a loopback filesystem based on the `ext4` filesystem on demand and bind mounting it as a volume specifically for the container:

```
$ DIR=$(mktemp -d)
$ DB_DIR=(mktemp -d)
$ dd if=/dev/zero of=$DIR/data count=102400
$ yes | mkfs -t ext4 $DIR/data
$ mkdir $DB_DIR/db
$ sudo mount -o loop=/dev/loop0 $DIR/data $DB_DIR
```

You can now bind mount the `$DB_DIR` directory to the container with the `-v` option of the `docker run` command:

```
$ docker run -v $DB_DIR:/var/lib/mysql mysql mysqld_safe.
```

# Managing data in containers with volumes

Some salient features of a volume in Docker are mentioned as follows:

- A volume is a directory that is separated from the container's `root` filesystem.
- It is managed directly by the `docker` daemon and can be shared across containers.
- A volume can also be used to mount a directory of the host system inside a container.
- Changes made to a volume will not be included when an image is updated from a running container.

- Since a volume is outside the filesystem of the container, it doesn't have the concept of data layers or snapshots. Hence, reads and writes happen directly on the volume.
- If multiple containers use the same volume, the volume persists until there is at least one container using it.

Creating a volume is easy. Just start a container with the `-v` option:

```
$ docker run -d -p 80:80 --name apache-1 -v /var/www apache.
```

Now note that volumes have no `ID` parameter, so you cannot exactly name a volume like you name a container or tag an image. However, the clause that says that a volume persists until at least one container uses it can be exploited, which introduces the concept of data-only containers.

Since Docker version 1.1, if you so wish, you can bind mount the whole filesystem of the host to a container using the -v option, like this:

`$ docker run -v /:/my_host ubuntu:ro ls /my_host`.

However, it is forbidden to mount to `/` of the container, so you cannot replace the `root` filesystem of the container, for security reasons.

# Data-only container

A data-only container is a container that does nothing except exposing a volume that other data-accessing containers can use. Data-only containers are used to prevent volumes from being destroyed if containers accessing the volume stop or crash due to an accident.

# Using volumes from another container

Once we start a container with a `-v` option, we have created a volume. We can share the volumes created by a container with other containers using the `--volumes-from` option. Possible use cases of this option can be backing up databases, processing logs, performing operations on user data, and so on.

# Use case – MongoDB in production on Docker

As a use case, say you want to use **MongoDB** in your production environment, you would be running a MongoDB server as well as a `cron` job, backing up your database snapshots at regular intervals.

MongoDB is a document database that provides high performance, high availability, and easy scalability. You can get more information about MongoDB at `http://www.mongodb.org`.

Let's see how make the MongoDB setup using `docker` volumes:

1. Firstly, we need a data-only container. The task of this container is only to expose the volume where MongoDB stores the data:

```
 $ docker run -v /data/db --name data-only mongo \
echo "MongoDB stores all its data in /data/db"
```

2. Then we need to run the MongoDB server, which uses the volume created by the data-only container:

```
$ docker run -d --volumes-from data-only -p 27017:27017 \
--name mongodb-server mongo mongod
```

The `mongod` command runs the MongoDB server and is usually run as a daemon/service. It is accessed through port `27017`.

3. Lastly, we will need to run the `backup` utility. In this case, we are just dumping the MongoDB data store to the current directory on the host:

```
$ docker run -d --volumes-from data-only --name mongo-backup \
 -v $(pwd):/backup mongo $(mkdir -p /backup && cd /backup &&
mongodump)
```

This is by no means an exhaustive example of setting up MongoDB in production. You might need a process that monitors the health of the MongoDB server. You will also need to make the MongoDB server container discoverable by your application containers (which we will learn in detail later).

# Configuring Docker to use a different storage driver

Before using a different storage driver, back up all your images with `docker save` and stop the `docker` daemon. Once you have backed up all your important images, remove `/var/lib/docker`. Once you change the storage driver, you can restore the saved images.

We are now going to change our default storage driver, AUFS, to two alternative storage drivers – `devicemapper` and `btrfs`.

## Using devicemapper as the storage driver

It is easy to switch to the `devicemapper` driver. Just start the `docker` daemon with the `-s` option:

```
$ docker -d -s devicemapper
```

Additionally, you can provide various `devicemapper` driver options with the `--storage-opts` flag. The various available options and examples for the `devicemapper` drivers have been covered under the *Constraining resources storage* section of this chapter.

If you are running on RedHat/Fedora that doesn't have AUFS out of the box, Docker will have been using `devicemapper` driver, which is available.

Once you have switched the storage driver, you can verify the change in it by running `docker info`.

## Using btrfs as the storage driver

To use btrfs as the storage driver, you have to first set it up. This section assumes you are running it on an Ubuntu 14.04 operating system. The commands may vary according to the Linux distribution you are running. The following steps will set up a block device with the `btrfs` filesystem:

1. Firstly, you need to install `btrfs` and its dependencies:

   ```
   # apt-get -y btrfs-tools
   ```

2. Next, you need to create a block device of the `btrfs` filesystem type:

   ```
   # mkfs btrfs /dev/sdb
   ```

3. Now create the directory for Docker (you should have backed up all important images and cleaned `/var/lib/docker` by this point.):

```
# mkdir /var/lib/docker
```

4. Then mount the btrfs block device at `/var/lib/docker`:

```
# mount /dev/sdb var/lib/docker
```

5. Check whether the mount is successful:

```
$ mount | grep btrfs

/dev/sdb on /var/lib/docker type btrfs (rw)
```

Source: `http://serverascode.com/2014/06/09/docker-btrfs.html`.

Now you can start the `docker` daemon with the `-s` option:

```
$ docker -d -s btrfs
```

Once you have switched the storage driver, you can verify the change in it by running the `docker info` command.

# Configuring Docker's network settings

Docker creates a separate network stack for each container and a virtual bridge (`docker0`) to manage network communication within the container, between the container and the host, and between two containers.

There are a few network configurations that can be set as arguments to the `docker run` command. They are as follows:

- `--dns`: A DNS server is what resolves a URL, such as `http://www.docker.io`, to the IP address of the server that is running the website.
- `--dns-search`: This allows you to set DNS search servers.

A DNS search server resolves `abc` to `abc.example.com` if `example.com` is set as the DNS search domain. This is useful if you have a lot of subdomains in your corporate website that you need to access frequently. It is too painful to repeatedly keep typing the entire URL. If you try to access a site that is not a fully qualified domain name (for example, `xyz.abc.com`.), it adds the search domains for the lookup. Source : `http://superuser.com/a/184366`.

- `-h` or `--hostname`: This allows you to set the hostname. This will be added as an entry to the `/etc/hosts` path against the host-facing IP of the container.
- `--link`: This is another option that can be specified while starting a container. It allows containers to communicate with other containers without needing to know their actual IP addresses.
- `--net`: This option allows you to set the network mode for the container. It can have four values:
    - `bridge` : This creates a network stack for the container on the docker bridge.
    - `none` : No networking stack will be created for this container. It will be completely isolated.
    - `container:<name|id>` : This uses another container's network stack.
    - `host` : This uses the host's network stack.

These values have side effects such as the local system services being accessible from the container. This option is considered insecure.

- `--expose`: This exposes the container's port without publishing it on the host.
- `--publish-all`: This publishes all exposed ports to the host's interfaces.
- `--publish`: This publishes a container's port to the host in the following format: `ip:hostPort:containerPort | ip::containerPort | hostPort:containerPort | containerPort`.

If `--dns` or `--dns-search` is not given, then the `/etc/resolv.conf` file of the container will be the same as the `/etc/resolv.conf` file of the host the daemon is running on.

However, there are some configurations that can be given to the `docker` daemon process too when you run it. They are mentioned as follows:

These options can only be supplied when starting the `docker` daemon and cannot be tweaked once it is running. This means you must provide these arguments along with the `docker -d` command.

- `--ip`: This option allows us to set the host's IP address at the container-facing `docker0` interface. As a result, this will be the default IP address used when binding container ports. For example this option can be shown as follows:

  ```
  $ docker -d --ip 172.16.42.1
  ```

- `--ip-forward`: This is a `Boolean` option. If it is set to `false`, the host running the daemon will not forward the packets between containers or from the outside world to the container, completely isolating it (from a network perspective).

This setting can be checked using the `sysctl` command:

```
$ sysctl net.ipv4.ip_forward
net.ipv4.ip_forward = 1.
```

- `--icc`: This is another `Boolean` option that stands for `inter-container communication`. If it is set to `false`, the containers will be isolated from each other, but will still be able to make general HTTP requests to package managers and so on.

How do you enable communication only between those two containers you need? Through links. We will explore links in detail in the *Linking containers* section.

- `-b or --bridge`: You can make Docker use a custom bridge instead of `docker0`. (The creation of a bridge is out of the scope of this discussion. However, if you are curious, you can find more information at `http://docs.docker.com/articles/networking/#building-your-own-bridge`.)
- `-H or --host`: This option can take multiple arguments. Docker has a RESTful API. The daemon acts as a server, and when you run client commands such as `run` and `ps`, it makes `GET` and `POST` requests to the server, which performs the necessary operations and returns a response. The `-H` flag is used to tell the `docker` daemon the channels it must listen to for client commands. The arguments can be as follows:
  - TCP sockets, represented in the form of `tcp://<host>:<port>`
  - UNIX socket in the form of `unix:///path/to/socket`

# Configuring port forwarding between container and host

Containers can make connections to the outside world without any special configurations, but the outside world is not allowed to peek into them. This is a security measure and is fairly obvious, since the containers are all connected to the host through a virtual bridge, thus effectively placing them in a virtual network. But what if you were running a service in a container that you wanted to expose to the outside world?

Port forwarding is the easiest way to expose services running in containers. It is always advisable to mention in the Dockerfile of an image the ports that need to be exposed. In earlier versions of Docker, it was possible to specify which host port the Dockerfile should be bound to in the Dockerfile itself, but this was dropped because sometimes, services already running in the host would interfere with the container. Now, you can still specify in a Dockerfile the ports that are intended to be exposed (with the `EXPOSE` instruction), but if you want to bind it to ports of your choice, you need to do this when starting the container.

There are two ways to start a container and bind its ports to host ports. They are explained as follows:

- `-P or --publish-all`: Starting a container using `docker run` with the `-P` option will publish all the ports that were exposed using the `EXPOSE` instruction in the image's Dockerfile. Docker will go through the exposed ports and bind them to a random port between `49000` and `49900`.
- `-p or --publish`: This option allows you to explicitly tell Docker which port on which IP should be bound to a port on a container (of course, one of the interfaces in the host should have this IP). Multiple bindings can be done by using the option multiple times:

    1. `docker run -p ip:host_port:container_port`
    2. `docker run -p ip::container_port`
    3. `docker run -p host_port:container_port`

# Custom IP address range

We've seen how to bind a container's port to a host's port, how to configure a container's DNS settings, and even how to set the host's IP address. But what if we wanted to set the subnet of the network between the containers and the host ourselves? Docker creates a virtual subnet in one of the available private ranges of IP addresses provided by RFC 1918.

Setting your own subnet range is marvelously easy. The `--bip` option of the `docker` daemon can be used to set the IP address of the bridge as well as the subnet in which it is going to create the containers:

```
$ docker -d --bip 192.168.0.1/24
```

In this case, we have set the IP address of `192.168.0.1` to the `docker` daemon and mentioned that it has to assign IP addresses to the containers in the subnet range `192.168.0.0/24` (that is, from `192.168.0.2` to `192.168.0.254`, a total of 252 possible IP addresses).

That's it! There are more advanced network configurations and examples at `https://docs.docker.com/articles/networking/`. Be sure to check them out.

# Linking containers

Binding container ports to host ports is all okay if you just have a plain web server that you want to expose to the Internet. Most production systems, however, are made of lots of individual components that are constantly communicating with each other. Components such as the database servers must not be bound to publicly visible IPs, but the containers running the frontend applications still need to discover the database containers and connect to them. Hardcoding a container's IP addresses in the application is neither a clean solution nor will it work because IP addresses are randomly assigned to the containers. So how do we solve this problem? The answer is as follows.

# Linking containers within the same host

A link can be specified when starting the container using the `--link` option:

```
$ docker run --link CONTAINER_IDENTIFIER:ALIAS . . .
```

How does this work? When a link option is given, Docker adds an entry to the container's `/etc/hosts` file, with the `ALIAS` command as the hostname and the IP address of the container named `CONTAINER_IDENTIFIER`.

The `/etc/hosts` file can be used to override DNS definitions, that is, to point a hostname to a certain IP address. During hostname resolution, `/etc/hosts` is checked before making a request to a DNS server.

For example the command line code is shown below:

```
$ docker run --name pg -d postgres
$ docker run --link pg:postgres postgres-app
```

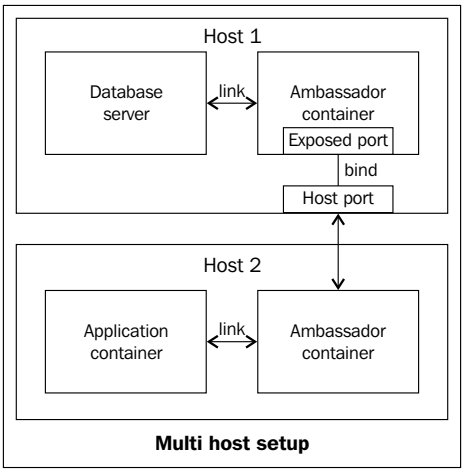
The preceding command runs a PostgreSQL server (whose Dockerfile exposes port 5432, PostgeSQL's default port) and the second container will link to it with the `postgres` alias.

PostgreSQL is a fully **ACID**-compliant, powerful open source object-relational database system.

# Cross-host linking using ambassador containers

Linking containers works fine when all the containers are within the same host, but Docker's containers might often be spread across hosts, and linking in these cases fails because the IP address of a container running in a different host is not known by the `docker` daemon running in the current host. Besides, links are static. This means that if a container restarts, its IP address changes and all containers linked to it will lose the connection. A portable solution is to use ambassador containers.

The following diagram displays the ambassador container:




**Multi host setup**

In this architecture, the database server in one host is exposed to the other. Here too, if the database container changes, only the ambassador container in the `host1` phase needs to be restarted.

# Use case - a multi-host Redis environment

Let's set up a multi-host Redis environment using the `progrium/ambassadord` command. There are other images that can be used as ambassador containers as well. They can be searched for either using the `docker search` command or at `https://registry.hub.docker.com`.

Redis is an open source, networked, in-memory, key-value data store with optional durability. It is known for its fast speed, both for reads and writes.

In this environment, there are two hosts, `Host 1` and `Host 2`. `Host 1` has an IP address of `192.168.0.100` and is private (not exposed to the public Internet). `Host 2` is at 192.168.0.1 and is bound to a public IP. This is the host that runs your frontend web application.

To try this example, start two virtual machines. If you use Vagrant, I suggest using an Ubuntu image with Docker installed. If you have Vagrant v1.5, you can use Phusion's Ubuntu image by running `$ vagrant init phusion/ubuntu-14.04-amd64`.

## Host 1

In the first host, run the following command:

```
$ docker run -d --name redis --expose 6379 dockerfile/redis
```

This command starts a Redis server and exposes port `6379` (which is the default port the Redis server runs at), but doesn't bind it to any host port.

The following command starts an ambassador container, links to the Redis server and binds the port 6379 to port 6379 of its private network's IP address (which in this case happens to be 192.168.0.100). This is still not public because the host is private (not exposed to public Internet):

```
$ docker run -d --name redis-ambassador-h1 \
  -p 192.168.0.100:6379:6379 --link redis:redis \
  progrium/ambassadord --links
```

### Host 2

In another host (another VM if you are using Vagrant in development), run the following command:

```
$ docker run -d --name redis-ambassador-h2 --expose 6379 \
progrium/ambassadord 192.168.0.100:6379
```

This ambassador container listens to the port of the destination IP, which in this case is Host 1's IP address. We have exposed port 6379 so that it can be now hooked to by our application container:

```
$ docker run -d --name application-container \
--link redis-ambassador-h2:redis myimage mycommand
```

This would be the container that would be exposed to the public on the Internet. As the Redis server is running in a private host, it cannot be attacked from outside the private network.

## Summary

In this chapter, we saw how to provision resources such as CPU, RAM, and storage in a Docker container. We also discussed how to use volumes and volume containers to manage persistent data produced by applications in containers. We realized what goes into switching storage drivers used by Docker and the various networking configurations and their relevant use cases. Lastly, we saw how to link containers both within a host and across hosts.

In the next chapter, we will look at the tools and approaches that will help when we are thinking about deploying our application using Docker. Some of the things we will be looking at are coordination of multiple services, service discovery, and Docker's remote API. We will also cover security considerations.

# 4
# Automation and Best Practices

At this point, we now know how to set up Docker in our development environments, are comfortable with the Docker commands, and have a good idea about the kind of situations Docker is suitable for. We also have an idea on how to configure Docker and its containers to suit all our needs.

In this chapter, we will focus on the various usage patterns that will help us deploy our web applications in production environments. We will begin with Docker's remote API because logging in to a production server and running commands is always considered dangerous. So, it is best to run an application that monitors and orchestrates the containers in a host. There are a host of orchestration tools available for Docker today, and with the announcement of v1.0, Docker also announced a new project, **libswarm**, which gives a standard interface to manage and orchestrate distributed systems, which will be another topic we will be delving into.

Docker developers recommend running only one process per container. This is difficult if you want to inspect an already running container. We will look at a command that allows us to inject a process into an already running container.

As your organization grows, so does the load, and you will need to start thinking about scaling. Docker in itself is meant to be used in a single host, but by using a host of tools such as `etcd` and `coreos`, you can easily run a bunch of Docker hosts in a cluster and discover every other container in that cluster.

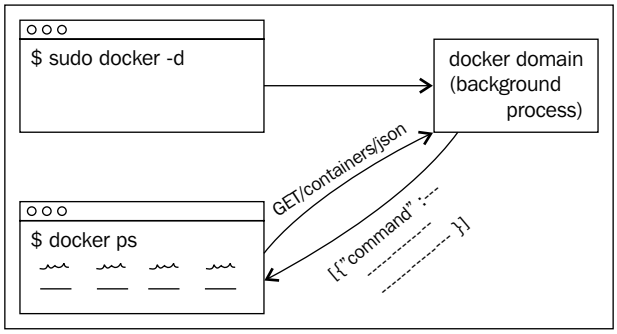
Every organization that has a web application running in production knows the importance of security. In this chapter, we are going to talk about the security aspects with respect to not only the `docker` daemon, but also the various Linux features used by Docker. To summarize, in this chapter, we will look at the following:

- Docker remote API
- Injecting processes into containers with the Docker exec command
- Service discovery
- Security

# Docker remote API

The Docker binary can run both as a client and as a daemon. When Docker is run as a daemon, it attaches itself to a Unix socket at `unix:///var/run/docker.sock` by default (this can be changed when starting docker, of course) and accepts commands over REST. The same Docker binary can then be used to run all the other commands (which is nothing but the client making REST calls to the `docker` daemon).

A diagram of the `docker` daemon is shown as follows:




This section will mainly be explained with examples as we have already encountered the working of these operations when we looked at the Docker commands.

To test these APIs, run the `docker` daemon at a TCP port like this:

```
$ export DOCKER_HOST=tcp://0.0.0.0:2375
$ sudo service docker restart
$ export DOCKER_DAEMON=http://127.0.0.1:2375 # or IP of your host
```

This is not going to be a reference guide, since we have already covered the features available with Docker when we disussed Docker commands in *Chapter 2, Docker CLI and Dockerfile*. Instead, we will be covering a select few APIs and you can look up the rest at `docs.docker.com/reference/api/docker_remote_api`.

Before we start, let's ensure that the `docker` daemon is responding to our requests:

```
$ curl $DOCKER_DAEMON/_ping
OK
```

Alright, everything is fine. Let's get going.

# Remote API for containers

Let's first look at the a few endpoints available that help create and manage containers.

## The create command

The `create` command creates a container:

```
$ curl \
> -H "Content-Type: application/json" \
> -d '{"Image":"ubuntu:14.04",\
> "Cmd":["echo", "I was started with the API"]}' \
> -X POST $DOCKER_DAEMON/containers/create?\
> name=api_container;
{"Id":"4e145a6a54f9f6bed4840ac730cde6dc93233659e7eafae947efde5caf583f
c3","Warnings":null}
```

The `curl` utility is a simple Unix utility that can be used to construct HTTP requests and analyze responses.

Here we make a `POST` request to the `/containers/create` endpoint and pass a `JSON` object containing the details of the image we want the container to be based upon and the command we expect the container to run.

Type of request: POST

The `JSON` data sent along with the `POST` request:

| Parameter | Type | Explanation |
|---|---|---|
| **`config`** | `JSON` | Describes the configuration of the container to start |

Query parameters for the POST request:

| Parameter | Type | Explanation |
|---|---|---|
| **`name`** | `String` | This assigns a name to the container. It must match the `/?[a-zA-Z0-9_-]+` regular expression. |

The following table shows the status code of the responses:

| Status code | Meaning |
|---|---|
| **`201`** | No error |
| **`404`** | No such container |
| **`406`** | Impossible to attach (container not running) |
| **`500`** | Internal server error |

# The list command

The `list` command gets a list of containers:

```
$ curl $DOCKER_DAEMON/containers/json?all=1\&limit=1

[{"Command":"echo 'I was started with the
API'","Created":1407995735,"Id":"96bdce1493715c2ca8940098db04b99e3629
4a333ddacab0e04f62b98f1ec3ae","Image":"ubuntu:14.04","Names":["/api_c
ontainer"],"Ports":[],"Status":"Exited (0) 3 minutes ago"}
```

This is a `GET` request API. A request to `/containers/json` will return a `JSON` response containing a list of containers that fulfill the criteria. Here, passing the `all` query parameter will list containers that are not running as well. The `limit` parameter is the number of containers that will be listed in the response.

There are query parameters that you can provide with these API calls, which can fine-tune the responses.

Type of Request: GET

| Parameter | Type | Explanation |
|---|---|---|
| **all** | 1/True/true or 0/False/false | This tells whether all containers should be shown. Only running containers are shown by default. |
| **limit** | Integer | This shows the last [*n*] containers, including non running containers. |
| **since** | Container ID | This only shows containers started since [x], including non running ones. |
| **before** | Container ID | This only shows containers started before [x], including non running ones. |
| **size** | 1/True/true or 0/False/false | This tells whether container sizes should be shown in the responses or not. |

Status codes of the response follow relevant **Request For Comments** (**RFC**) 2616:

| Status code | Meaning |
|---|---|
| **200** | No error |
| **400** | Bad parameter and client error |
| **500** | Server error |

Other endpoints for containers can be read about at `docs.docker.com/reference/api/docker_remote_api_v1.13/#21-containers`.

# Remote API for images

Similar to containers, there are APIs to build and manage images as well.

## Listing the local Docker images

The following command lists the local images:

```
$ curl $DOCKER_DAEMON/images/json
```

```
[{"Created":1406791831,"Id":"7e03264fbb7608346959378f270b32bf31daca14d15e
9979a5803ee32e9d2221","ParentId":"623cd16a51a7fb4ecd539eb1e5d9778
c90df5b96368522b8ff2aafcf9543bbf2","RepoTags":["shrikrishna/apt-
moo:latest"],"Size":0,"VirtualSize":281018623}
,{"Created":1406791813,"Id":"c5f4f852c7f37edcb75a0b712a16820bb8c729a6
a5093292e5f269a19e9813f2","ParentId":"ebe887219248235baa0998323342f7f
5641cf5bff7c43e2b802384c1cb0dd498","RepoTags":["shrikrishna/onbuild:l
atest"],"Size":0,"VirtualSize":281018623}
,{"Created":1406789491,"Id":"0f0dd3deae656e50a78840e58f63a5808ac53cb4
dc87d416fc56aaf3ab90c937","ParentId":"061732a839ad1ae11e9c7dcaa183105
138e2785954ea9e51f894f4a8e0dc146c","RepoTags":["shrikrishna/optimus:g
it_url"],"Size":0,"VirtualSize":670857276}
```

This is a `GET` request API. A request to `/images/json` will return a `JSON` response containing a list that contains details of the images that fulfill the criteria.

Type of request: GET

| Parameter | Type | Explanation |
|---|---|---|
| `all` | 1/`True`/`true` or 0/`False`/`false` | This tells whether even intermediary containers should be shown. False by default. |
| `filters` | `JSON` | These are used to provide a filtered list of images. |

Other endpoints for images can be read about at `docs.docker.com/reference/api/docker_remote_api_v1.13/#22-images`.

# Other operations

There are other APIs too, such as the ping API we checked at the beginning of this section. Some of them are explored in the following section.

## Getting system-wide information

The following command gets the system-wide information on Docker. This is the endpoint that handles the `docker info` command:

```
$ curl $DOCKER_DAEMON/info
```

```
{"Containers":41,"Debug":1,"Driver":"aufs","DriverStatus":[["Root
Dir","/mnt/sda1/var/lib/docker/aufs"],["Dirs","225"]],"ExecutionDrive
r":"native-
0.2","IPv4Forwarding":1,"Images":142,"IndexServerAddress":"https://in
dex.docker.io/v1/","InitPath":"/usr/local/bin/docker","InitSha1":"","
KernelVersion":"3.15.3-
tinycore64","MemoryLimit":1,"NEventsListener":0,"NFd":15,"NGoroutines
":15,"Sockets":["unix:///var/run/docker.sock","tcp://0.0.0.0:2375"],"
SwapLimit":1}
```

# Committing an image from a container

The following command commits an image from a container:

```
$ curl \
> -H "Content-Type: application/json" \
> -d '{"Image":"ubuntu:14.04",\
> "Cmd":["echo", "I was started with the API"]}' \
> -X POST $DOCKER_DAEMON/commit?\
> container=96bdce149371\
> \&m=Created%20with%20remote%20api\&repo=shrikrishna/api_image;


{"Id":"5b84985879a84d693f9f7aa9bbcf8ee8080430bb782463e340b241ea760a5a
6b"}
```

Commit is a `POST` request to the `/commit` parameter with data about the image it's based on and the command associated with the image that will be created on commit. Key pieces of information include the `container ID` parameter to commit, the commit message, and the repository it belongs to, all of which are passed as query parameters.

Type of request: POST

The `JSON` data sent along with the `POST` request:

| Parameter | Type | Explanation |
|---|---|---|
| **config** | JSON | This describes the configuration of the container to commit |

The following table shows query parameters for the `POST` request:

| Parameter | Type | Explanation |
|---|---|---|
| **container** | Container ID | The ID of the container you intend to commit |
| **repo** | String | The repository to create the image in |
| **tag** | String | The tag for the new image |
| **m** | String | Commit message |
| **author** | String | Author information |

The following table shows the status code of the responses:

| Status code | Meaning |
|---|---|
| `201` | No error |
| `404` | No such container |
| `500` | Internal server error |

## Saving the image

Get a tarball backup of all the images and metadata of a repository from the following command:

```
$ curl $DOCKER_DAEMON/images/shrikrishna/code.it/get > \
> code.it.backup.tar.gz
```

This will take some time, as the image has to be first compressed into a tarball and then streamed, but then it will be saved in the tar archive.

Other endpoints can be read about at `docs.docker.com/reference/api/docker_remote_api_v1.13/#23-misc`.

# How docker run works

Now that we have realized that every Docker command that we run is nothing but a series of RESTful operations carried out by the client, let's enhance our understanding of what happens when you run a `docker run` command:

1. To create an API, `/containers/create` parameter is called.
2. If the status code of the response is 404, it means the image doesn't exist. Try to pull the image using `/images/create` parameter and go back to step 1.
3. Get the `ID` of the created container and start it using `/containers/(id)/start` parameter.

The query parameters to these API calls will depend on the flags and arguments passed to the `docker run` command.

# Injecting processes into containers with the Docker execute command

During the course of your explorations of Docker, you may have wondered whether the single command per container rule enforced by Docker is limiting its capabilities. In fact, you might be forgiven for assuming that a Docker container runs only a single process. But no! A container can run any number of processes, but can only start with one command and the container lives as long as the process associated with the command does. This restriction has been enforced because Docker believes in the philosophy of one app per container. Instead of loading everything in a single container, a typical Docker-reliant application architecture will consist of multiple containers, each running a specialized service, all linked together. This helps keep the container light, makes debugging easier, reduces the attack vectors, and ensures that if one service goes down, others aren't affected.

Sometimes, however, you might need to look into the container while it is running. Over time, a number of approaches have been taken by the Docker community to debug running containers. Some members loaded SSH into the container and ran a process management solution such as **supervisor** to run the SSH + application server. Then came tools such as **nsinit** and **nsenter** that helped spawn a shell in the namespace the container was running in. However, all of these solutions were hacks. So with v1.3, Docker decided to provide the `docker exec` command, a safe alternative that could debug running containers.

The `docker exec` command, allows a user to spawn a process inside their Docker container via the Docker API and CLI, for example:

```
$ docker run -dit --name exec_example -v $(pwd):/data -p 8000:8000
dockerfile/python python -m SimpleHTTPServer
$ docker exec -it exec_example bash
```

The first command starts a simple file server container. The container is sent to the background with the `-d` option. In the second command, with `docker exec`, we log in to the container by creating a bash process inside it. Now we will be able to inspect the container, read the log (if we have logged in to a file), run diagnostics (if the need to inspect arises because of a bug), and so on.

Docker still hasn't moved from its one-app-per-container philosophy. The `docker exec` command exists just to provide us with a way to inspect containers, which otherwise would've required workarounds or hacks.

# Service discovery

Docker assigns an IP to a container dynamically from a pool of available addresses. While this is good in some ways, it creates a problem when you are running containers that need to communicate with each other. You just cannot know when building an image what its IP address is going to be. Your first instinct might be to start the containers, then log in to them (via `docker exec`), and set the IP addresses of the other containers manually. But remember, this IP address can change when a container restarts, so then you would have to manually log in to each container and enter the new IP address. Could there be a better way? Yes, there is.

Service discovery is a collection of everything that needs to be done to let services know how to find and communicate with other services. Under service discovery, containers do not know their peers when they are just started. Instead, they discover them dynamically. This should work both when the containers are in the same host as well as when they are in a cluster.

There are two techniques to achieve service discovery:

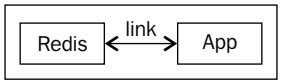
- Using default Docker features such as  names and links
- Using a dedicated service such as `Etcd` or `Consul`

# Using Docker names, links, and ambassador containers

We learned how to link conatiners in the section titled *Linking Containers* in *Chapter 3*, *Configuring Docker Containers*. To refresh your memory, this is how it works.

# Using links to make containers visible to each other

The use of links is shown in the following diagram:




Link allows a container to connect to another container without any need to hardcode its IP address. It is achieved by inserting the first container's IP address in `/etc/hosts` when starting the second container.

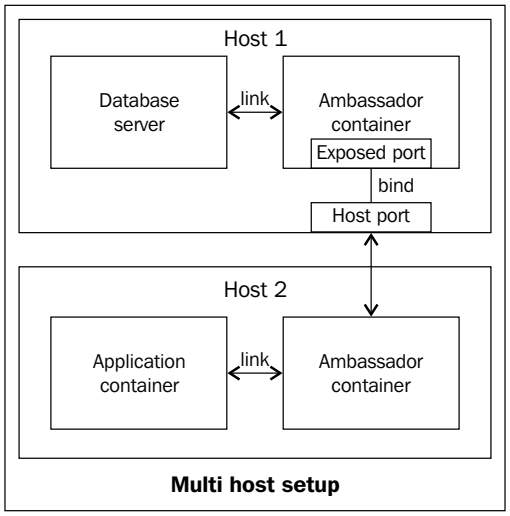
A link can be specified when starting the container using the `--link` option:

```
$ docker run --link CONTAINER_IDENTIFIER:ALIAS . . .
```

You can find out more about linking in *Chapter 3, Configuring Docker Containers*.

# Cross-host linking using ambassador containers

The following diagram represents cross-host linking using ambassador containers:



Ambassador containers are used to link containers across hosts. In this architecture, you can restart/replace the database container without needing to restart the application container.

You can find out more about ambassador containers in *Chapter 3, Configuring Docker Containers*.

# Service discovery using etcd

Why do we need specialized solutions for service discovery? While ambassador containers and links solve the problem of finding containers without needing to know their IP addresses, they do have one fatal flaw. You still need to manually monitor the health of the containers.

Imagine a situation where you have a cluster of backend servers and frontend servers linked to them via ambassador containers. If one of the servers goes down, the frontend servers still keep trying to connect to the backend server, because as far as they are concerned, that is the only available backend server, which is of course wrong.

Modern service discovery solutions such as `etcd`, `Consul`, and `doozerd` do more than merely providing the right IP addresses and ports. They are, in effect, distributed key-value stores, but are fault tolerant and consistent and handle master election in the event of failure. They can even act as lock servers.

The `etcd` service is an open source, distributed key-value store developed by **CoreOS**. In a cluster, the `etcd` client runs on each machine in the cluster. The `etcd` service gracefully handles master election during network partitions and the loss of the current master.

Your applications can read and write data to the `etcd` service. Common examples for `etcd` services are storing database connection details, cache settings, and so on.

Features of the `etcd` service are listed here:

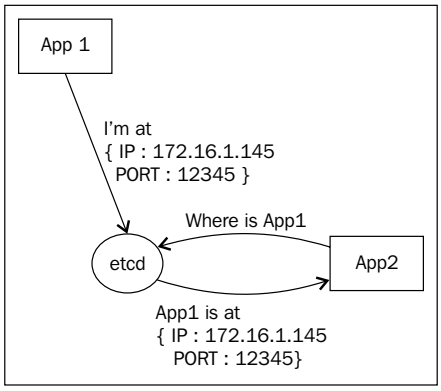
- Simple, curlable API (HTTP + JSON)
- Optional **Secure Sockets Layer** (**SSL**) client certificate authentication
- Keys support **Time To Live** (**TTL**)

The `Consul` service is a great alternative to the `etcd` service. There is no reason why one should be chosen over the other. This section is just meant to introduce you to the concept of service discovery.

We use the `etcd` service in two stages as follows:

1. We register our services with the `etcd` service.
2. We do a lookup to find services thus registered.

   The following diagram shows the `etcd` service:




This seems like a simple task to do, but building a solution that is fault tolerant and consistent is not simple. You will also need to be notified in case of failure of a service. If you run the service discovery solution itself in a naive centralized manner, it might become a single point of failure. So, all instances in a cluster of service discovery servers need to be synchronized with the right answer, which makes for interesting approaches. The team at CoreOS developed a consensus algorithm called **Raft** to solve this problem. You can read more about it at `http://raftconsensus.github.io`.

Let's look at an example to get a lay of the land. In this example, we will run the `etcd` server in a container and see how easy it is to register a service and discover it.

1. Step 1: Run the `etcd` server:

   ```
   $ docker run -d -p 4001:4001 coreos/etcd:v0.4.6 -name myetcd
   ```

2. Step 2: Once the image is downloaded and the server starts, run the following command to register a message:

   ```
   $ curl -L -X PUT http://127.0.0.1:4001/v2/keys/message -d
   value="Hello"
   ```

   ```
   {"action":"set","node":{"key":"/message","value":"Hello","modified
   Index":3,"createdIndex":3}}
   ```

   This is nothing but a `PUT` request to the server at the `/v2/keys/message` path (`message` being the key here).

3. Step 3: Get the key back with the following command:

```
$ curl -L http://127.0.0.1:4001/v2/keys/message
{"action":"get","node":{"key":"/message","value":"Hello","modified
Index":4,"createdIndex":4}}
```

You can go ahead and experiment by changing the value, trying an invalid key, and so on. You will find that the responses are in `JSON`, which means you can easily integrate it with your application without needing to use any libraries.

But how would I use it in my application? If your application needs to run multiple services, they can be connected together with links and ambassador containers, but if one of them becomes unavailable or needs to be redeployed, a lot of work needs to be done to restore the links.

Now imagine that your services use the `etcd` service. Every service registers its IP address and port number against its name and discovers other services by their names (that are constant). Now, if a container restarts because of a crash/redeployment, the new container will register against the modified IP address. This will update the value that the `etcd` service returns for subsequent discovery requests. However, this means that a single `etcd` server can also be a single point of failure. The solution for this is to run a cluster of `etcd` servers. This is where the Raft consensus algorithm, developed by CoreOS (the team that created `etcd` service), comes in. A complete example of an application service being deployed with the `etcd` service can be found at `http://jasonwilder.com/blog/2014/07/15/docker-service-discovery/`

# Docker Orchestration

As soon as you go beyond simple applications to complex architectures, you will start using tools and services such as `etcd`, `consul`, and `serf`, and you will notice that all of them come with their own set of APIs, even though they have overlapping features. If you set up your infrastructure to one set of tooling and find a need to switch, it takes considerable effort, sometimes even changes in the code, to switch vendors. Such situations can lead to vendor lock-in, which would ruin a promising ecosystem that Docker has managed to create. To provide a standard interface to these service providers so that they can almost be used as plug-and-play solutions, Docker has released a suite of orchestration services. In this section, we will take a look at them. Note, however, that at the time of writing this book, these projects (Machine, Swarm, and Compose) are still in Alpha and in active development.

# Docker Machine

Docker Machine aims to provide a single command to take you from zero-to-Docker project.

Before Docker Machine, if you intended to start working with Docker on a new host, be it a virtual machine or a remote host in an infrastructure provider such as **Amazon Web Services** (**AWS**) or Digital Ocean, you would have to log in to the instance, and run the setup and configuration commands specific to the operating system running in it.

With Docker Machine, whether provisioning the `docker` daemon on a new laptop, on virtual machines in the data center, or on a public cloud instance, the same, single command gets the target host ready to run Docker containers:

```
$ machine create -d [infrastructure provider] [provider options]
[machine name]
```

Then you can manage multiple Docker hosts from the same interface regardless of their location and run any Docker command on them.

Apart from this, the machine also has pluggable backends, which makes adding support to infrastructure providers easy, while retaining the common user-facing API. Machine ships by default with drivers to provision Docker locally with Virtualbox as well as remotely on Digital Ocean instances.

Note that Docker Machine is a separate project from the Docker Engine. You can find the updated details about this project on its Github page at `https://github.com/docker/machine`.

# Swarm

**Swarm** is a native clustering solution provided by Docker. It takes Docker Engine and extends it to enable you to work on a cluster of containers. With Swarm, you can manage a resource pool of Docker hosts and schedule containers to run transparently on top, automatically managing workload and providing failover services.

To schedule, it takes the container's resource requirements, looks at the available resources in the hosts, and tries to optimize placement of workloads.

For example, if you wanted to schedule a Redis container requiring 1 GB of memory, here is how you would schedule it with Swarm:

```
$ docker run -d -P -m 1g redis
```

Apart from resource scheduling, Swarm also supports policy-based scheduling with standard and custom constraints. For instance, if you want to run your **MySQL** container on an SSD-backed host (in order to ensure better write and read performance), you can specify that as follows:

```
$ docker run -d -P -e constraint:storage=ssd mysql
```

In addition to all of this, Swarm provides high-availability and failover. It continuously monitors the health of the containers, and if one were to suffer an outage, automatically rebalances by moving and restarting the Docker containers from the failed host to a new one. The best part is that regardless of whether you are just starting with one instance or have scaled up to 100 instances, the interface remains the same.

Like Docker Machine, Docker Swarm is in Alpha and is continuously evolving. Head over to its repository on Github to know more about it: `https://github.com/docker/swarm/`.

# Docker Compose

**Compose** is the last piece of the puzzle. With Docker Machine, we have provisioned the Docker daemons. With Docker Swarm, we can rest assured that we'll be able to control our containers from anywhere and that they'll remain available if there are any failures. Compose helps us compose our distributed applications on top of this cluster.

Comparing this to something we already know might help us understand how all of this works together. Docker Machine acts just as an operating system acts with respect to a program. It provides a place for containers to run. Docker Swarm acts like a programming language runtime to a program. It manages resources, provides exception handling, and so on for containers.

Docker Compose is more like an IDE, or a language syntax, that provides a way to express what the program needs to do. With Compose, we specify how our distributed apps must run in the cluster.

We use Docker Compose by writing a `YAML` file to declare the configurations and states of our multi-container app. For example, let's assume we have a Python app that uses a Redis DB. Here is how we would write the `YAML` file for Compose:

```
containers:
  web:
     build: .
     command: python app.py
     ports:
     - "5000:5000"
     volumes:
     - .:/code
     links:
     - redis
     environment:
     - PYTHONUNBUFFERED=1
  redis:
     image: redis:latest
     command: redis-server --appendonly yes
```

In the preceding example, we defined two applications. One is a Python application that needs to be built from the Dockerfile in the current directory. It has a port (`5000`) exposed and has either a volume or a piece of code bind mounted to the current working directory. It also has an environment variable defined and is linked to the second application container, `redis`. The second container uses the `redis` container from the Docker registry.

With the configuration defined, we can start both the containers with the following command:

```
$ docker up
```

With this single command, the Python container gets built using the Dockerfile, and the `redis` image gets pulled from the registry. However, the `redis` container is started first, because of the links directive in the Python container's specification and because the Python container depends on it.

As with Docker Machine and Docker Swarm, Docker Compose is a "work in progress" and its development can be tracked at `https://github.com/docker/docker/issues/9459`.

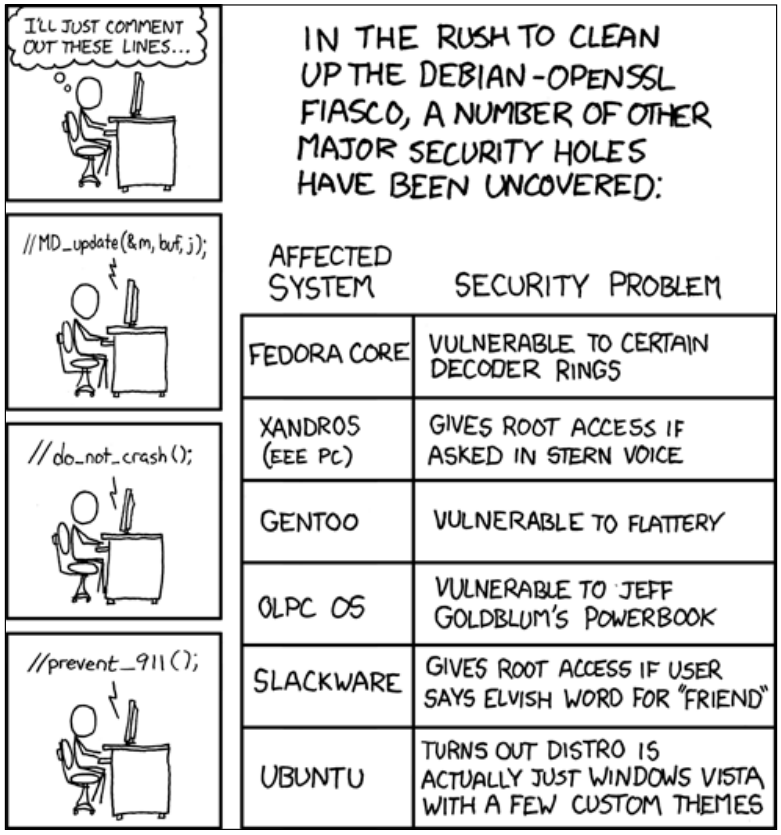
More information about swarm can be found at `http://blog.docker.com/2014/12/announcing-docker-machine-swarm-and-compose-for-orchestrating-distributed-apps/`.

# Security

Security is of prime importance when it comes to deciding whether to invest in a technology, especially when that technology has implications on the infrastructure and workflow. Docker containers are mostly secure, and since Docker doesn't interfere with other systems, you can use additional security measures to harden the security around the `docker` daemon. It is better to run the `docker` daemon in a dedicated host and run other services as containers (except services such as `ssh`, `cron`, and so on).

In this section, we will discuss Kernel features used in Docker that are pertinent to security. We will also consider the `docker` daemon itself as a possible attack vector.

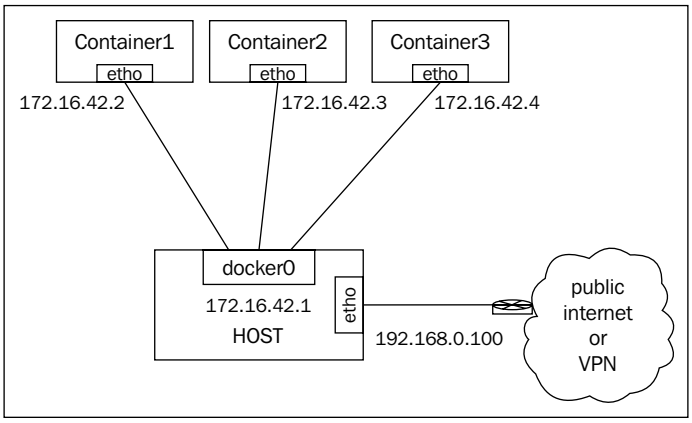
Image credit `http://xkcd.com/424/`



# Kernel namespaces

Namespaces provide sandboxing to containers. When a container is started, Docker creates a set of namespaces and cgroups for the container. Thus, a container that belongs to a particular namespace cannot see or affect the behavior of another container that belongs to other namespaces or the host.

The following diagram explains containers in Docker:




The kernel namespace also creates a network stack for the container, which can be configured to the last detail. The default Docker network setup resembles a simple network, with the host acting as the router and the `docker0` bridge acting as an Ethernet switch.

The namespace feature is modeled after OpenVZ, which is an operating system level virtualization technology based on the Linux kernel and operating system. OpenVZ is what is used in most of the cheap VPSes available in market today. It has been around since 2005, and the namespace feature was added to the kernel in 2008. It has been subjected to production use since then, so it can be called "battle hardened."

# Control groups

Control groups provide resource management features. Although this has nothing to do with privileges, it is relevant to security because of its potential to act as the first line of defence against denial-of-service attacks. Control groups have been around for quite some time as well, so can be considered safe for production use.

For further reading for control groups, refer to `https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt`.

# The root in a container

The `root` command in a container is stripped of many privileges. For instance, you cannot mount a device using the `mount` command by default. On the other end of the spectrum, running a container with the `--privileged flag` flag will give the `root` user in the container complete access to all the privileges that the root user in the host does. How does docker achieve this?

You can think of the standard `root` user as someone having a wide range of capabilities. One of them, is the `net_bind_service` service that binds to any port (even below 1024). Another, the `cap_sys_admin` service, is what is needed to mount physical drives. These are called capabilities, tokens used by a process to prove that it is allowed to perform an operation.

Docker containers are started with a reduced capability set. Hence, you will find that you can perform some root operations but not others. Specifically, it is not possible for a `root` user in an unprivileged container to do the following:

- Mount/unmount devices
- Managing raw sockets
- Filesystem operations such as creating device nodes and changing file ownerships

Before v1.2, if you needed to use any capability that was blacklisted, the only solution was to run the container with the `--privileged` flag. But v1.2 introduced three new flags, `--cap-add`, `--cap-drop`, and `--device`, to aid us to run a container that needed specific capabilities without compromising on the security of the host.

The `--cap-add` flag adds a capability to the container. For example, let's change the status of a container's interface (which requires the `NET_ADMIN` service capability):

```
$ docker run --cap-add=NET_ADMIN ubuntu sh -c "ip link eth0 down"
```

The `--cap-drop` flag blacklists a capability in a container. For example, let's blacklist all but the `chown` command in a container, and then try to add a user. This will fail as it needs the `CAP_CHOWN` service:

```
$ docker run --cap-add=ALL --cap-drop=CHOWN -it ubuntu useradd test
```

```
useradd: failure while writing changes to /etc/shadow
```

The `--devices` flag is used to mount an external/virtual device directly on the container. Before v1.2, we had to mount it on the host and bind mount with the `-v` flag in a `--privileged` container. With the `--device` flag, you can now use a device in a container without needing to use the `--privileged` container.

For example, to mount the DVD-RW device of your laptop on the container, run this command:

```
$ docker run --device=/dev/dvd-rw:/dev/dvd-rw ...
```

More information about the flags can be found at `http://blog.docker.com/tag/docker-1-2/`.

There were additional improvements introduced with the Docker 1.3 release. A `--security-opts` flag was added to the CLI, which allows you to set custom **SELinux** and **AppArmor** labels and profiles. For example, suppose you had a policy that allowed a container process to listen only to Apache ports. Assuming you had defined this policy in `svirt_apache`, you can apply it to the container as follows:

```
$ docker run --security-opt label:type:svirt_apache -i -t centos \
bash
```

One of benefits of this feature is that users will be able to run Docker in Docker without having to use the `docker run --privileged` container on the kernels supporting SELinux or AppArmor. Not giving the running container all the host access rights as the `--privileged` container significantly reduces the surface area of potential threats.

Source: `http://blog.docker.com/2014/10/docker-1-3-signed-images-process-injection-security-options-mac-shared-directories/`.

You can see the complete list of enabled capabilities at `https://github.com/docker/docker/blob/master/daemon/execdriver/native/template/default_template.go`.

For the inquisitive mind, the complete list of all available capabilities can be found in the Linux manual page for capabilities. It can also be found online at `http://man7.org/linux/man-pages/man7/capabilities.7.html`.

# Docker daemon attack surface

The `docker` daemon takes care of creating and managing containers, which includes creating filesystems, assigning IP addresses, routing packets, managing processes, and many more tasks that require root privileges. So it is imperative to start the daemon as a `sudo` user. This is the reason the `docker` daemon binds itself to a Unix socket by default, instead of a TCP socket, which it used until v5.2.

One of the end goals of Docker is to be able to run even the daemon as a non-root user, without affecting its functionalities, and delegate operations that do require root (such as filesystem operations and networking) to a dedicated subprocess with elevated privileges.

If you do want to expose Docker's port to the outside world (to make use of the remote API), it is advised to ensure that only trusted clients are allowed access. One straightforward way is to secure Docker with SSL. You can find ways of setting this up at `https://docs.docker.com/articles/https`.

# Best practices for security

Now let's summarize some key security best practices when running Docker in your infrastructure:

- Always run the `docker` daemon in a dedicated server.
- Unless you have a multiple-instance setup, run the `docker` daemon on a Unix socket.
- Take special care about bind mounting host directories as volumes as it is possible for a container to gain complete read-write access and perform irreversible operations in these directories.
- If you have to bind to a TCP port, secure it with SSL-based authentication.
- Avoid running processes with root privileges in your containers.
- There is absolutely no sane reason why you will ever need to run a privileged container in production.
- Consider enabling AppArmor/SELinux profiles in the host. This enables you to add an additional layer of security to the host.
- Unlike virtual machines, all containers share the host's kernel. So it is important to keep the kernel updated with the latest security patches.

# Summary

In this chapter, we learned about the various tools, APIs, and practices that help us deploy our application in a Docker-based environment. Initially, we looked at the Remote API and realized that all Docker commands are nothing but a result of REST-based calls to the `docker` daemon.

Then we saw how to inject processes to help debug running containers.

We then looked at various methods to achieve service discovery, both using native Docker features such as links, and with the help of specialized `config` stores such as the `etcd` services.

Finally, we discussed various aspects of security when using Docker, the various kernel features it relies on, their reliability, and their implications on the security of the host the containers run on.

In the next chapter, we will be taking the approach of this chapter further, and checking out various open source projects. We will learn how to integrate or use them to fully realize the potential of Docker.

# 5
# Friends of Docker

Up until now, we have been busy learning all about Docker. One major factor influencing the lifetime of open source projects is the community around it. The creators of Docker, Docker Inc. (the offshoot of **dotCloud)**, take care of developing and maintaining Docker and its sister projects such as libcontainer, libchan, swarm, and so on (the complete list can be found at `github.com/docker`). However, like any other open source project, the development is open (in GitHub), and they accept pull requests.

The industry has embraced Docker as well. Bigwigs such as Google, Amazon, Microsoft, eBay, and RedHat actively use and contribute to Docker. Most popular IaaS solutions such as Amazon Web Services, Google Compute Cloud, and so on support creating images preloaded with and optimized for Docker. Many start-ups are betting their fortunes on Docker as well. CoreOS, Drone.io, and Shippable are some of the start-ups that are modeled such that they provide services based around Docker. So you can rest assured that it's not going away any time soon.

In this chapter, we will discuss some of the projects surrounding Docker and how to use them. We will also be looking at projects you may already be familiar with that can facilitate your Docker workflow (and make your life a lot easier).

Firstly, we will talk about using Chef and Puppet recipes with Docker. Many of you might already be using these tools in your workflow. This section will help you integrate Docker with your current workflow, and ease you into the Docker ecosystem.

Next, we will try to set up an **apt-cacher** so that our Docker builds won't spend a lot of time fetching frequently used packages all the way from Canonical server. This will considerably reduce the time it takes to build images from Dockerfiles.

One of the things that gave Docker so much hype in the early stages was how easy some things that have been known to be hard seemed so easy when implemented with Docker. One such project is **Dokku**, a 100-line bash script that sets up a **mini-Heroku** like PaaS. We will set up our own PaaS using Dokku in this chapter. The very last thing we will be covering in this book is deploying a highly available service using CoreOS and Fleet.

In short, in this final leg of our journey, we will be looking at the following topics:

- Using Docker with Chef and Puppet
- Setting up an apt-cacher
- Setting up your own mini-Heroku
- Setting up a highly available service

# Using Docker with Chef and Puppet

When businesses started moving into the cloud, scaling became a whole lot easier as one could go from a single machine to hundreds without breaking a sweat. But this also meant configuring and maintaining these machines. Configuration management tools such as Chef and Puppet arose from the need to automate deploying applications in public/private clouds. Today, Chef and Puppet are used every day by start-ups and corporates all over the world to manage their cloud environments.

## Using Docker with Chef

Chef's website states the following:

> *"Chef turns infrastructure into code. With Chef, you can automate how you build, deploy, and manage your infrastructure. Your infrastructure becomes as versionable, testable, and repeatable as application code."*

Now, assuming that you have already set up Chef and are familiar with the Chef workflow, let's see how to use Docker with Chef using the chef-docker cookbook.

You can install this cookbook with any of the cookbook dependency managers. The installation instructions for each of Berkshelf, Librarian, and Knife are available at the Chef community site for the cookbook (`https://supermarket.getchef.com/cookbooks/docker`).

## Installing and configuring Docker

Installing Docker is simple. Just add the `recipe[docker]` command to your run-list (the list of configuration settings). An example is worth a million words, so let's see how to write a Chef recipe to run the `code.it` file (our sample project) on Docker.

## Writing a Chef recipe to run Code.it on Docker

The following Chef recipe starts a container based on `code.it`:

```
# Include Docker recipe
include_recipe 'docker'

# Pull latest image
docker_image 'shrikrishna/code.it'

# Run container exposing ports
docker_container 'shrikrishna/code.it' do
  detach true
  port '80:8000'
  env 'NODE_PORT=8000'
  volume '/var/log/code.it:/var/log/code.it'
end
```

The first non-comment statement includes the Chef-Docker recipe. The `docker_image 'shrikrishna/code.it'` statement is equivalent to running the `$ docker pull shrikrishna/code.it` command in the console. The block of statements at the end of the recipe is equivalent to running the `$ docker run --d -p '8000:8000' -e 'NODE_PORT=8000' -v '/var/log/code.it:/var/log/code.it' shrikrishna/code.it` command.

# Using Docker with Puppet

PuppetLabs's website states the following:

> *"Puppet is a configuration management system that allows you to define the state of your IT infrastructure, then automatically enforces the correct state. Whether you're managing just a few servers or thousands of physical and virtual machines, Puppet automates tasks that sysadmins often do manually, freeing up time and mental space so sysadmins can work on the projects that deliver greater business value."*

Puppet's equivalent of Chef cookbooks are modules. There is a well-supported module available for Docker. Its installation is carried out by running this command:

```
$ puppet module install garethr-docker
```

## Writing a Puppet manifest to run Code.it on Docker

The following Puppet manifest starts a `code.it` container:

```
# Installation
include 'docker'

# Download image
docker::image {'shrikrishna/code.it':}

# Run a container
docker::run { 'code.it-puppet':
  image   => 'shrikrishna/code.it',
  command => 'node /srv/app.js',
  ports   => '8000',
  volumes => '/var/log/code.it'
}
```

The first non-comment statement includes the `docker` module. The `docker::image {'shrikrishna/code.it':}` statement is equivalent to running the `$ docker pull shrikrishna/code.it` command in the console. The block of statements at the end of the recipe is equivalent to running the `$ docker run --d -p '8000:8000' -e 'NODE_PORT=8000' -v '/var/log/code.it:/var/log/code.it' shrikrishna/code.it node /srv/app.js` command.

# Setting up an apt-cacher

When you have multiple Docker servers, or when you are building multiple unrelated Docker images, you might find that you have to download packages every time. This can be prevented by having a caching proxy in-between the servers and clients. It caches packages as you install them. If you attempt to install a package that is already cached, it is served from the proxy server itself, thus reducing the latency in fetching packages and greatly speeding up the build process.

Let's write a Dockerfile that sets up an apt-caching server as a caching proxy server:

```
FROM        ubuntu

VOLUME      ["/var/cache/apt-cacher-ng"]
RUN       apt-get update ; apt-get install -yq apt-cacher-ng
```

```
EXPOSE      3142
RUN     echo "chmod 777 /var/cache/apt-cacher-ng ;" +
"/etc/init.d/apt-cacher-ng start ;" +
"tail -f /var/log/apt-cacher-ng/*" >> /init.sh
CMD     ["/bin/bash", "/init.sh"]
```

This Dockerfile installs the `apt-cacher-ng` package in the image and exposes port `3142` (for the target containers to use).

Build the image using this command:

```
$ sudo docker build -t shrikrishna/apt_cacher_ng
```

Then run it, binding the exposed port:

```
$ sudo docker run -d -p 3142:3142 --name apt_cacher
shrikrishna/apt_cacher_ng
```

To see the logs, run the following command:

```
$ sudo docker logs -f apt_cacher
```

# Using the apt-cacher while building your Dockerfiles

So we have set up an apt-cacher. We now have to use it in our Dockerfiles:

```
FROM ubuntu
RUN  echo 'Acquire::http { Proxy "http://<host's-docker0-ip-
here>:3142"; };' >> /etc/apt/apt.conf.d/01proxy
```

In the second instruction, replace the `<host's-docker0-ip-here>` command with your Docker host's IP address (at the `docker0` interface). While building this Dockerfile, if it encounters any `apt-get install` installation command for a package that has already been installed before (either for this image or for any other image), instead of using Docker's or Canonical package repositories, it will fetch the packages from the local proxy server, thus speeding up package installations in the build process. If the package being installed is not present in the cache, then it is fetched from Canonical repositories and saved in the cache.

An apt-cacher will only work for Debian-based containers (such as Ubuntu) that use the Apt package management tool.

# Setting up your own mini-Heroku

Now let's do something cool. For the uninitiated, Heroku is a cloud PaaS, which means that all you need to do upon building an application is to push it to Heroku and it will get deployed on `https://www.herokuapp.com`. You don't need to worry how or where your application runs. As long as the PaaS supports your technology stack, you can just develop locally and push the application to the service to have it running live on the public Internet.

There are a lot of PaaS providers apart from Heroku. Some popular providers are Google App Engine, Red Hat Cloud, and Cloud Foundry. Docker was developed by one such PaaS provider—dotCloud. Almost every PaaS works by running the applications in predefined sandboxed environments, and this is something Docker excels at. Today, Docker has made setting up a PaaS easier, if not simple. The project that proved this was Dokku. Dokku shares the usage pattern and terminologies (such as `buildpacks`, `slug builder` scripts) with Heroku, which makes it easier to use. In this section, we will be setting up a mini-PaaS using Dokku and pushing our `code.it` application.

The next steps should be done on either a **Virtual Private Server** (**VPS**) or a virtual machine. The host you are working from should have git and SSH set up.

# Installing Dokku using a bootstrapper script

There is a `bootstrapper` script that will set up Dokku. Run this command inside the VPS/virtual machine:

```
$ wget -qO- https://raw.github.com/progrium/dokku/v0.2.3/bootstrap.sh
| sudo DOKKU_TAG=v0.2.3 bash
```

Users on version 12.04 will need to run the `$ apt-get install -y python-software-properties` command before running the preceding `bootstrapper` script.

The `bootstrapper` script will download all the dependencies and set up Dokku.

# Installing Dokku using Vagrant

Step 1: Clone Dokku:

```
$ git clone https://github.com/progrium/dokku.git
```

Step 2: Set up SSH hosts in your `/etc/hosts` file:

```
10.0.0.2 dokku.app
```

Step 3: Set up SSH Config in `~/.ssh/config`

```
Host dokku.app
    Port 2222
```

Step 4: Create a VM

Here are some optional ENV arguments to set up:

```
# - `BOX_NAME`
# - `BOX_URI`
# - `BOX_MEMORY`
# - `DOKKU_DOMAIN`
# - `DOKKU_IP`.
cd path/to/dokku
vagrant up
```

Step 5 : Copy your SSH key using this command:

```
$ cat ~/.ssh/id_rsa.pub | pbcopy
```

Paste your SSH key in the dokku-installer at `http://dokku.app` (which points to `10.0.0.2` as assigned in the `/etc/hosts` file). Change the **Hostname** field on the **Dokku Setup** screen to your domain and then check the box that says **Use virtualhost naming**. Then, click on **Finish Setup** to install your key. You'll be directed to application deployment instructions from here.

You are now ready to deploy an app or install plugins.

# Configuring a hostname and adding the public key

Our PaaS will be routing subdomains to applications deployed with the same name. This means that the machine where Dokku has been set up must be visible to your local setup as well as to the machine where Dokku runs.

Set up a wildcard domain that points to the Dokku host. After running the `bootstrapper` script, check whether the `/home/dokku/VHOST` file in the Dokku host is set to this domain. It will only be created if the hostname can be resolved by the dig tool.

In this example, I have set my Dokku hostname to `dokku.app` by adding the following configuration to my `/etc/hosts` file (of the local host):

```
10.0.0.2 dokku.app
```

I have also set up an SSH port forwarding rule in the `~/.ssh/config` file (of the local host):

```
Host dokku.app
    Port 2222
```

According to Wikipedia, **Domain Information Groper** (**dig**) is a network administration command-line tool used to query DNS name servers. This means that given a URL, dig will return the IP address of the server that the URL points to.

If the `/home/dokku/VHOST` file hasn't been automatically created, you will have to manually create it and set it to your preferred domain name. If this file is missing when you deploy your application, Dokku will publish the application with a port name instead of the subdomain.

The last thing to do is to upload your public `ssh` key to the Dokku host and associate it with a username. To do so, run this command:

```
$ cat ~/.ssh/id_rsa.pub | ssh dokku.app "sudo sshcommand acl-add
dokku shrikrishna"
```

In the preceding command, replace the `dokku.app` name with your domain name and `shrikrishna` with your name.

Great! Now that we're up and ready, it's time to deploy our application.

# Deploying an application

We now have a PaaS of our own where we can deploy our applications. Let's deploy the `code.it` file there. You can also try deploying your own application there:

```
$ cd code.it
$ git remote add dokku dokku@dokku.app:codeit
$ git push dokku master
```

```
Counting objects: 456, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (254/254), done.
Writing objects: 100% (456/456), 205.64 KiB, done.
Total 456 (delta 34), reused 454 (delta 12)
-----> Building codeit ...
       Node.js app detected
-----> Resolving engine versions

......
......
......

-----> Application deployed:
       http://codeit.dokku.app
```

That's it! We now have a working application in our PaaS. For more details about Dokku, you can check out its GitHub repository page at `https://github.com/progrium/dokku`.

If you want a production-ready PaaS, you must look up Deis at `http://deis.io/`, which provides multi-host and multi-tenancy support.

# Setting up a highly available service

While Dokku is great to deploy occasional side projects, it may not be suitable for larger projects. A large-scale deployment essentially has the following requirements:

- **Horizontally scalable**: There is only so much that can be done with a single instance of a server. As the load increases, an organization on the hockey stick growth curve will find itself having to balance the load among a cluster of servers. In the earlier days, this meant having to design data centers. Today, this means adding more instances to the cloud.
- **Fault tolerant**: Just as road accidents occur even when there are extensive traffic rules in place to avoid them, crashes might occur even after you take extensive measures to prevent them, but a crash in one of the instances must not create service downtime. A well-designed architecture will handle failure conditions and will make another server available to take the place of the server that crashed.

- **Modular**: While this may not seem so, modularity is a defining feature of a large-scale deployment. A modular architecture makes it flexible and future-proof (because a modular architecture will accommodate newer components as the scope and the reach of the organization grow).

This is by no means an exhaustive list, but it marks the amount of effort it takes to build and deploy a highly available service. However, as we have seen until now, Docker is used in a single host, and there are no tools available in it (until now) to manage a cluster of instances running Docker.

This is where CoreOS comes in. It is a minimal operating system built with the single intention of being the building block in large-scale deployments of services on Docker. It comes with a highly available key-value config store called `etcd`, which is used for configuration management and service discovery (discovering where each of the other components is located in the cluster). The `etcd` service was explored in *Chapter 4*, *Automation and Best Practices*. It also comes with fleet, a tool that leverages `etcd` to provide a way to perform actions on the entire cluster as opposed to doing so on individual instances.

You can think of fleet as an extension of the `systemd` suite that operates at the cluster level instead of the machine level. The `systemd` suite is a single-machine init system whereas fleet is a cluster init system. You can find out more about fleet at `https://coreos.com/using-coreos/clustering/`.

In this section, we will try to deploy our standard example, `code.it`, on a three-node CoreOS cluster in our local host. This is a representative example and an actual multi-host deployment will take a lot more work, but this serves as a good starting point. It also helps us appreciate the great work that has been done over the years, both in terms of hardware and software, to make it possible, even easy, to deploy a high-availability service, a task that had until only a few years ago been only possible in huge data centers.

# Installing dependencies

Running the preceding example requires the following dependencies:

1. **VirtualBox**: VirtualBox is a popular type of virtual machine management software. Installation executables for your platform can be downloaded from `https://www.virtualbox.org/wiki/Downloads`.
2. **Vagrant**: Vagrant is an open source tool that can be considered a virtual machine equivalent for Docker. It can be downloaded from `https://www.vagrantup.com/downloads.html`.

3. **Fleetctl**: Fleet is, in short, a distributed init system, which means that it will allow us to manage services in a cluster level. Fleetctl is a CLI client to interface to run the fleet commands. To install fleetctl, run the following commands:

```
$ wget \ https://github.com/coreos/fleet/releases/download/v0.3.2/
fleet -v0.3.2-darwin-amd64.zip && unzip fleet-v0.3.2-darwin-amd64.
zip

$ sudo cp fleet-v0.3.2-darwin-amd64/fleetctl /usr/local/bin/
```

# Getting and configuring the Vagrantfile

Vagrantfiles are the Vagrant equivalent of Dockerfiles. A Vagrantfile contains details such as the base virtual machine to get, the setup commands to run, the number of instances of the virtual machine image to start, and so on. CoreOS has a repository that contains the Vagrantfile that can be used to download and use CoreOS within virtual machines. This is the ideal way to try out CoreOS's features in a development environment:

```
$ git clone https://github.com/coreos/coreos-vagrant/
$ cd coreos-vagrant
```

The preceding command clones the `coreos-vagrant` repository, which contains the Vagrantfile that downloads and starts CoreOS-based virtual machines.

Vagrant is a piece of free and open source software used to create and configure virtual development environments. It can be seen as a wrapper around virtualization software such as VirtualBox, KVM, or VMware, and around configuration management software such as Chef, Salt, or Puppet. You can download Vagrant from `https://www.vagrantup.com/downloads.html`.

Before starting the virtual machines though, we have some configuring to do.

## Getting discovery tokens

Each CoreOS host runs an instance of the `etcd` service to coordinate the services running in that machine and to communicate with services running in other machines in the cluster. For this to happen, the `etcd` instances themselves need to discover each other.

A discovery service (`https://discovery.etcd.io`) has been built by the CoreOS team, which provides a free service to help the `etcd` instances communicate with each other by storing peer information. It works by providing a unique token that identifies the cluster. Each `etcd` instance in the cluster identifies every other `etcd` instance with this token using the discovery service. Generating a token is easy and is done by sending a `GET` request to `discovery.etcd.io/new`:

```
$ curl -s https://discovery.etcd.io/new
https://discovery.etcd.io/5cfcf52e78c320d26dcc7ca3643044ee
```

Now open the file named `user-data.sample` in the `coreos-vagrant` directory and find the commented-out line that holds the `discovery` configuration option under the `etcd` service. Uncomment it and provide the token that is returned from the previously run `curl` command. Once this is done, rename the file to `user-data`.

The `user-data` file is used to set configuration parameters for the `cloud-config` program in CoreOS instances. Cloud-config is inspired by the `cloud-config` file from the `cloud-init` project, which defines itself as the DE-facto multi-distribution package that handles early initialization of a cloud instance (`cloud-init` docs). In short, it helps configure the various parameters such as ports to be opened, and in the case of CoreOS, the `etcd` configurations, and so on. You can find out more at:

`https://coreos.com/docs/cluster-management/setup/cloudinit-cloud-config/` and `http://cloudinit.readthedocs.org/en/latest/index.html`.

The following is an example of the code of CoreOS:

```
coreos:
  etcd:
    # generate a new token for each unique cluster from https://
discovery.etcd.io/new
    # WARNING: replace each time you 'vagrant destroy'
    discovery: https://discovery.etcd.io/5cfcf52e78c320d26dcc7ca36430
44ee
    addr: $public_ipv4:4001
    peer-addr: $public_ipv4:7001
  fleet:
    public-ip: $public_ipv4
  units:
```

You will have to generate a new token each time you run the cluster. Simply reusing the token will not work.

## Setting the number of instances

In the `coreos-vagrant` directory, there is another file called `config.rb.sample`. Find the commented line in this file that reads `$num_instances=1`. Uncomment it and set the value to `3`. This will make Vagrant spawn three instances of CoreOS. Now save the file as `config.rb`.

The `cnfig.rb` file holds the configurations for the Vagrant environment and the number of machines in the cluster.

The following is the code example for Vagrant instances:

```
# Size of the CoreOS cluster created by Vagrant
$num_instances=3
```

## Spawning instances and verifying health

Now that we have the configurations ready, it's time to see a cluster running in your local machine:

```
$ vagrant up
Bringing machine 'core-01' up with 'virtualbox' provider...
Bringing machine 'core-02' up with 'virtualbox' provider...
Bringing machine 'core-03' up with 'virtualbox' provider...
==> core-01: Box 'coreos-alpha' could not be found. Attempting to find
and install...
    core-01: Box Provider: virtualbox
    core-01: Box Version: >= 0
==> core-01: Adding box 'coreos-alpha' (v0) for provider: virtualbox
. . . . .
. . . . .
. . . . .
```

After the machines are created, you can SSH into them to try out the following commands, but you will need to add `ssh` keys to your SSH agent. Doing so will allow you to forward your SSH session to other nodes in the cluster. To add the keys, run the following command:

```
$ ssh-add ~/.vagrant.d/insecure_private_key
Identity added: /Users/CoreOS/.vagrant.d/insecure_private_key (/Users/
CoreOS/.vagrant.d/insecure_private_key)
$ vagrant ssh core-01 -- -A
```

Now let's verify that the machines are up and ask fleet to list the machines running in the cluster:

```
$ export FLEETCTL_TUNNEL=127.0.0.1:2222
$ fleetctl list-machines
MACHINE     IP            METADATA
daacff1d... 172.17.8.101 -
20dddafc... 172.17.8.102 -
eac3271e... 172.17.8.103 -
```

# Starting the service

To run a service in your newly started cluster, you will have to write the `unit-files` files. Unit files are configuration files that list the services that must be run in each machine and some rules on how to manage these services.

Create three files named `code.it.1.service`, `code.it.2.service`, and `code.it.3.service`. Populate them with the following configurations:

`code.it.1.service`

```
[Unit]
Description=Code.it 1
Requires=docker.service
After=docker.service

[Service]
ExecStart=/usr/bin/docker run --rm --name=code.it-1 -p 80:8000
shrikrishna/code.it
ExecStartPost=/usr/bin/etcdctl set /domains/code.it-1/%H:%i
running
ExecStop=/usr/bin/docker stop code.it-1
ExecStopPost=/usr/bin/etcdctl rm /domains/code.it-1/%H:%i
```

```
    [X-Fleet]
    X-Conflicts=code.it.*.service
```

code.it.2.service

```
    [Unit]
    Description=Code.it 2
    Requires=docker.service
    After=docker.service

    [Service]
    ExecStart=/usr/bin/docker run --rm --name=code.it-2 -p 80:8000
    shrikrishna/code.it
    ExecStartPost=/usr/bin/etcdctl set /domains/code.it-2/%H:%i
    running
    ExecStop=/usr/bin/docker stop code.it-2
    ExecStopPost=/usr/bin/etcdctl rm /domains/code.it-2/%H:%i

    [X-Fleet]
    X-Conflicts=code.it.2.service
```

code.it.3.service

```
    [Unit]
    Description=Code.it 3
    Requires=docker.service
    After=docker.service

    [Service]
    ExecStart=/usr/bin/docker run --rm --name=code.it-3 -p 80:8000
    shrikrishna/code.it
    ExecStartPost=/usr/bin/etcdctl set /domains/code.it-3/%H:%i
    running
    ExecStop=/usr/bin/docker stop code.it-3
    ExecStopPost=/usr/bin/etcdctl rm /domains/code.it-3/%H:%i

    [X-Fleet]
    X-Conflicts=code.it.*.service
```

You might have noticed a pattern in these files. The `ExecStart` parameter holds the command that must be executed in order to start the service. In our case, this means running the `code.it` container. `ExecStartPost` is the command that is executed once the `ExecStart` parameter succeeds. In our case, the service's availability is registered in the `etcd` service. Conversely, the `ExecStop` command will stop the service, and the `ExecStopPost` command executes once the `ExecStop` command succeeds, which in this case means removing the service's availability from the `etcd` service.

`X-Fleet` is a CoreOS-specific syntax that tells fleet that two services cannot run on the same machine (as they would conflict while trying to bind to the same port). Now that all the blocks are in place, it's time to submit the jobs to the cluster:

```
$ fleetctl submit code.it.1.service code.it.2.service
code.it.3.service
```

Let's verify that the services have been submitted to the cluster:

```
$ fleetctl list-units
UNIT              LOAD  ACTIVE  SUB  DESC                MACHINE
code.it.1.service  -     -       -   Code.it 1  -
code.it.2.service  -     -       -   Code.it 2  -
code.it.3.service  -     -       -   Code.it 3  -
```

The machine column is empty and the active status is not set. This means our services haven't started yet. Let's start them:

```
$ fleetctl start code.it.{1,2,3}.service
Job code.it.1.service scheduled to daacff1d.../172.17.8.101
Job code.it.1.service scheduled to 20dddafc.../172.17.8.102
Job code.it.1.service scheduled to eac3271e.../172.17.8.103
```

Let's verify that they are running by executing the `$ fleetctl list-units` file again:

```
$ fleetctl list-units
UNIT               LOAD    ACTIVE   SUB     DESC
MACHINE
code.it.1.service  loaded  active  running  Code.it 1
daacff1d.../172.17.8.101
code.it.1.service  loaded  active  running  Code.it 2
20dddafc.../172.17.8.102
code.it.1.service  loaded  active  running  Code.it 3
eac3271e.../172.17.8.103
```

Congratulations! You have just set up your very own cluster! Now head over to `172.17.8.101`, `172.17.8.102`, or `172.17.8.103` in a web browser and see the `code.it` application running!

We have only set up a cluster of machines running a highly available service in this example. If we add a load balancer that maintains a connection with the `etcd` service to route requests to available machines, we will have a complete end-to-end production level service running in our systems. But doing so would veer off the topic, so is left as an exercise for you.

With this, we come to the end. Docker is still under active development, and so are the projects like CoreOS, Deis, Flynn, and so on. So, although we have seen great stuff coming out over the past few months, what is coming is going to be even better. We are living in exciting times. So, let's make the best of it and build stuff that makes this world a better place to live in. Happy shipping!

# Summary

In this chapter, we learned how to use Docker with Chef and Puppet. Then we set up an apt-cacher to speed up package downloads. Next, we set up our own mini PaaS with Dokku. In the end, we set up a high-availability service using CoreOS and Fleet. Congratulations! Together, we have gained the necessary knowledge of Docker to build our containers, "dockerize" our applications and even run clusters. Our journey ends here. But for you, dear reader, a new journey has just begun. This book was meant to lay the groundwork to help you build the next big thing using Docker. I wish you all the success in the world. If you liked this book, give me a hoot at `@srikrishnaholla` on Twitter. If you didn't like it, let me know how I can make it better.

# Index

## Symbols



## A



## B



## C

## D

## R

## S



## T



## U



## V

## W


## Thank you for buying
# Orchestrating Docker

## About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

## About Packt Open Source

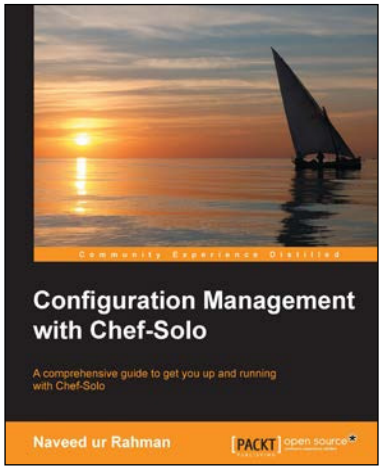
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

## Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING



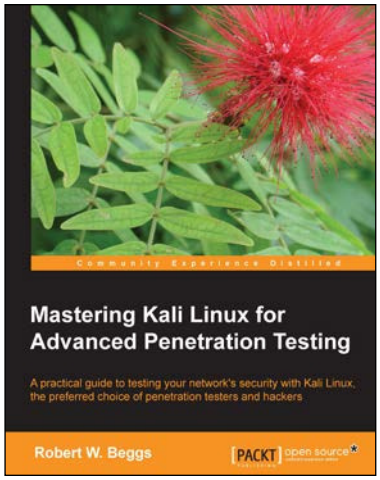

## Configuration Management with Chef-Solo

ISBN: 978-1-78398-246-2 Paperback: 116 pages

A comprehensive guide to get you up and running with Chef-Solo

1. Explore various techniques that will help you save time in infrastructure management.
2. Use the power of Chef-Solo to run your servers and configure and deploy applications in an automated manner.
3. This book will help you to understand the need for the configuration management tool and provides you with a step-by-step guide to maintain your existing infrastructure.

---




## Mastering Kali Linux for Advanced Penetration Testing

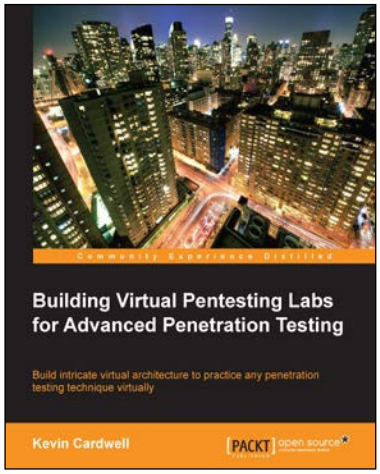
ISBN: 978-1-78216-312-1 Paperback: 356 pages

A practical guide to testing your network's security with Kali Linux, the preferred choice of penetration testers and hackers

1. Conduct realistic and effective security tests on your network.
2. Demonstrate how key data systems are stealthily exploited, and learn how to identify attacks against your own systems.
3. Use hands-on techniques to take advantage of Kali Linux, the open source framework of security tools.

Please check **www.PacktPub.com** for information on our titles

[PACKT] open source
community experience distilled
PUBLISHING



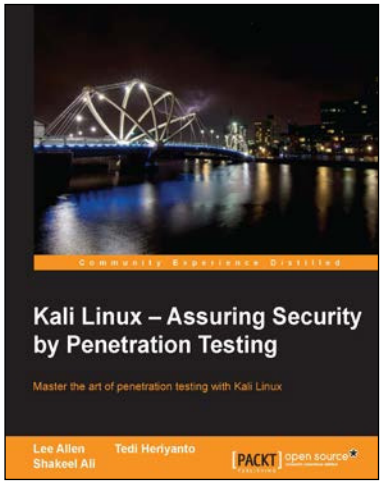

## Building Virtual Pentesting Labs for Advanced Penetration Testing

ISBN: 978-1-78328-477-1 Paperback: 430 pages

Build intricate virtual architecture to practice any penetration testing technique virtually

1. Build and enhance your existing pentesting methods and skills.
2. Get a solid methodology and approach to testing.
3. Step-by-step tutorial helping you build complex virtual architecture.

---




## Kali Linux – Assuring Security by Penetration Testing

ISBN: 978-1-84951-948-9 Paperback: 454 pages

Master the art of penetration testing with Kali Linux

1. Learn penetration testing techniques with an in-depth coverage of Kali Linux distribution.
2. Explore the insights and importance of testing your corporate network systems before the hackers strike.
3. Understand the practical spectrum of security tools by their exemplary usage, configuration, and benefits.

Please check **www.PacktPub.com** for information on our titles