



Community Experience Distilled

# Docker High Performance

Master performance enhancement practices for Docker, and unlock faster and more efficient container deployment that will improve your development workflow

Allan Espinosa

[PACKT] open source\*  
PUBLISHING community experience distilled

# Docker High Performance

Master performance enhancement practices for Docker, and unlock faster and more efficient container deployment that will improve your development workflow

**Allan Espinosa**



BIRMINGHAM - MUMBAI

# Docker High Performance

Copyright © 2016 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 2016

Production reference: 1220116

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78588-680-5

[www.packtpub.com](http://www.packtpub.com)

# Credits

**Author**

Allan Espinosa

**Project Coordinator**

Milton Dsouza

**Reviewer**

Shashikant Bangera

**Proofreader**

Safis Editing

**Acquisition Editor**

Aaron Lazar

**Indexer**

Monica Ajmera Mehta

**Content Development Editor**

Rashmi Suvarna

**Graphics**

Jason Monteiro

**Technical Editors**

Vijin Boricha

Humera Shaikh

**Production Coordinator**

Nilesh Mohite

**Copy Editor**

Shruti Iyer

**Cover Work**

Nilesh Mohite

# About the Author

**Allan Espinosa** is a DevOps practitioner living in Tokyo. He is an active open source contributor to various distributed systems tools, such as Docker and Chef. Allan maintains several Docker images for popular open source software that were popular even before their official release from the upstream open source groups.

In his career, Allan has worked on large distributed systems containing hundreds to thousands of servers in production. He has built scalable applications on various platforms ranging from large supercomputing centers in the U.S. to production enterprise systems in Japan.

Allan can be contacted through his Twitter handle @AllanEspinosa. His personal website at <http://aespinoza.github.io> contains several blog posts on Docker and distributed systems in general.

---

I would like to thank my wife, Kana, for the continuous support that allowed me to spend significant time writing this book.

---

# About the Reviewer

**Shashikant Bangera** is a DevOps architect with 16 years of experience in the IT sector. He has had a vast exposure to DevOps tools with core expertise in open source DevOps tools. Shashikant has worked on a large number of multimillion-pound projects. He helped in making the transition from employing traditional development practices to adopting agile tooling and processes that increase the release frequency and quality of software. Moreover, Shashikant has designed an automated on-demand environment with open source tools. He has hands-on experience with a variety of DevOps tools across the domain.

Shashikant has also reviewed the book *Learning Docker* by Packt Publishing.

# www.PacktPub.com

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

For support files and downloads related to your book, please visit [www.PacktPub.com](http://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](http://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](mailto:service@packtpub.com) for more details.

At [www.PacktPub.com](http://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](http://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

<b>Preface</b>	<b>v</b>
<b>Chapter 1: Preparing Docker Hosts</b>	<b>1</b>
<b>Preparing a Docker host</b>	<b>1</b>
<b>Working with Docker images</b>	<b>2</b>
Building Docker images	3
Pushing Docker images to a repository	4
Pulling Docker images from a repository	6
<b>Running Docker containers</b>	<b>7</b>
Exposing container ports	7
Publishing container ports	9
--publish-all	9
--publish	10
Linking containers	11
Interactive containers	12
<b>Summary</b>	<b>14</b>
<b>Chapter 2: Optimizing Docker Images</b>	<b>15</b>
<b>Reducing deployment time</b>	<b>16</b>
<b>Improving image build time</b>	<b>19</b>
Using registry mirrors	19
Reusing image layers	22
Reducing the build context size	28
Using caching proxies	30
<b>Reducing Docker image size</b>	<b>33</b>
Chaining commands	33
Separating build and deployment images	35
<b>Summary</b>	<b>39</b>



---

<b>Chapter 3: Automating Docker Deployments with Chef</b>	<b>41</b>
<b>An introduction to configuration management</b>	<b>41</b>
<b>Using Chef</b>	<b>43</b>
Signing up for a Chef server	44
Setting up our workstation	46
Bootstrap nodes	48
<b>Configuring the Docker host</b>	<b>50</b>
<b>Deploying Docker containers</b>	<b>54</b>
<b>Alternative methods</b>	<b>58</b>
<b>Summary</b>	<b>59</b>
<b>Chapter 4: Monitoring Docker Hosts and Containers</b>	<b>61</b>
<b>The importance of monitoring</b>	<b>62</b>
<b>Collecting metrics to Graphite</b>	<b>63</b>
Graphite in production	67
<b>Monitoring with collectd</b>	<b>68</b>
Collecting Docker-related data	71
Running collectd inside Docker	74
<b>Consolidating logs in an ELK stack</b>	<b>74</b>
<b>Forwarding Docker container logs</b>	<b>79</b>
<b>Other monitoring and logging solutions</b>	<b>81</b>
<b>Summary</b>	<b>82</b>
<b>Chapter 5: Benchmarking</b>	<b>83</b>
<b>Setting up Apache JMeter</b>	<b>84</b>
Deploying a sample application	84
Installing JMeter	87
<b>Building a benchmark workload</b>	<b>89</b>
Creating a test plan in JMeter	89
<b>Analyzing benchmark results</b>	<b>92</b>
Viewing the results of JMeter runs	92
Calculating throughput	92
Plotting response time	94
Observing performance in Graphite and Kibana	95
<b>Tuning the benchmark</b>	<b>99</b>
Increasing concurrency	99
Running distributed tests	100
<b>Other benchmarking tools</b>	<b>102</b>
<b>Summary</b>	<b>102</b>
<b>Chapter 6: Load Balancing</b>	<b>103</b>
<b>Preparing a Docker host farm</b>	<b>103</b>
<b>Balancing load with Nginx</b>	<b>105</b>

<b>Scaling out our Docker applications</b>	<b>108</b>
Deploying with zero downtime	110
<b>Other load balancers</b>	<b>114</b>
<b>Summary</b>	<b>114</b>
<b>Chapter 7: Troubleshooting Containers</b>	<b>115</b>
<b>Inspecting containers</b>	<b>115</b>
<b>Debugging from the outside</b>	<b>119</b>
Tracing system calls	119
Analyzing network packets	122
Observing block devices	124
<b>A stack of troubleshooting tools</b>	<b>127</b>
<b>Summary</b>	<b>128</b>
<b>Chapter 8: Onto Production</b>	<b>129</b>
<b>Performing web operations</b>	<b>129</b>
<b>Supporting web applications with Docker</b>	<b>131</b>
<b>Deploying applications</b>	<b>133</b>
<b>Scaling applications</b>	<b>134</b>
<b>Further reading</b>	<b>135</b>
<b>Summary</b>	<b>135</b>
<b>Index</b>	<b>137</b>

---



# Preface

Docker is a great tool to build and deploy our applications. Its portable container format allows us to run code anywhere, from our developer workstations to popular cloud computing providers. The workflow around Docker makes development, testing, and deployment easier and faster. However, this is very important to Docker's internals and continuously improving best practices to realize its full potential.

## What this book covers

Engineers that have a basic understanding of Docker can read the book sequentially, chapter by chapter. Tech leads who have an advanced understanding of Docker or have deployed applications in production before can go ahead and read *Chapter 8, Onto Production*, first to understand how Docker can fit in your existing applications. The following is a list of topics covered in this book:

*Chapter 1, Preparing Docker Hosts*, gives a quick refresher on setting up and running Docker. It documents the setup that you will be using throughout the book.

*Chapter 2, Optimizing Docker Images*, shows why it is important to tune your Docker images. A few tuning tips will be shown to improve the deployability and performance of our Docker containers.

*Chapter 3, Automating Docker Deployments with Chef*, shows how to automate the provisioning and setup of Docker hosts. It will discuss the importance of investing in automation and how it facilitates a scalable way of deploying your Docker containers.

*Chapter 4, Monitoring Docker Hosts and Containers*, gives a walk-through of setting up a monitoring system with Graphite and logging systems with an Elasticsearch-Logstash-Kibana (ELK) stack..

*Chapter 5, Benchmarking*, is a tutorial on how to use Apache JMeter to create workloads to benchmark the performance of your Docker containers. The chapter reviews the monitoring system you set up in *Chapter 4, Monitoring Docker Hosts and Containers*, to analyze some Docker application benchmark results, such as response time and throughput.

*Chapter 6, Load Balancing*, shows you how to configure and deploy an Nginx-based load balancer Docker container. The chapter also gives a tutorial on how to use the load balancer you set up to scale out the performance and deployability of our Docker applications.

*Chapter 7, Troubleshooting Containers*, illustrates how common debugging tools in a typical Linux system can be used to troubleshoot your Docker containers. They describe how each tool works and how it can read the diagnostics coming from your running Docker containers.

*Chapter 8, Onto Production*, synthesizes all the performance optimizations you did in the previous chapter and relates what it means to operate any web application in production with Docker.

## What you need for this book

A Linux workstation with a recent kernel is needed to serve as a host for Docker 1.10.0. This book uses Debian Jessie 8.2 as its base operating system to install and set up Docker.

More details on how to get Docker up and running is covered in *Chapter 1, Preparing Docker Hosts*.

## Who this book is for

This book is written for developers and operations people who want to deploy their Docker application and infrastructure to production. If you have learned the basics of Docker already but want to move forward to the next level, then this book is for you.

## Conventions

In this book, you will find a number of text styles 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 will use `--link <source>:<alias>` to create a link from the source container, named `source`, to an alias called `webapp`."

A block of code is set as follows:

```
FROM ubuntu:14.04
MAINTAINER Docker Education Team <education@docker.com>
RUN apt-get update
RUN DEBIAN_FRONTEND=noninteractive apt-get \
    install -y -q python-all python-pip
ADD ./webapp/requirements.txt /tmp/requirements.txt
RUN pip install -qr /tmp/requirements.txt
ADD ./webapp /opt/webapp/
WORKDIR /opt/webapp
EXPOSE 5000
CMD ["python", "app.py"]
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:


```
import os
from flask import Flask
app = Flask(__name__)
@app.route('/')
def hello():
    provider = str(os.environ.get('PROVIDER', 'world'))
    return 'Hello '+provider+'!'
if __name__ == '__main__':
    # Bind to PORT if defined, otherwise default to 5000.
    port = int(os.environ.get('PORT', 5000))
    app.run(host='0.0.0.0', port=port)
```


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

```
dockerhost$ docker inspect -f "{{ .NetworkSettings.IPAddress }}" \
    source
172.17.0.15
dockerhost$ docker inspect -f "{{ .NetworkSettings.IPAddress }}" \
    destination
172.17.0.28
dockerhost$ iptables -L DOCKER
Chain DOCKER (1 references)
```

target	prot	opt	source	destination	
ACCEPT	tcp	--	172.17.0.28	172.17.0.15	tcp dpt:5000
ACCEPT	tcp	--	172.17.0.15	172.17.0.28	tcp spt:5000

New **terms** and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Finally, click on **Download Starter Kit**."

[  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 disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail [feedback@packtpub.com](mailto:feedback@packtpub.com), and mention the book's title in the subject of your message.

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

## 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 could 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 to our website or added to any list of existing errata under the Errata section of that title.

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

## Piracy

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

Please contact us at [copyright@packtpub.com](mailto:copyright@packtpub.com) with a link to the suspected pirated material.

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

## Questions

If you have a problem with any aspect of this book, you can contact us at [questions@packtpub.com](mailto:questions@packtpub.com), and we will do our best to address the problem.





# 1

## Preparing Docker Hosts

Docker allows us to deliver applications to our customers faster. It simplifies the workflows needed to get code from development to production by enabling us to easily create and launch Docker containers. This chapter will be a quick refresher on how to get our environment ready to run a Docker-based development and operations workflow by:

- Preparing a Docker host
- Working with Docker images
- Running Docker containers

Most parts of this chapter are concepts that we are already familiar with and are readily available on the Docker documentation website. This chapter shows selected commands and interactions with the Docker host that will be used in the succeeding chapters.

### Preparing a Docker host

It is assumed that we are already familiar with how to set up a Docker host. For most of the chapters of this book, we will run our examples against the following environment, unless explicitly mentioned otherwise:

- Operating system – Debian 8.2 Jessie
- Docker version – 1.10.0


The following command displays the operating system and Docker version:

```
$ ssh dockerhost
dockerhost$ lsb_release -a
No LSB modules are available.
Distributor ID: Debian
Description:   Debian GNU/Linux 8.2 (jessie)
```

```
Release:      8.2
Codename:     jessie
dockerhost$ docker version
Client:
  Version:     1.10.0
  API version: 1.21
  Go version:  go1.4.2
  Git commit:  a34a1d5
  Built:       Fri Nov 20 12:59:02 UTC 2015
  OS/Arch:     linux/amd64
```

```
Server:
  Version:     1.10.0
  API version: 1.21
  Go version:  go1.4.2
  Git commit:  a34a1d5
  Built:       Fri Nov 20 12:59:02 UTC 2015
  OS/Arch:     linux/amd64
```

If we haven't set up our Docker environment yet, we can follow the instructions on the Docker website found at <https://docs.docker.com/installation/debian> to prepare our Docker host.



**Downloading the example code**

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

## Working with Docker images

Docker images are artifacts that contain our application and other supporting components to help run it, such as the base operating system, runtime and development libraries, and so on. They get deployed and downloaded into Docker hosts in order to run our applications as Docker containers. This section will cover the following Docker commands to work with Docker images:

- `docker build`
- `docker images`

- `docker push`
- `docker pull`



Most of the material in this section is readily available on the Docker documentation website at <https://docs.docker.com/userguide/dockerimages>.

## Building Docker images

We will use the `Dockerfile` of `training/webapp` from the Docker Education Team to build a Docker image. The next few steps will show us how to build this web application:

1. To begin, we will clone the Git repository of `webapp`, which is available at <https://github.com/docker-training/webapp> via the following command:

```
dockerhost$ git clone https://github.com/docker-training/webapp.
git training-webapp
Cloning into 'training-webapp'...
remote: Counting objects: 45, done.
remote: Total 45 (delta 0), reused 0 (de..., pack-reused 45
Unpacking objects: 100% (45/45), done.
Checking connectivity... done.
```

2. Then, let's build the Docker image with the `docker build` command by executing the following:

```
dockerhost$ cd training-webapp
dockerhost$ docker build -t hubuser/webapp .
Sending build context to Docker daemon 121.3 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu:14.04
Repository ubuntu already being ... another client. Waiting.
---> 6d4946999d4f
Step 1 : MAINTAINER Docker Education Team <education@docker.com>
---> Running in 0fd24c915568
---> e835d0c77b04
Removing intermediate container 0fd24c915568
Step 2 : RUN apt-get update
---> Running in 45b654e66939
Ign http://archive.ubuntu.com trusty InRelease
...
Removing intermediate container c08be35b1529
```

```
Step 9 : CMD python app.py
---> Running in 48632c5fa300
---> 55850135bada
Removing intermediate container 48632c5fa300
Successfully built 55850135bada
```



The `-t` flag is used to tag the image as `hubuser/webapp`. Tagging containers as `<username>/<imagename>` is an important convention to be able to push our Docker images in the later section. More details on the `docker build` command can be found at <https://docs.docker.com/reference/commandline/build> or by running `docker build --help`.

3. Finally, let's confirm that the image is already available in our Docker host with the `docker images` command:

```
dockerhost$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
hubuser/webapp	latest	55850135	5 minutes ago	360 MB
ubuntu	14.04	6d494699	3 weeks ago	188.3 MB

## Pushing Docker images to a repository

Now that we have made a Docker image, let's push it to a repository to share and deploy across other Docker hosts. The default installation of Docker pushes images to Docker Hub. Docker Hub is a publicly hosted repository of Docker, Inc., where anyone with an account can push and share their Docker images. The following steps will show us how to do this:

1. Before being able to push to Docker Hub, we will need to authenticate with the `docker login` command, as follows:

```
dockerhost$ docker login
Username: hubuser
Password: *****
Email: hubuser@hubemail.com
WARNING: login credentials saved in /home/hubuser/.dockercfg.
Login Succeeded
```

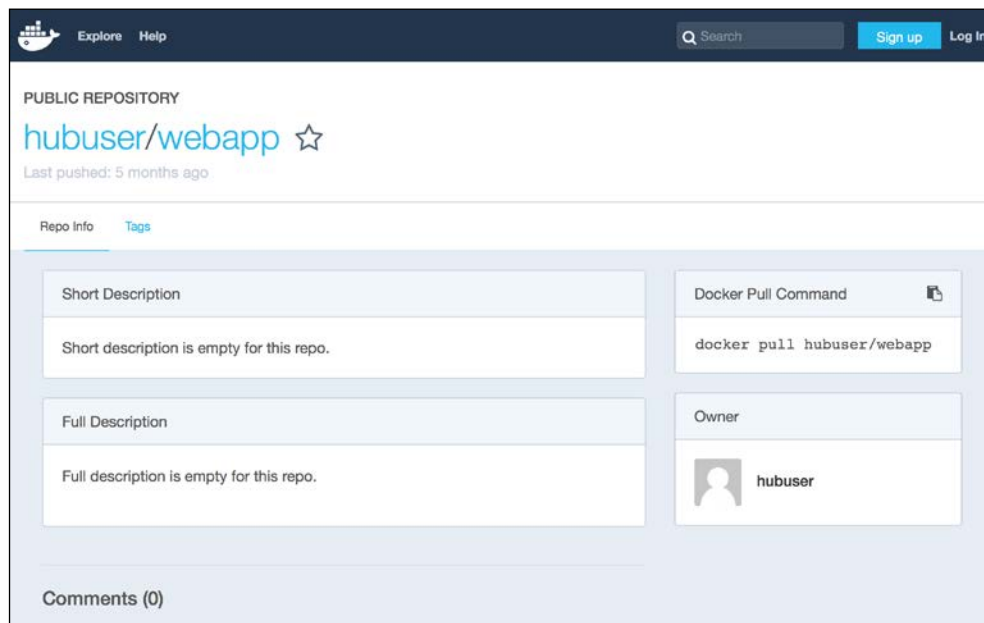



If we don't have a Docker Hub account yet, we can follow the instructions to sign up for an account at <https://hub.docker.com/account/signup>.

- We can now push our images to Docker Hub. As mentioned in the previous section, the tag of the image identifies `<username>/<imagename>` in the repository. Issue the `docker push` command shown as follows in order to push our image to Docker Hub:

```
dockerhost$ docker push hubuser/webapp
The push refers to a repository [hubuser/webapp] (len: 1)
Sending image list
Pushing repository hubuser/webapp (1 tags)
428b411c28f0: Image already pushed, skipping
...
7d04572a66ec: Image successfully pushed
55850135bada: Image successfully pushed
latest: digest: sha256:b00a3d4e703b5f9571ad6a... size: 2745
```

Now that we have successfully pushed our Docker image, it will be available in Docker Hub. We can also get more information about the image we pushed in its Docker Hub page, which is similar to that shown in the following image. In this example, our Docker Hub URL is `https://hub.docker.com/r/hubuser/webapp`:




 More details on pushing Docker images to a repository are available at `docker push --help` and <https://docs.docker.com/reference/commandline/push>.

Docker Hub is a good place to start hosting our Docker images. However, there are some cases where we want to host our own image repository. For example, when we want to save bandwidth when pulling images to our Docker hosts. Another reason could be that our Docker hosts inside a datacenter may have firewalled off the Internet. In *Chapter 2, Optimizing Docker Images*, we will discuss in greater detail how to run our own Docker registry to have an in-house repository of Docker images.

## Pulling Docker images from a repository

Once our Docker images are built and pushed to a repository, such as Docker Hub, we can pull them to our Docker hosts. This workflow is useful when we first build our Docker image in our development workstation Docker host and want to deploy it to our production environment's Docker host in the cloud. This removes the need to rebuild the same image in our other Docker hosts. Pulling images can also be used to grab the existing Docker images from Docker Hub to build over our own Docker images. So, instead of cloning the Git repository as we did earlier and redoing the build in another one of our Docker hosts, we can pull it instead. The next few steps will walk us through pulling the `hubuser/webapp` Docker image that we just pushed earlier:

1. First, let's clean our existing Docker host to make sure that we will download the image from Docker Hub. Type the following command to make sure we have a clean start:

```
dockerhost$ dockerhost rmi hubuser/webapp
```

2. Next, we can now download the image using `docker pull`, as follows:

```
dockerhost$ docker pull hubuser/webapp
latest: Pulling from hubuser/webapp
e9e06b06e14c: Pull complete
...
b37deb56df95: Pull complete
02a8815912ca: Already exists
Digest: sha256:06e9c1983bd6d5db5fba376ccd63bfa529e8d02f23d5
Status: Downloaded newer image for hubuser/webapp:latest
```

3. Finally, let's confirm again that we have downloaded the image successfully by executing the following command:

```
dockerhost$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
ubuntu	14.04	6d494699	3 weeks ago	188.3 MB
hubuser/webapp	latest	2a8815ca	7 weeks ago	348.8 MB



More details on how to pull Docker images is available at `docker pull --help` and <https://docs.docker.com/reference/commandline/pull>.

## Running Docker containers

Now that we have pulled or built Docker images, we can run and test them with the `docker run` command. This section will review selected command-line flags that we will use throughout the succeeding chapters. This section will also use the following Docker commands to get more information about the Docker containers being run inside the Docker host:

- `docker ps`
- `docker inspect`



More comprehensive details on all the command-line flags are found at `docker run --help` and <https://docs.docker.com/reference/commandline/run>.

## Exposing container ports

In the `training/webapp` example, its Docker container is run as a web server. To have the application serve web traffic outside its container environment, Docker needs information on which port the application is bound to. Docker refers to this information as exposed ports. This section will walk us through how to expose port information when running our containers.

Going back to the `training/webapp` Docker image we worked on earlier, the application serves a Python Flask web application that listens to port 5000, as highlighted here in `webapp/app.py`:

```
import os
from flask import Flask
app = Flask(__name__)
@app.route('/')
def hello():
    provider = str(os.environ.get('PROVIDER', 'world'))
    return 'Hello '+provider+'!'
if __name__ == '__main__':
    # Bind to PORT if defined, otherwise default to 5000.
    port = int(os.environ.get('PORT', 5000))
    app.run(host='0.0.0.0', port=port)
```



Correspondingly, the Docker image makes the Docker host aware that the application is listening on port 5000 via the EXPOSE instruction in the Dockerfile, which can be described as follows:

```
FROM ubuntu:14.04
MAINTAINER Docker Education Team <education@docker.com>
RUN apt-get update
RUN DEBIAN_FRONTEND=noninteractive apt-get \
    install -y -q python-all python-pip
ADD ./webapp/requirements.txt /tmp/requirements.txt
RUN pip install -qr /tmp/requirements.txt
ADD ./webapp /opt/webapp/
WORKDIR /opt/webapp
EXPOSE 5000
CMD ["python", "app.py"]
```

Now that we have a basic idea of how Docker exposes our container's ports, follow the next few steps to run the hubuser/webapp container:

1. Use `docker run` with the `-d` flag to run the container as a daemon process, as follows:

```
dockerhost$ docker run --name ourapp -d hubuser/webapp
```

2. Finally, confirm that the Docker host has the container running with port 5000 exposed with `docker ps`. We can do this through the following command:

```
dockerhost:~/training-webapp$ docker ps
CONTAINER ID   IMAGE      ...   STATUS      PORTS      NAMES
df3e6b788fd8  hubuser... Up 4 seconds 5000/tcp   ourapp
```

In addition to the EXPOSE instruction, exposed ports can be overridden during runtime with the `--expose=[]` flag. For example, use the following command to have the hubuser/webapp application expose ports 4000-4500:

```
dockerhost$ docker run -d --expose=4000-4500 \
    --name app hubuser/webapp

dockerhost $ docker ps
CONTAINER ID   IMAGE      ...   PORTS      NAMES
ca4dc1da26d    hubuser/webapp:latest  ...  4000-4500/tcp,5000/tcp  app
df3e6b788fd8  hubuser/webapp:1...    5000/tcp  ourapp
```

This ad hoc `docker run` flag is useful when debugging applications. For example, let's say our web application uses ports 4000-4500. However, we normally don't want these ranges to be available in production. We can then use `--expose=[]` to enable it temporarily to spin up a debuggable container. Further details on how to use techniques such as this to troubleshoot Docker containers will be discussed in *Chapter 7, Troubleshooting Containers*.

## Publishing container ports

Exposing only makes the port available inside the container. For the application to be served outside its Docker host, the port needs to be published. The `docker run` command uses the `-P` and `-p` flags to publish a container's exposed ports. This section talks about how to use these two flags to publish ports on the Docker host.

### --publish-all

The `-P` or `--publish-all` flag publishes all the exposed ports of a container to random high ports in the Docker host port within the ephemeral port range defined in `/proc/sys/net/ipv4/ip_local_port_range`. The next few steps will go back to the `hubuser/webapp` Docker image that we were working on to explore publishing exposed ports:

1. First, type the following command to run a container publishing all the exposed ports:

```
dockerhost$ docker run -P -d --name exposed hubuser/webapp
```

2. Next, let's confirm that the Docker host publishes port 32771 to forward traffic to the Docker container's exposed port 5000. Type the `docker ps` command as follows to perform this verification:

```
dockerhost$ docker ps
CONTAINER ID IMAGE ... PORTS NAMES
508cf1fb3e5 hubuser/webapp:latest ... 0.0.0.0:32771->5000/tcp exposed
```

3. We can also verify that the allocated port 32771 is within the configured ephemeral port range of our Docker host:

```
dockerhost$ cat /proc/sys/net/ipv4/ip_local_port_range
32768 61000
```

4. In addition, we can confirm that our Docker host is listening on the allocated port 32771 as well via the following command:

```
dockerhost$ ss -lt 'sport = *:32771'
State Recv-Q Send-Q Local Address:Port Peer Address:Port
LISTEN 0 128 :::32771 :::*
```

5. Finally, we can validate that the Docker host's port 32771 is indeed mapped to the running Docker container by confirming that it is the `training/webapp` Python application responding by making an actual HTTP request. Run the following command to confirm:

```
$ curl http://dockerhost:32771
Hello world!
```

## --publish

The `-p` or `--publish` flag publishes container ports to the Docker host. If the container port is not yet exposed, the said container will also be exposed. According to the documentation, the `-p` flag can take the following formats to publish container ports:

- `containerPort`
- `hostPort:containerPort`
- `ip::containerPort`
- `ip:hostPort:containerPort`

By specifying the `hostPort`, we can specify which port in the Docker host the container port should be mapped to instead of being assigned a random ephemeral port. By specifying `ip`, we can restrict the interfaces that the Docker host will accept connections from to relay the packets to the mapped Docker container's exposed port. Going back to the `hubuser/webapp` example, the following is the command to map the Python application's exposed port 5000 to our Docker host's port 80 on the loopback interface:

```
$ ssh dockerhost
dockerhost$ docker run -d -p 127.0.0.1:80:5000 training/webapp
dockerhost$ curl http://localhost
Hello world!
dockerhost$ exit
logout
Connection to dockerhost closed.
$ curl http://dockerhost
curl: (7) Failed connect to dockerhost:80; Connection refused
```

With the preceding invocation of `docker run`, the Docker host can only serve HTTP requests in the application from `http://localhost`.

## Linking containers

The published ports described in the previous section also allow containers to talk to each other by connecting to the published Docker host ports. Another way to directly connect containers with each other is establishing container links. Linked containers allow a source container to send information to the destination containers. It enables the communicating containers to discover each other in a secure manner.



More details about linked containers can be found on the Docker documentation site at <https://docs.docker.com/userguide/dockerlinks>.

In this section, we will work with the `--link` flag to connect containers securely. The next few steps give us an example of how to work with linked containers:

1. As preparation, make sure that our `hubuser/webapp` container runs with only the exposed ports. We will create a container called `source` that will serve as our source container. Type the following command to recreate this container:

```
dockerhost$ docker run --name source -d hubuser/webapp
```

2. Next, we will create a destination container. We will use `--link <source>:<alias>` to create a link from the source container named `source` to an alias called `webapp`. Type the following command to create this link to our destination container:

```
dockerhost$ docker run -d --link source:webapp \
    --name destination busybox /bin/ping webapp
```

3. Let's now confirm that the link was made by inspecting the newly created destination container called `destination`. Execute the following command:

```
dockerhost$ docker inspect -f "{{ .HostConfig.Links }}" \
    destination
[/source:/destination/webapp]
```

What happened during the linking process was that the Docker host created a secure tunnel between the two containers. We can confirm this tunnel in the Docker host's iptables, as follows:

```
dockerhost$ docker inspect -f "{{ .NetworkSettings.IPAddress }}" \
    source
172.17.0.15
dockerhost$ docker inspect -f "{{ .NetworkSettings.IPAddress }}" \
    destination
```

172.17.0.28

```
dockerhost$ iptables -L DOCKER
```

Chain DOCKER (1 references)

target	prot	opt	source	destination	
ACCEPT	tcp	--	172.17.0.28	172.17.0.15	tcp dpt:5000
ACCEPT	tcp	--	172.17.0.15	172.17.0.28	tcp spt:5000

In the preceding iptables, the Docker host allowed the destination container called `destination` (172.17.0.28) to accept outbound connections to port 5000 of the source container called `source` (172.17.0.15). The second iptable's entry allows the container called `source` to receive connections to its port 5000 from the container called `destination`.

In addition to the secure connections established by the Docker host between containers, the Docker host also exposes information about the source container to the destination container through the following:

- Environment variables
- Entries in `/etc/hosts`

These two sources of information will be further explored in the next section as an example use case of working with interactive containers.

## Interactive containers

By specifying the `-i` flag, we can specify that a container running in the foreground is attached to the standard input stream. By combining it with the `-t` flag, a pseudoterminal is also allocated to our container. With this, we can use our Docker container as an interactive process, similar to normal shells. This feature is useful when we want to debug and inspect what is happening inside our Docker containers. Continuing from the previous section, we can debug what happens when containers are linked through the following steps:

1. To prepare, type the following command to establish an interactive container session linking to the container called `source` that we ran earlier:

```
dockerhost$ docker run -i -t --link source:webapp \  
                    --name interactive_container \  
                    busybox /bin/sh  
  
/ #
```

2. Next, let's first explore the environment variables that are exposed to the interactive destination container via the following command:

```
/ # env | grep WEBAPP
WEBAPP_NAME=/interactive_container/webapp
WEBAPP_PORT_5000_TCP_ADDR=172.17.0.15
WEBAPP_PORT_5000_TCP_PORT=5000
WEBAPP_PORT_5000_TCP_PROTO=tcp
WEBAPP_PORT_5000_TCP=tcp://172.17.0.15:5000
WEBAPP_PORT=tcp://172.17.0.15:5000
```

In general, the following environment variables are set in linked containers:



- `<alias>_NAME=/container_name/alias_name` for each source container
- `<alias>_PORT_<port>_<protocol>` shows the URL of each exposed port. It also serves as a unique prefix expanding to the following more environment variables:
  - `<prefix>_ADDR` contains the IP address of the source container
  - `<prefix>_PORT` shows the exposed port's number
  - `<prefix>_PROTO` describes the protocol of the exposed port which is either TCP or UDP
- `<alias>_PORT` shows the source container's first exposed port

3. The second container discovery feature in linked containers is an updated `/etc/hosts` file. The alias of the `webapp` linked container is mapped to the IP address of the source container. The name of the source container is also mapped to the same IP address. The following snippet is the content of the `/etc/hosts` file inside our interactive container session, and it contains this mapping:

```
172.17.0.29      d4509e3da954
127.0.0.1       localhost
::1            localhost ip6-localhost ip6-loopback
fe00::0         ip6-localnet
ff00::0         ip6-mcastprefix
ff02::1         ip6-allnodes
ff02::2         ip6-allrouters
172.17.0.15     webapp 85173b8686fc source
```

4. Finally, we can use the alias to connect to our source container. In the following example, we will connect to the web application running in our source container by making an HTTP request to its alias, webapp:

```
/ # nc webapp 5000
GET /
```

```
Hello world!
/ #
```



Interactive containers can be used to build containers as well, together with `docker commit`. However, this is a tedious process, and this development process doesn't scale beyond a single developer. Use `docker build` instead and manage our Dockerfile in version control.

## Summary

Hopefully by this time, we are refamiliarized with most of the commands that will be used throughout the book. We prepared a Docker host to be able to interact with Docker containers. We then built, downloaded, and uploaded various Docker images to develop and deploy containers to our development and production Docker hosts alike. Finally, we ran Docker containers from built or downloaded Docker images. In addition, we established some basic skills of how to communicate and interact with running containers by learning about how Docker containers are run.

In the next chapter, you'll learn how to optimize our Docker images. So, let's dive right in!

# 2

## Optimizing Docker Images

Now that we have built and deployed our Docker containers, we can start reaping the benefits of using them. We have a standard package format that lets developers and sysadmins work together to simplify the management of our application's code. Docker's container format allows us to rapidly iterate the versions of our application and share it with the rest of our organization. Our development, testing, and deployment time has decreased because of the lightweight feature and speed of Docker containers. The portability of Docker containers allows us to scale our applications from physical servers to virtual machines in the cloud.

However, we will start noticing that the same reasons for which we used Docker in the first place are losing their effect. Development time is increasing because we have to always download the newest version of our application's Docker image runtime library. Deployment takes a lot of time because Docker Hub is slow. At worst, Docker Hub may be down, and we would not be able to do any deployment at all. Our Docker images are now so big, in the order of gigabytes, that simple single-line updates take the whole day.

This chapter will cover the following scenarios of how Docker containers get out of hand and suggest steps to remediate the problems mentioned earlier:

- Reducing image deployment time
- Reducing image build time
- Reducing image size



## Reducing deployment time

As time goes by while we build our Docker container, its size gets bigger and bigger. Updating running containers in our existing Docker hosts is not a problem. Docker takes advantage of the Docker image layers that we build over time as our application grows. However, consider a case in which we want to scale out our application. This requires deploying more Docker containers to additional Docker hosts. Each new Docker host has to then download all the large image layers that we built over time. This section will show you how a *large* Docker application affects deployment time on new Docker hosts. First, let's build this problematic Docker application by carrying out the following steps:

1. Write the following Dockerfile to create our "large" Docker image:

```
FROM debian:jessie
```

```
RUN dd if=/dev/urandom of=/largefile bs=1024 count=524288
```

2. Next, build the Dockerfile as hubuser/largeapp using the following command:

```
dockerhost$ docker build -t hubuser/largeapp .
```

3. Take note of how large the created Docker image is. In the following illustrated output, the size is 662 MB:

```
dockerhost$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
hubuser/largeapp	latest	450e3123	5 minutes ago	662 MB
debian	jessie	9a61b6b1	4 days ago	125.2 MB

4. Using the time command, record how long it takes to push and pull it from Docker Hub, as follows:

```
dockerhost$ time docker push hubuser/largeapp
```

```
The push refers to a repository [hubuser/largeapp] (len: 1)
```

```
450e319e42c3: Image already exists
```

```
9a61b6b1315e: Image successfully pushed
```

```
902b87aaaec9: Image successfully pushed
```

```
Digest: sha256:18ef52e36996dd583f923673618483a4466aa2d1d0d6ce9f0...
```

```
real    11m34.133s
```

```
user     0m0.164s
```

```
sys      0m0.104s
```

```
dockerhost$ time docker pull hubuser/largeapp
```

```

latest: Pulling from hubuser/largeapp
902b87aaaec9: Pull complete
9a61b6b1315e: Pull complete
450e319e42c3: Already exists
Digest: sha256:18ef52e36996dd583f923673618483a4466aa2d1d0d6ce
9f0...
Status: Downloaded newer image for hubuser/largeapp:latest

real    2m56.805s
user    0m0.204s
sys     0m0.188s

```

As we can note in the preceding time values highlighted, it takes a lot of time when we perform `docker push` to upload an image to Docker Hub. Upon deployment, `docker pull` takes just as long in order to propagate our newly created Docker image to our new production Docker hosts. These upload and download time values also depend on the network connection between Docker Hub and our Docker hosts. Ultimately, when Docker Hub goes down, we will lose the ability to deploy new Docker containers or scale out to additional Docker hosts on demand.

In order to take advantage of Docker's fast delivery of applications and ease of deployment and scaling, it is important that our method of pushing and pulling Docker images is reliable and fast. Fortunately, we can run our own Docker registry to be able to host and distribute our Docker images without relying on the public Docker Hub. The next few steps describe how to set this up to confirm the improvement in performance:

1. Let's run our own Docker registry by typing the following command. This gives us a local one running at `tcp://dockerhost:5000`:
2. Next, let's confirm how our Docker image deployments have improved. First, create a tag for the image we created earlier in order to push it to the local Docker registry via the following:

```

dockerhost$ docker run -p 5000:5000 -d registry:2

dockerhost$ docker tag hubuser/largeapp \
                dockerhost:5000/largeapp

```

3. Observe how much faster it is to push the same Docker image over our newly running Docker registry. The following tests show that pushing Docker images is now at least 10 times faster:

```

dockerhost$ time docker push dockerhost:5000/largeapp
The push refers to a ...[dockerhost:5000/largeapp] (len: 1)

```

...


```
real    0m52.928s
user    0m0.084s
sys     0m0.048s
```

4. Now, confirm the new performance of the pulling of our Docker images before testing that of the pulling of images from our local Docker registry. Let's make sure we remove the image we built earlier. The following tests show that the downloading of Docker images is now 30 times faster:

```
dockerhost$ docker rmi dockerhost:5000/largeapp \
                hubuser/largeapp
Untagged: dockerhost:5000/largeapp:latest
Untagged: hubuser/largeapp:latest
Deleted:
549d099c0edaef424edb6cfca8f16f5609b066ba744638990daf3b43...
dockerhost$ time docker pull dockerhost:5000/largeapp
latest: Pulling from dockerhost:5000/largeapp
549d099c0eda: Already exists
902b87aaaec9: Already exists
9a61b6b1315e: Already exists
Digest: sha256:323bed623625b3647a6c678ee6840be23616edc357dbe07c5a0
c68b62dd52ecf
Status: Downloaded newer image for dockerhost:5000/largeapp:latest

real    0m10.444s
user    0m0.160s
sys     0m0.056s
```

The main cause of these improvements is that we uploaded and downloaded the same images from our local network. We saved on the bandwidth of our Docker hosts, and our deployment time got shorter. The best part of all is that we no longer have to rely on the availability of Docker Hub in order to deploy.

[  In order to deploy our Docker images to other Docker hosts, we need to set up security for our Docker registry. Details on how to set this up are outside the scope of this book. However, more details on how to set up a Docker registry are available at <https://docs.docker.com/registry/deploying>. ]

## Improving image build time

Docker images are the main resulting artifacts that developers work on all the time. The simplicity of Docker files and speed of container technology allows us to enable rapid iteration on the application that we are working on. However, these advantages of using Docker start to diminish once the time it takes to build Docker images starts to grow uncontrollably. In this section, we will discuss some cases of building Docker images that take some time to run. Then, we will give you a few tips on how to remediate these effects.

## Using registry mirrors

A big contributor to image build time is the time spent in fetching upstream images. Suppose we have a `Dockerfile` with the following line:

```
FROM java:8u45-jre
```

This image will have to download `java:8u45-jre` to be built. When we move to another Docker host, or if the `java:8u45-jre` image is updated in Docker Hub, our build time will increase momentarily. Configuring a local registry mirror will reduce such image build time instances. This is very useful in an organization setting, where each developer has his/her own Docker hosts at their workstations. The organization's network only downloads the image from Docker Hub once. Each workstation Docker host in the organization can now directly fetch the images from the local registry mirror.

Setting up a registry mirror is as simple as setting up a local registry in the previous section. However, in addition, we need to configure the Docker host to be aware of this registry mirror by passing the `--registry-mirror` option to the Docker daemon. Here are the steps to perform this setup:

1. In our Debian Jessie Docker host, configure the Docker daemon by updating and creating a Systemd drop-in file at `/etc/systemd/system/docker.service.d/10-syslog.conf` to contain the following line:

```
[Service]
ExecStart=
ExecStart=/usr/bin/docker daemon-H fd:// \
--registry-mirror=http://dockerhost:5000
```

2. Now, we will reload Systemd to pick up the new drop-in configuration for the `docker.service` unit, as follows:

```
dockerhost$ systemctl daemon-reload
```

3. Next, restart the Docker daemon to start it with the newly configured Systemd unit via the following command:

```
dockerhost$ systemctl restart docker.service
```

4. Finally, run the registry mirror Docker container. Run the following command:

```
dockerhost$ docker run -p 5000:5000 -d \
    -e STANDALONE=false \
    -e MIRROR_SOURCE=https://registry-1.docker.io \
    -e MIRROR_SOURCE_INDEX=https://index.docker.io \
    registry
```

To confirm that the registry mirror works as expected, perform the following steps:

1. Build the Dockerfile described at the start of this subsection and take note of its build time. Note that most of the time needed to build the Docker image is taken up by the time to download the upstream `java:8u45-jre` Docker image, as shown in the following command:

```
dockerhost$ time docker build -t hubuser/mirrorupstream .
Sending build context to Docker daemon 2.048 kB
Sending build context to Docker daemon
Step 0 : FROM java:8u45-jre
Pulling repository java
4ac125456dd3: Download complete
902b87aaaec9: Download complete
9a61b6b1315e: Download complete
1ff9f26f09fb: Download complete
6f6bffbbf095: Download complete
4b61c52d7fe4: Download complete
1a9b1e5c4dd5: Download complete
2e8cff440182: Download complete
46bc3bbea0ec: Download complete
3948efdeee11: Download complete
918f0691336e: Download complete
Status: Downloaded newer image for java:8u45-jre
--> 4ac125456dd3
Successfully built 4ac125456dd3

real    1m58.095s
user    0m0.036s
sys     0m0.028s
```

- Now, remove the image and its upstream dependency and rebuild the image again using the following commands:


```

dockerhost$ docker rmi java:8u45-jre hubuser/mirrorupstream
dockerhost$ time docker build -t hubuser/mirrorupstream .
Sending build context to Docker daemon 2.048 kB
Sending build context to Docker daemon
Step 0 : FROM java:8u45-jre
Pulling repository java
4ac125456dd3: Download complete
902b87aaaec9: Download complete
9a61b6b1315e: Download complete
1ff9f26f09fb: Download complete
6f6bffbbf095: Download complete
4b61c52d7fe4: Download complete
1a9b1e5c4dd5: Download complete
2e8cff440182: Download complete
46bc3bbea0ec: Download complete
3948efdeee11: Download complete
918f0691336e: Download complete
Status: Downloaded newer image for java:8u45-jre
--> 4ac125456dd3
Successfully built 4ac125456dd3

real    0m59.260s
user    0m0.032s
sys     0m0.028s

```

When the `java:8u45-jre` Docker image was downloaded for the second time, it was retrieved from the local registry mirror instead of being connected to Docker Hub. Setting up a Docker registry mirror improved the time of downloading the upstream image by almost two times the usual. If we have other Docker hosts pointed at this same registry mirror, it will do the same thing: skip the downloading from Docker Hub.


 This guide on how to set up a registry mirror is based on the one on the Docker documentation website. More details can be found at [https://docs.docker.com/articles/registry\\_mirror](https://docs.docker.com/articles/registry_mirror).

## Reusing image layers

As we already know, a Docker image consists of a series of layers combined using the union filesystem of a single image. When we work on building our Docker image, the preceding instructions in our `Dockerfile` are examined by Docker to check whether there is an existing image in its build cache that can be reused instead of creating a similar or duplicate image for these instructions. By finding out how the build cache works, we can greatly increase the speed of the subsequent builds of our Docker images. A good example of this is when we develop our application's behavior; we will not add dependencies to our application all the time. Most of the time, we will just want to update the core behavior of the application itself. Knowing this, we can design the way we will build our Docker images around this in our development workflow.



Detailed rules on how Dockerfile instructions are cached can be found at [http://docs.docker.com/articles/dockerfile\\_best-practices/#build-cache](http://docs.docker.com/articles/dockerfile_best-practices/#build-cache).

For example, suppose we are working on a Ruby application whose source tree looks similar to the following:

Name	Size
config.ru	62 bytes
Dockerfile	92 bytes
Gemfile	57 bytes

The `config.ru` would be as follows:

```
app = proc do |env|
  [200, {}, %w(hello world)]
end
run app
```

The `Gemfile` would be as follows:

```
source 'https://rubygems.org'

gem 'rack'
gem 'nokogiri'
```

The Dockerfile would be as follows:

```
FROM ruby:2.2.2

ADD . /app
WORKDIR /app
RUN bundle install

EXPOSE 9292
CMD rackup -E none
```

The following steps will show you how to build the Ruby application we wrote earlier as a Docker image:

1. First, let's build this Docker image through the following command. Note that the time it took to build is around one minute:

```
dockerhost$ time docker build -t slowdependencies .
Sending build context to Docker daemon 4.096 kB
Sending build context to Docker daemon
Step 0 : FROM ruby:2.2.2
---> d763add83c94
Step 1 : ADD . /app
---> 6663d8b8b5d4
Removing intermediate container 2fda8dc40966
Step 2 : WORKDIR /app
---> Running in f2bec0dealc9
---> 289108c6655f
Removing intermediate container f2bec0dealc9
Step 3 : RUN bundle install
---> Running in 7025de40c01d
Don't run Bundler as root. Bundler can ask for sudo if ...
Fetching gem metadata from https://rubygems.org/.....
Fetching version metadata from https://rubygems.org/..
Resolving dependencies...
Installing mini_portile 0.6.2
Installing nokogiri 1.6.6.2 with native extensions
Installing rack 1.6.4
Using bundler 1.10.5
Bundle complete! 2 Gemfile dependencies, 4 gems now installed.
```



```
Bundled gems are installed into /usr/local/bundle.
```

```
---> ab26818ccd85
```

```
Removing intermediate container 7025de40c01d
```

```
Step 4 : EXPOSE 9292
```

```
---> Running in e4d7647e978b
```

```
---> a602159cb786
```

```
Removing intermediate container e4d7647e978b
```

```
Step 5 : CMD rackup -E none
```

```
---> Running in 407308682d13
```

```
---> bffce44702f8
```

```
Removing intermediate container 407308682d13
```

```
Successfully built bffce44702f8
```

```
real    0m54.428s
```

```
user    0m0.004s
```

```
sys     0m0.008s
```

2. Next, update `config.ru` to change the application's behavior, as follows:

```
app = proc do |env|  
  [200, {}, %w(hello other world)]  
end  
run app
```

3. Let's now build again the Docker image and note the time it takes to finish the build. Run the following command:

```
dockerhost$ time docker build -t slowdependencies .
```

```
Sending build context to Docker daemon 4.096 kB
```

```
Sending build context to Docker daemon
```

```
Step 0 : FROM ruby:2.2.2
```

```
---> d763add83c94
```

```
Step 1 : ADD . /app
```

```
---> 05234a367589
```

```
Removing intermediate container e9d33db67914
```

```
Step 2 : WORKDIR /app
```

```
---> Running in 65b3f40d6228
```

```
---> c656079a833f
```

```
Removing intermediate container 65b3f40d6228
```

```
Step 3 : RUN bundle install
```

---

```
---> Running in c84bd4aa70a0
Don't run Bundler as root. Bundler can ask for sudo ...
Fetching gem metadata from https://rubygems.org/.....
Fetching version metadata from https://rubygems.org/..
Resolving dependencies...
Installing mini_portile 0.6.2
Installing nokogiri 1.6.6.2 with native extensions
Installing rack 1.6.4
Using bundler 1.10.5
Bundle complete! 2 Gemfile dep..., 4 gems now installed.
Bundled gems are installed into /usr/local/bundle.
---> 68f5dc363171
Removing intermediate container c84bd4aa70a0
Step 4 : EXPOSE 9292
---> Running in 68c1462c2018
---> c257c74eb7a8
Removing intermediate container 68c1462c2018
Step 5 : CMD rackup -E none
---> Running in 7e13fd0c26f0
---> e31f97d2d96a
Removing intermediate container 7e13fd0c26f0
Successfully built e31f97d2d96a

real    0m57.468s
user    0m0.008s
sys     0m0.004s
```

We can note that even with a single-line change to our application, we have to run `bundle install` for each iteration of the Docker image that we are building. This can be very inefficient, and it disrupts the flow of our development because it takes one minute to build and run our Docker application. For impatient developers such as us, this feels like an eternity!

In order to optimize this workflow, we can separate the phase in which we prepare our application's dependencies from that in which we prepare its actual artifacts. The next steps show us how to do this:

1. First, update our Dockerfile with the following changes:

```
FROM ruby:2.2.2
```

```
ADD Gemfile /app/Gemfile
```

```
WORKDIR /app
```

```
RUN bundle install
```

```
ADD . /app
```

```
EXPOSE 9292
```

```
CMD rackup -E none
```

2. Next, build the newly refactored Docker image via this command:

```
dockerhost$ time docker build -t separatedependencies .
```

```
Sending build context to Docker daemon 4.096 kB
```

```
Sending build context to Docker daemon
```

```
...
```

```
Step 3 : RUN bundle install
```

```
---> Running in b4cbc6803947
```

```
Don't run Bundler as root. Bundler can ask for sudo if it is  
needed, and
```

```
installing your bundle as root will break this application for all  
non-root
```

```
users on this machine.
```

```
Fetching gem metadata from https://rubygems.org/.....
```

```
Fetching version metadata from https://rubygems.org/..
```

```
Resolving dependencies...
```

```
Installing mini_portile 0.6.2
```

```
Installing nokogiri 1.6.6.2 with native extensions
```

```
Installing rack 1.6.4
```

```
Using bundler 1.10.5
```

```
Bundle complete! 2 Gemfile dependencies, 4 gems now installed.
```

```
Bundled gems are installed into /usr/local/bundle.
```

```
---> 5c009ed03934
```

```
Removing intermediate container b4cbc6803947
```

```
Step 4 : ADD . /app
...
Successfully built ff2d4efd233f
```

```
real    0m57.908s
user    0m0.008s
sys     0m0.004s
```

3. The build time is still the same at first, but note the image ID generated in Step 3. Now, try updating `config.ru` again and rebuilding the image, as follows:

```
dockerhost$ vi config.ru # edit as we please
dockerhost$ time docker build -t separatedependencies .
Sending build context to Docker daemon 4.096 kB
Sending build context to Docker daemon
Step 0 : FROM ruby:2.2.2
---> d763add83c94
Step 1 : ADD Gemfile /app/Gemfile
---> Using cache
---> a7f68475cf92
Step 2 : WORKDIR /app
---> Using cache
---> 203b5b800611
Step 3 : RUN bundle install
---> Using cache
---> 5c009ed03934
Step 4 : ADD . /app
---> 30b2bfc3f313
Removing intermediate container cd643f871828
Step 5 : EXPOSE 9292
---> Running in a56bfd37f721
---> 553ae65c061c
Removing intermediate container a56bfd37f721
Step 6 : CMD rackup -E none
---> Running in 0ceaa70bee6c
---> 762b7ccf7860
Removing intermediate container 0ceaa70bee6c...
```

```
Successfully built 762b7ccf7860
```

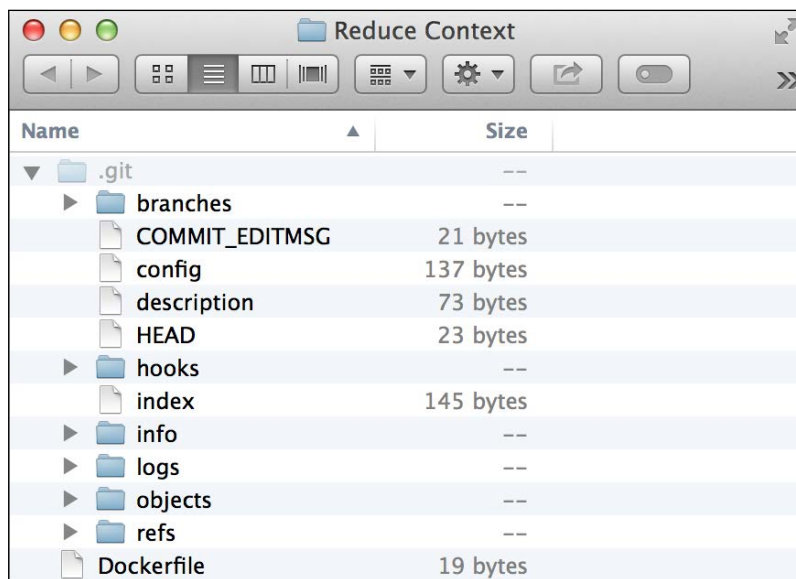
```
real    0m0.734s
user    0m0.008s
sys     0m0.000s
```

As we can note in the preceding output, `docker build` reused the cache until `Step 3` as there was no change in `Gemfile`. Note that our Docker image's build time decreased by 80 times the usual!

This kind of refactoring for our Docker image is also useful to reduce deployment time. As our Docker hosts in production already have image layers until `Step 3` of our Docker image in the previous version of our container, having a new version of our Docker application will only require the Docker host to pull new image layers for `Step 4` to `Step 6` in order to update our application.

## Reducing the build context size

Let's suppose that we have a `Dockerfile` in the Git version control similar to the following:



At some point, we will notice that our `.git` directory is too big. This is probably the result of having more and more code committed into our source tree:

```
dockerhost$ du -hsc .git
1001M    .git
1001M    total
```

Now, when we build our Docker application, we will notice that the time taken to build our Docker application is very big as well. Take a look at the following output:

```
dockerhost$ time docker build -t hubuser/largecontext .
Sending build context to Docker daemon 1.049 GB
Sending build context to Docker daemon
...
Successfully built 9a61b6b1315e

real    0m17.342s
user    0m0.408s
sys     0m1.360s
```

If we look closely at the preceding output, we will see that the Docker client uploaded the whole `.git` directory of 1 GB onto the Docker daemon because it is a part of our build context. Also, as this is a large build context, it takes time for the Docker daemon to receive it before being able to start building our Docker image.

However, these files are not necessary to build our application. Moreover, these Git-related files are not at all needed when we run our application in production. We can set Docker to ignore a specific set of files that are not needed to build our Docker image. Follow the next few steps to perform this optimization:

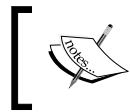
1. Create a `.dockerignore` file with the following content in the same directory as our `Dockerfile`:  
`.git`

2. Finally, build our Docker image again by executing the following command:

```
dockerhost$ time docker build -t hubuser/largecontext .
Sending build context to Docker daemon 3.072 kB
...
Successfully built 9a61b6b1315e

real    0m0.030s
user    0m0.004s
sys     0m0.004s
```

Note now that the build time is improved by over 500 times the usual just by decreasing the size of the build context!



More information on how to use `.dockerignore` files can be found at <https://docs.docker.com/reference/builder/#dockerignore-file>.

## Using caching proxies

Another common source causing the long runtime in building Docker images are instructions that download dependencies. For example, a Debian-based Docker image needs to fetch packages from APT repositories. Depending on how large these packages are, the build time for an `apt-get install` instruction may be long. A useful technique to reduce the time for these build instructions is to introduce proxies that cache such dependency packages. A popular caching proxy is `apt-cacher-ng`. This section will describe running and setting it up to improve our Docker image building workflow.

The following is an example `Dockerfile` that installs a lot of Debian packages:

```
FROM debian:jessie

RUN echo deb http://httpredir.debian.org/debian \
    jessie-backports main > \
    /etc/apt/sources.list.d/jessie-backports.list
RUN apt-get update &&\
    apt-get --no-install-recommends \
    install -y openjdk-8-jre-headless
```

Note that its build time in the following output is quite long because this Dockerfile file downloads a lot of dependencies and packages related to Java (openjdk-8-jre-headless). Run the following command:

```
dockerhost$ time docker build -t beforecaching .
```

```
...
```

```
Successfully built 476f2ebd35f6
```

```
real    3m22.949s
```

```
user    0m0.048s
```

```
sys     0m0.020s
```

In order to improve the workflow for building this Docker image, we will set up a caching proxy with apt-cacher-ng. Fortunately, it is already available as a ready-to-run container from Docker Hub. Follow the next few steps to prepare apt-cacher-ng:

1. Run the following command in our Docker host to start apt-cacher-ng:  
**dockerhost\$ docker run -d -p 3142:3142 sameersbn/apt-cacher-ng**

2. After this, we will use the caching proxy we ran earlier, as described in the following Dockerfile:

```
FROM debian:jessie
```

```
RUN echo Acquire::http { \
    Proxy\"http://dockerhost:3142\"; \
} > /etc/apt/apt.conf.d/01proxy
```

3. Build the Dockerfile we created earlier as a Docker image tagged as hubuser/debian:jessie via the following command line:

```
dockerhost$ docker build -t hubuser/debian:jessie
```

4. Finally, make hubuser/debian:jessie our new base Docker image by updating our Dockerfile that installs a lot of Debian packages for dependencies such as the following:

```
FROM hubuser/debian:jessie
```

```
RUN echo deb http://httpredir.debian.org/debian \
    jessie-backports main > \
    /etc/apt/sources.list.d/jessie-backports.list
RUN apt-get update && \
    apt-get --no-install-recommends \
    install -y openjdk-8-jre-headless
```



5. To confirm the new workflow, run an initial build to warm up the cache using the following command:
6. Finally, execute the following commands to build the image again. However, make sure to remove the image first:

```
dockerhost$ docker build -t aftercaching .

dockerhost$ docker rmi aftercaching
dockerhost$ time docker build -t aftercaching .

...

Removing intermediate container 461637e26e05
Successfully built 2b80ca0d16fd

real    0m31.049s
user    0m0.044s
sys     0m0.024s
```

Note how the subsequent build is faster even though we do not use Docker's build cache. This technique is useful when we develop base Docker images for our team or organization. Team members that try to rebuild our Docker image will run their builds 6.5 times faster because they can download packages from our organization's cache proxy that we prepared earlier. Builds on our continuous integration server will also be faster upon check-in because we already warmed up the caching server during development.

This section gave a glance at how to use a very specific caching server. Here are a few others that we can use and their corresponding pages of documentation:

- **apt-cacher-ng:** This supports caching Debian, RPM, and other distribution-specific packages and can be found at <https://www.unix-ag.uni-kl.de/~bloch/acng>.
- **Sonatype Nexus:** This supports Maven, Ruby Gems, PyPI, and NuGet packages out of the box. It is available at <http://www.sonatype.org/nexus>.
- **Polipo:** This is a generic caching proxy useful for development that can be found at <http://www.pps.univ-paris-diderot.fr/~jch/software/polipo>.
- **Squid:** This is another popular caching proxy that can work with other types of network traffic as well. You can look this up at <http://www.squid-cache.org>.

## Reducing Docker image size

As we keep working on our Docker applications, the size of images tends to get bigger and bigger if we are not careful. Most people using Docker observe that their team's custom Docker images increase in size to at least 1 GB or more. Having larger images means that the time to build and deploy our Docker application increases as well. As a result, the feedback we get to determine the result of the application we're deploying gets reduced. This diminishes the benefits of Docker, enabling us to develop and deploy our applications in rapid iterations.

This section examines some further details of how Docker's image layers work and how they affect the size of the resulting image. Next, we will learn how to optimize these image layers by exploiting how Docker images work.

## Chaining commands

Docker images get big because some instructions are added that are unnecessary to build or run an image. A popular use case is packaging metadata and cache. After installing the packages necessary to build and run our application, such downloaded packages are no longer needed. The following patterns of instructions in a Dockerfile are commonly found in the wild (such as in Docker Hub) to *clean* the images of such unnecessary files from Docker images:

```
FROM debian:jessie

RUN echo deb http://httpredir.debian.org/debian \
    jessie-backports main \
    > /etc/apt/sources.list.d/jessie-backports.list
RUN apt-get update
RUN apt-get --no-install-recommends \
    install -y openjdk-8-jre-headless
RUN rm -rfv /var/lib/apt/lists/*
```

However, a Docker image's size is basically the sum of each individual layer image; this is how union filesystems work. Hence, the *clean* steps do not really delete the space. Take a look at the following commands:

```
dockerhost$ docker build -t fakeclean .
dockerhost$ docker history fakeclean
```

IMAGE	CREATED	CREATED BY	SIZE
33c8eedfc24a	2 minutes ago	/bin/sh -c rm -rfv /var/lib...	0 B
48b87c35b369	2 minutes ago	/bin/sh -c apt-get install ...	318.6 MB
dad9efad9e2d	4 minutes ago	/bin/sh -c apt-get update	9.847 MB

a8f7bf731a7d	5 minutes ago	/bin/sh -c echo 'deb http:/...	61 B
9a61b6b1315e	6 days ago	/bin/sh -c #(nop) CMD ["/bi...	0 B
902b87aaaec9	6 days ago	/bin/sh -c #(nop) ADD file:...	125.2 MB

There is no such thing as "negative" layer size. Hence, each instruction in a Dockerfile can only keep the image size constant or increase it. Also, as each step also introduces some metadata, the total size keeps increasing.

In order to reduce the total image size, the cleaning steps should be performed in the same image layer. Hence, the solution is to chain commands from the previously multiple instructions into a single one. As Docker uses `/bin/sh` to run each instruction, we can use the Bourne shell's `&&` operator to perform the chaining, as follows:

```
FROM debian:jessie

RUN echo deb http://httpredir.debian.org/debian \
    jessie-backports main \
    > /etc/apt/sources.list.d/jessie-backports.list
RUN apt-get update && \
    apt-get --no-install-recommends \
        install -y openjdk-8-jre-headless && \
    rm -rfv /var/lib/apt/lists/*
```

Note how each individual layer is much smaller now. As the individual layers' sizes were reduced, the total image size also decreased. Now, run the following commands and take a look at the output:

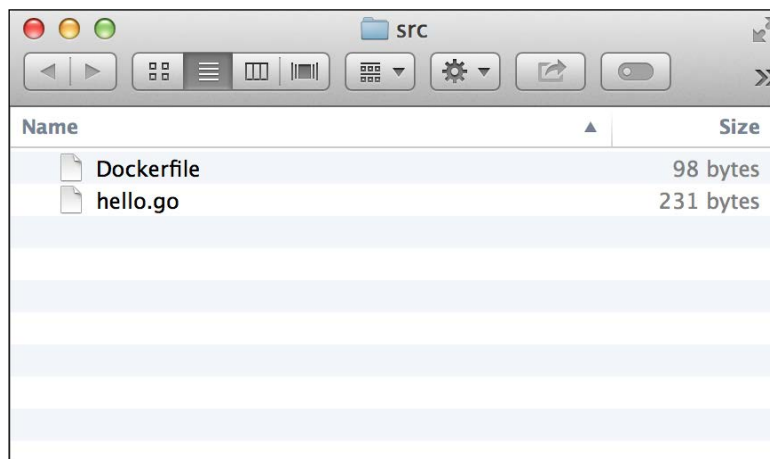
```
dockerhost$ docker build -t trueclean .
dockerhost$ docker history trueclean
```

IMAGE	CREATED	CREATED BY	SIZE
03d0b15bad7f	About a minute ago	/bin/sh -c apt-get update...	318.6 MB
a8f7bf731a7d	9 minutes ago	/bin/sh -c echo deb h...	61 B
9a61b6b1315e	6 days ago	/bin/sh -c #(nop) CMD...	0 B
902b87aaaec9	6 days ago	/bin/sh -c #(nop) ADD...	125.2 MB

## Separating build and deployment images

Another source of unnecessary files in Docker images are build time dependencies. Source libraries, such as compilers and source header files, are only necessary when building an application inside a Docker image. Once the application is built, these files are no longer necessary as only the compiled binary and related shared libraries are needed to run the application.

For example, build the following application that is now ready to be deployed to a Docker host that we prepared in the cloud. The following source tree is a simple web application written in Go:



The following is the content of `hello.go` describing the application:

```
package main

import (
    "fmt"
    "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "hello world")
}

func main() {
    http.HandleFunc("/", handler)
    http.ListenAndServe(":8080", nil)
}
```

The following corresponding Dockerfile shows how to build the source code and run the resulting binary:

```
FROM golang:1.4.2

ADD hello.go hello.go
RUN go build hello.go
EXPOSE 8080
ENTRYPOINT ["/hello"]
```

In the next few steps, we will show you how this Docker application's image size gets big:

1. First, build the Docker image and note its size. We will run the following commands for this:

```
dockerhost$ docker build -t largeapp .
dockerhost$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
largeapp	latest	47a64e67fb81	4 minute...	523.1 MB
golang	1.4.2	124e2127157f	5 days ago	517.3 MB

2. Now, compare this to the size of the actual application that is run, as follows:

```
dockerhost$ docker run --name large -d largeapp
dockerhost$ docker exec -it large/bin/ls -lh
total 5.6M
drwxrwxrwx 2 root root 4.0K Jul 14 06:26 bin
-rwxr-xr-x 1 root root 5.6M Jul 20 02:40 hello
-rw-r--r-- 1 root root 231 Jul 18 05:59 hello.go
drwxrwxrwx 2 root root 4.0K Jul 14 06:26 src
```

One of the advantages of writing Go applications, and compiled code in general, is that we can produce a single binary that is easy to deploy. The remaining size of the Docker image is made up of the unnecessary files provided by the base Docker image. We can note the large overhead coming from the base Docker image that increases the total image size by 100 times the usual.

We can also optimize the end Docker image deployed to production by only packing the final `hello` binary and some dependent shared libraries. Follow the next few steps to perform the optimization:

1. First, copy the binary from the running container to our Docker host via the following command line:

```
dockerhost$ docker cp -L large:/go/hello ../build
```

2. If the preceding library were a static binary, we would now be done and would proceed with the next step. However, Go tooling builds share binaries by default. In order for the binary to run properly, it needs the shared libraries. Run the following command to list them:

```
dockerhost$ docker exec -it large /usr/bin/ldd hello
linux-vdso.so.1 (0x00007ffd84747000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0
(0x00007f32f3793000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
(0x00007f32f33ea000)
/lib64/ld-linux-x86-64.so.2 (0x00007f32f39b0000)
```

3. Next, save all required shared libraries to our Docker host. Issuing the following `docker cp -L` commands will do this:

```
dockerhost$ docker cp -L large:/lib/x86_64-linux-gnu/libpthread.so.0 \
    ../build
dockerhost$ docker cp -L large:/lib/x86_64-linux-gnu/libc.so.6 \
    ../build
dockerhost$ docker cp -L large:/lib64/ld-linux-x86-64.so.2 \
    ../build
```

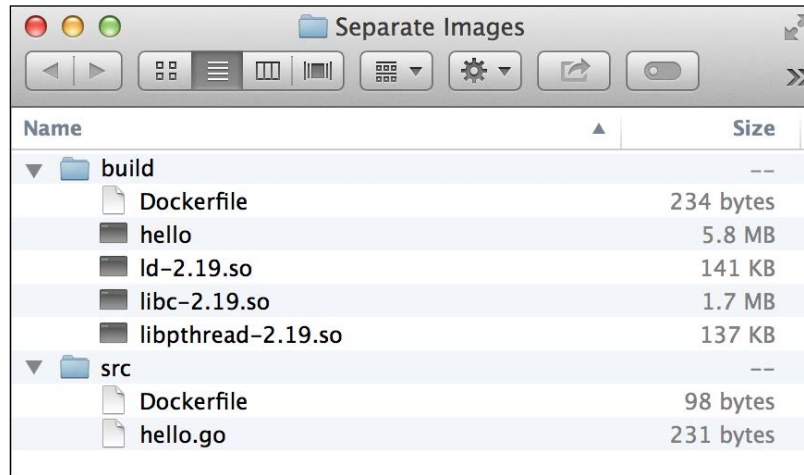
4. Create a new Dockerfile to build this "binary-only" image. Note how the `ADD` instructions recreate the shared library paths that the `hello` application expects in this new Docker image in the following output:

```
FROM scratch

ADD hello /app/hello
ADD libpthread-2.19.so \
    /lib/x86_64-linux-gnu/libpthread.so.0
ADD libc-2.19.so /lib/x86_64-linux-gnu/libc.so.6
ADD ld-2.19.so /lib64/ld-linux-x86-64.so.2

EXPOSE 8080
ENTRYPOINT ["/app/hello"]
```

- Now we have all the necessary files needed to run the new "binary-only" Docker image. In the end, the files in our directory tree will look similar to the following screenshot:



- Now, build the deployable binary Docker image with the following build/Dockerfile. The resulting image will be smaller now:

```
dockerhost$ docker build -t binary .
```

```
dockerhost$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
binary	latest	45c327c815	seconds ago	7.853 MB
largeapp	latest	47a64e67f	52 minutes ago	523.1 MB
golang	1.4.2	124e21271	5 days ago	517.3 MB

The same approach can also be used to make other compiled applications, such as the software normally installed using the `./configure && make && make install` combinations. We can do the same for interpreted languages such as Python, Ruby, or PHP. However, it will need a little more work to create a "runtime" Ruby Docker image from a "build" Ruby Docker image. An example of a good time to perform this kind of optimization is when the delivery of our applications gets too long because the images are too big for a sustainable development workflow.

## Summary

In this chapter, you learned more about how Docker builds images and applied it to improve several factors, such as the deploy time, build time, and image size. The techniques specified in this chapter are not comprehensive; there will surely be more ways on how to achieve these objectives as more people discover how to use Docker for their applications. More techniques will also arise as Docker itself matures and develops more features. The most important guiding factor for these optimizations is to ask ourselves whether we are really getting the benefits of using Docker. Some good example questions to ask are as follows:

- Is deploy time improving?
- Is the development team getting feedback fast enough from what the operations team learned when running our application?
- Are we able to iterate on new features fast enough to incorporate the new feedback that we discovered from customers using our application?

By keeping in mind our motivation and objective of using Docker, we can come with our own ways to improve our workflows.

Using some of the preceding optimizations will require updating the configuration of our Docker hosts. To be able to manage several Docker hosts at a scale, we will need some form of automation for their provisioning and configuration. In the next chapter, we will talk about how to automate setting up Docker hosts with configuration management software.





# 3

## Automating Docker Deployments with Chef

By this time, we already know the various aspects of the Docker ecosystem. The Docker host has several configuration parameters. However, manually configuring Docker hosts is a slow and error-prone process. We will have problems scaling our Docker deployments in production if we don't have an automation strategy in place.

In this chapter, we will learn the concept of configuration management to solve this problem. We will use Chef, a configuration management software, to manage Docker hosts in scale. This chapter will cover the following topics:

- The importance of configuration management
- An introduction to Chef
- Automatically configuring Docker hosts
- Deploying Docker containers
- Alternative automation tools

### **An introduction to configuration management**

The Docker engine has several parameters to tune, such as cgroups, memory, CPU, filesystems, networking, and so on. Identifying which Docker containers run on which Docker hosts is another aspect of configuration. The Docker containers themselves need to be configured differently with cgroups settings, shared volumes, linked containers, public ports, and so on. Getting the combination of parameters to optimize our application will take time.

Replicating all the preceding configuration items to another Docker host is difficult to perform manually. We might not remember all the steps required to create a host, and it is an error-prone and slow process. Creating a "documentation" to get this process captured doesn't help either because such artifacts tend to get stale over time.

If we cannot provision new Docker hosts in a timely and reliable manner, we will have no space to scale out our Docker application. It is just as important to prepare and configure our Docker hosts in a consistent and fast manner. Otherwise, Docker's ability to create container packages for our application will become useless very fast.

Configuration management is a strategy to manage the changes happening in all aspects of our application, and it reports and audits the changes made to our system. This does not only apply when developing our application. For our case, it records all the changes to Docker hosts and the running of the Docker containers itself. Docker, in a sense, accomplishes the following aspects of configuration management for our application:

- Docker containers reproduce any environment for our application, from development to staging, testing, and production.
- Building Docker images is a simple way to make application changes and have them deployed to all environments.
- Docker enables all team members to get information about our application and make the needed changes to deliver the software efficiently to customers. By inspecting the `Dockerfile`, they can know which part of the application needs to be updated and what it needs in order to properly run.
- Docker tracks any change in our environment to a particular Docker image. Then, it traces it back to the corresponding version of the `Dockerfile`. It traces what the change is, who made it, and when it was made.

However, what about the Docker host running our application? Just as how a `Dockerfile` allows us to manage our application's environment in version control, configuration management tools can describe our Docker hosts in code. It simplifies the process to create Docker hosts. In the case of scaling out our Docker application, we can recreate a new Docker host from scratch easily. When there is a hardware failure, we can bring up new Docker hosts somewhere else from their known configuration. If we want to deploy a new version of our Docker containers, we can just update the Docker host's configuration code to point to the new image. Configuration management enables us to manage our Docker deployments in scale.

## Using Chef

Chef is a configuration management tool that provides a domain-specific language to model the configuration of our infrastructure. Each configuration item in our infrastructure is modeled as a resource. A resource is basically a Ruby method that accepts several parameters in a block. The following example resource describes installing the `docker-engine` package:

```
package 'docker-engine' do
  action :install
end
```

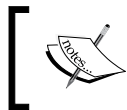
These resources are then written together in Ruby source files called recipes. When running a recipe against a server (a Docker host in our case), all the defined resources are executed to reach its desired state configuration.

Some Chef recipes may depend on other supplemental items, such as configuration templates and other recipes. All this information is gathered in cookbooks together with the recipes. A cookbook is the fundamental unit of distributing configuration and policy to our servers.

We will write Chef recipes to represent the desired state configuration of our Docker hosts. Our recipes will be organized in Chef cookbooks to distribute them to our infrastructure. However, first, let's prepare our Chef environment so that we can start describing our Docker-based infrastructure in recipes. A Chef environment consists of three things:

- A Chef server
- A workstation
- A node

The next few subsections will give you a detailed description of each component. Then, we will set them up to prepare our Chef environment to be able to manage our Docker host.



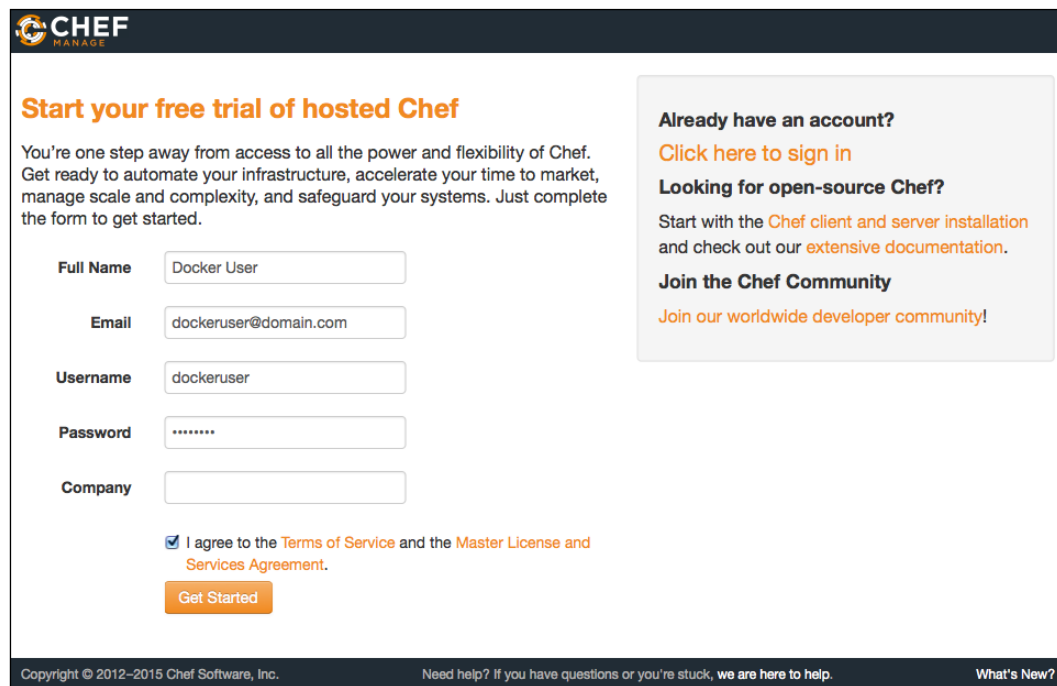
There are more details of setting up a Chef environment that are out of this chapter's scope. More information can be found at the Chef documentation website at <http://docs.chef.io>.

## Signing up for a Chef server

The Chef server is the central repository of cookbooks and other policy items governing our entire infrastructure. It contains metadata about the infrastructure that we are managing. In our case, the Chef server contains the cookbook, policy, and metadata on our Docker host.

To prepare a Chef server, we will simply sign up for a hosted Chef server. A free Chef server account allows us to manage up to five nodes in our infrastructure. Follow the next few steps to prepare a hosted Chef server account:

1. Go to <https://manage.chef.io/signup> and fill out the form for our account details as shown in the following screenshot:



The screenshot shows the 'Start your free trial of hosted Chef' page. The main heading is 'Start your free trial of hosted Chef' in orange. Below it, a paragraph states: 'You're one step away from access to all the power and flexibility of Chef. Get ready to automate your infrastructure, accelerate your time to market, manage scale and complexity, and safeguard your systems. Just complete the form to get started.' The form contains the following fields: 'Full Name' (with 'Docker User' entered), 'Email' (with 'dockeruser@domain.com' entered), 'Username' (with 'dockeruser' entered), 'Password' (with '\*\*\*\*\*' entered), and 'Company' (empty). Below the fields is a checkbox labeled 'I agree to the Terms of Service and the Master License and Services Agreement.' with a 'Get Started' button. To the right, a grey box contains links for 'Already have an account?' (Click here to sign in), 'Looking for open-source Chef?' (Start with the Chef client and server installation and check out our extensive documentation), and 'Join the Chef Community' (Join our worldwide developer community!). The footer contains copyright information (© 2012-2015 Chef Software, Inc.), a help link (Need help? If you have questions or you're stuck, we are here to help.), and a 'What's New?' link.

2. After creating a user account, the hosted Chef server will now prompt us to create an organization. Organizations are simply used to manage role-based access control for our Chef server. Create an organization by providing the details on the form and click on the **Create Organization** button:

**CHEF MANAGE**

## Create Organization

Full Name (example: Chef, Inc.)

Docker Org

Short Name (example: chef)

dockerorg

Cancel Create Organization

Copyright © 2012–2015 Chef Software, Inc. Need Feedback What's New? About Chef Manage

help? If you have questions or you're stuck, we're here to help.

3. We are now almost done getting our hosted Chef server account. Finally, click on **Download Starter Kit**. This will download a zip file containing our starter chef-repo. We will talk more about the chef-repo in the next section.

**CHEF MANAGE**

Nodes Reports Policy Administration dockerorg 0 0

> Organizations

- Create
- Reset Validation Key
- Generate Knife Config
- Invite User
- Leave Organization
- Starter Kit

Users

Groups

Global Permissions

## Thank you for choosing hosted Chef!

Follow these steps to be on your way to using hosted Chef

Download Starter Kit Learn Chef

### What's next?

**Chef Documentation**  
The best place to start learning about Chef in general.

**Browse Community Cookbooks**  
Hundreds of members of the Chef community have contributed cookbooks you can use or draw inspiration from.

**Contact Support**  
Our support team is here to provide assistance with any issues you may have while using hosted Chef. [Learn more about support for Chef.](#)

**More Resources**

- [Chat With Us](#)
- [Join the Mailing List](#)
- [Watch the Food Fight show](#)
- [Get Professional Training](#)

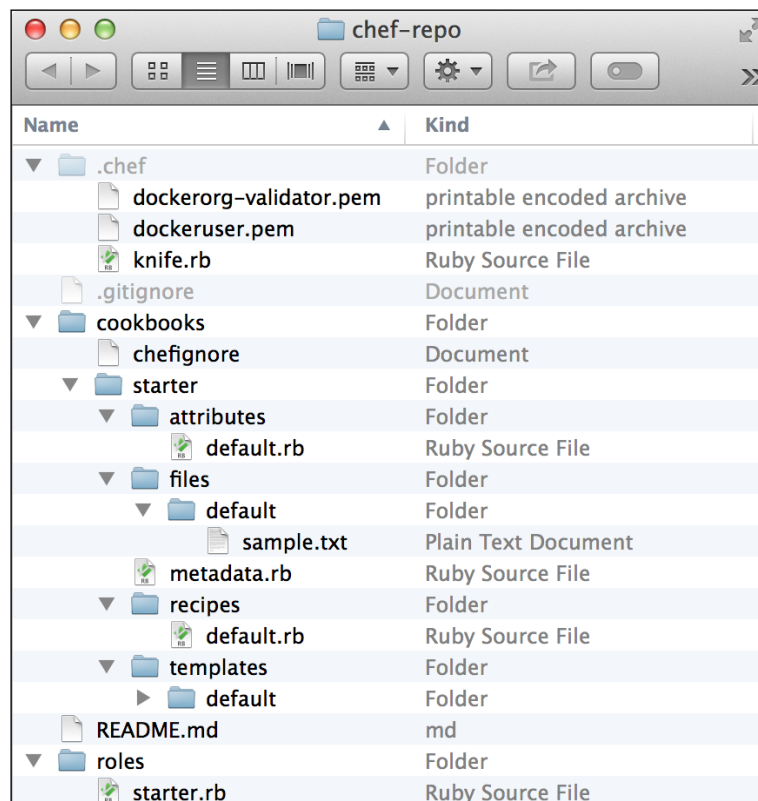
Copyright © 2012–2015 Chef Software, Inc. Need help? If you have questions or you're stuck, we're here to help. Feedback What's New? About Chef Manage

## Setting up our workstation

The second part of our Chef environment is the workstation. The workstation is used to interact with the Chef server. This is where we will do most of the preparation work and create the code to send to the Chef server. In our workstation, we will prepare the configuration items of our infrastructure in a Chef repository.

The Chef repository contains all the information needed to interact and synchronize with the Chef server. It contains the private key and other configuration files needed to authenticate and interact with the Chef server. These files will be found in the `.chef` directory of our Chef repository. It also contains the cookbooks that we will write and synchronize later with the Chef server in the `cookbooks/` directory. There are other files and directories inside a Chef repository, such as data bags, roles, and environments as well. However, it is enough for now to know about the cookbooks and authentication files to be able to configure our Docker host.

Do you remember that starter kit we downloaded in the previous section? Unzip this file to extract our `chef-repo`. We should have the following files described in the directory tree:



Another important component in our workstation is the Chef development kit. It contains all the programs needed to read all the configuration in our chef-repo and interact with the Chef server. Convenient programs to create, develop, and test our cookbooks are also available in the Chef development kit. We will use various programs in the development kit throughout the rest of this chapter.

Now, let's download the Chef development kit from <https://downloads.chef.io/chef-dk> according to our workstation's platform.



Next, open the downloaded installer. Install the Chef development kit according to the prompts from our platform. Finally, confirm that the installation was successful with the following commands:

```
$ chef -v
Chef Development Kit Version: 0.6.2
chef-client version: 12.3.0
berks version: 3.2.4
kitchen version: 1.4.0
```

Now that we have set up our workstation, let's go to our chef-repo/ directory to prepare the last component of our Chef environment.



## Bootstrap nodes

The last part of our Chef environment is nodes. A node is any computer that is managed by Chef. It can be a physical machine, virtual machine, a server in the cloud, or a networking device. In our case, our Docker host is a node.

The central component for any node to be managed by Chef is the chef-client. It connects to the Chef server to download the necessary files to bring our node to its configuration state. When a chef-client is run on our node, it performs the following steps:

1. It registers and authenticates the node with the Chef server.
2. It gathers system information in our node to create a node object.
3. Then, it synchronizes the Chef cookbooks needed by our node.
4. It compiles the resources by loading our node's needed recipes.
5. Next, it executes all the resources and performs the corresponding actions to configure our node.
6. Finally, it reports the result of the chef-client run back to the Chef server and other configured notification endpoints.

Now, let's prepare our Docker host as a node by bootstrapping it from our workstation. The bootstrapping process installs and configures the chef-client. Run the following command to get this bootstrap process started:

```
$ knife bootstrap dockerhost
...
Connecting to dockerhost
dockerhost Installing Chef Client...
...
dockerhost trying wget...
dockerhost Comparing checksum with sha256sum...
dockerhost Installing Chef 12.3.0
dockerhost installing with dpkg...
...
```

```
dockerhost Thank you for installing Chef!
dockerhost Starting first Chef Client run...
dockerhost Starting Chef Client, version 12.3.0
dockerhost Creating a new client identity for dockerhost using the
validator key.
dockerhost resolving cookbooks for run list: []
dockerhost Synchronizing Cookbooks:
dockerhost Compiling Cookbooks...
dockerhost ... WARN: Node dockerhost has an empty run list.
dockerhost Converging 0 resources
dockerhost
dockerhost Running handlers:
dockerhost Running handlers complete
dockerhost Chef Client finished, 0/0 resources updated in 12.78s
```

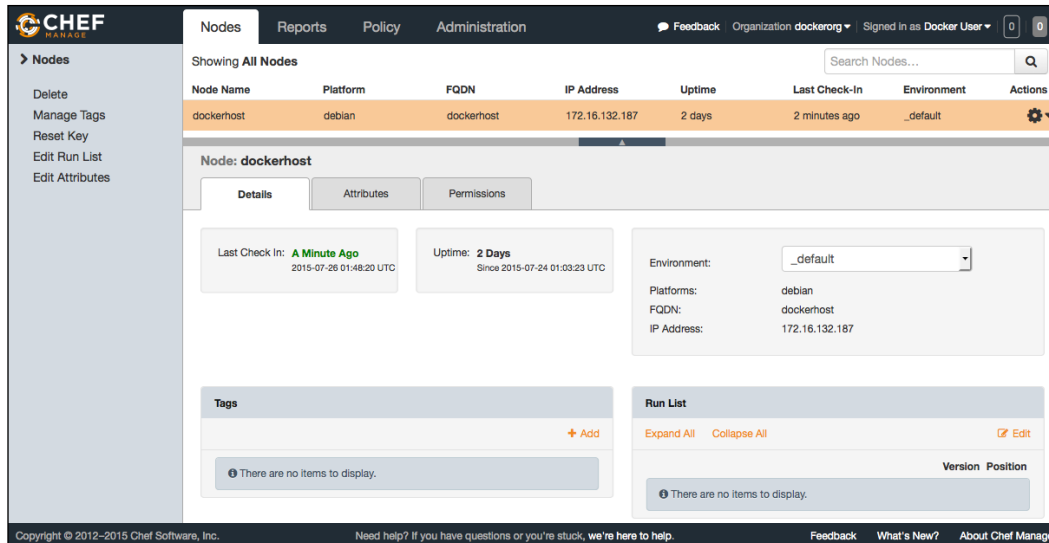
As we can note in the preceding command, the bootstrapping process did two things. First, it installed and configured the chef-client on our Docker host node. Next, it started the chef-client to synchronize its desired state with our Chef server. As we haven't assigned any designed state yet to our Docker host, it didn't do anything.



We can customize this bootstrap process according to our needs. More information on how to use `knife bootstrap` can be found at [http://docs.chef.io/knife\\_bootstrap.html](http://docs.chef.io/knife_bootstrap.html).

In some cases, cloud providers have a deep Chef integration already out of the box. So, instead of `knife bootstrap`, we will just use the cloud provider's SDK. There, we just need to specify that we want to have Chef integrated. We will provide it with the information, such as the chef-client's `client.rb` configuration and validation keys' credentials.

Our Docker host is now properly registered to the Chef server, ready to grab its configuration. Go to <https://manage.chef.io/organizations/dockerorg/nodes/dockerhost> to check our Docker host as a node in our Chef environment, as shown in the following screenshot:



## Configuring the Docker host

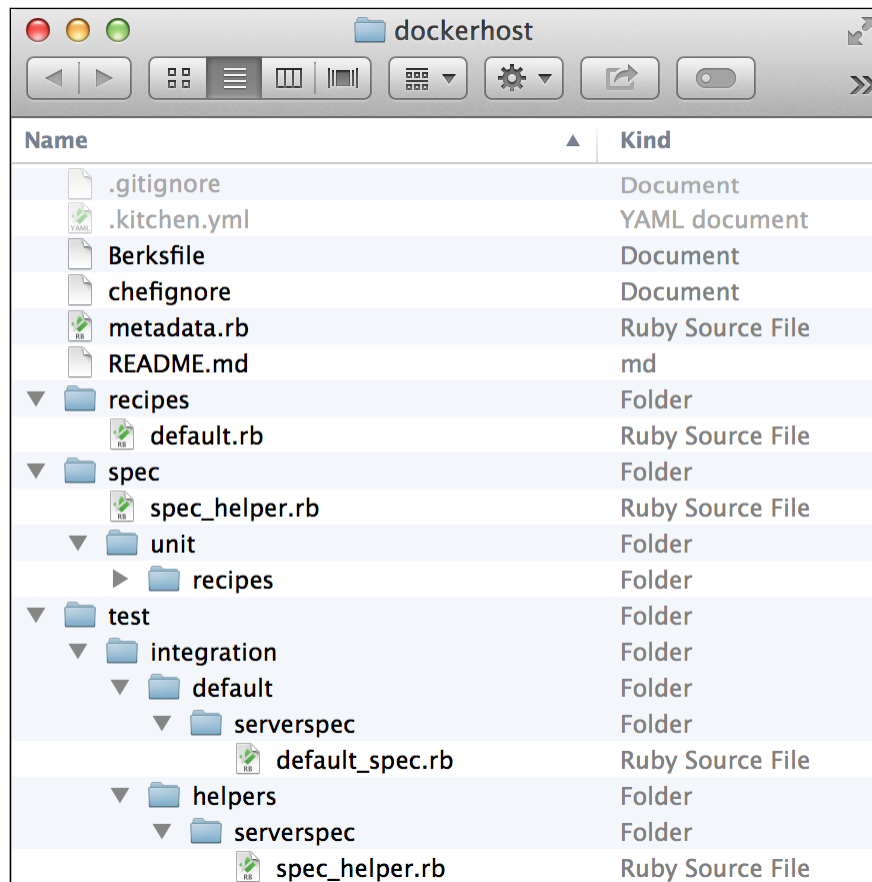
Now that we have all the components of our Chef environment properly set up, we can now start writing Chef recipes to actually describe what configuration our Docker host should have. In addition, we will leapfrog our productivity by taking advantage of existing Chef cookbooks in the Chef ecosystem. As Docker is a popular infrastructure stack to deploy, we can use cookbooks in the wild that allow us to configure our Docker hosts. Chef cookbooks provided by the community can be found in the Chef supermarket. We can go to <http://supermarket.chef.io> to discover other cookbooks that we can readily use.

In this section, you will learn how to write Chef recipes and apply it to our node. Follow the next few steps to write the recipe for our Docker host:

1. Use the Chef development kit's `chef generate cookbook` command to generate a boilerplate for our cookbook. After entering the cookbooks directory, issue the following command:

```
$ cd cookbooks
$ chef generate cookbook dockerhost
```

The boilerplate cookbook directory structure will look similar to the following screenshot:



- Next, we will prepare to edit our cookbook. Change our working directory to the cookbook we created earlier using the following command:

```
$ cd dockerhost
```

- Install the following cookbooks from the Chef supermarket as dependencies: `apt` and `docker`. These cookbooks provide additional resource definitions that can be used in our recipes. We will use them later as building blocks to set up our Docker host. To add the dependencies, update the `metadata.rb` file, as follows:

```
name 'dockerhost'
maintainer 'The Authors'
maintainer_email 'you@example.com'
```

```
license 'all_rights'
description 'Installs/Configures dockerhost'
long_description 'Installs/Configures dockerhost'
version '0.1.0'

depends 'apt', '~> 2.7.0'
depends 'docker', '~> 0.40.3'
```



The `metadata.rb` file provides metadata about our Chef cookbooks. The information in the metadata provides hints to the Chef server so that the cookbook can be properly deployed to our nodes. For more information on how to configure metadata to our Chef cookbooks, visit [http://docs.chef.io/config\\_rb\\_metadata.html](http://docs.chef.io/config_rb_metadata.html).

4. Now that we have our dependencies declared, we can download them by issuing the following command:

```
$ berks install

Resolving cookbook dependencies...
Fetching 'dockerhost' from source at .
Fetching cookbook index from https://supermarket.chef.io...
Installing apt (2.7.0)
Installing docker (0.40.3)
Using dockerhost (0.1.0) from source at .
```

5. Finally, we will write the Chef recipe equivalent to the installation instructions found at <http://blog.docker.com/2015/07/new-apt-and-yum-repos>. We will use the `apt_repository` resource provided by the `apt` dependency cookbook we added earlier. Then, add the following lines to `recipes/default.rb`:

```
apt_repository 'docker' do
  uri 'http://apt.dockerproject.org/repo'
  components %w(debian-jessie main)
  keyserver 'p80.pool.sks-keyservers.net'
  key '58118E89F3A912897C070ADB76221572C52609D'
  cache_rebuild true
end

package 'docker-engine'
```

Now, we are done preparing our `dockerhost` / Chef cookbook. The final step is to apply it to our Docker host so that it can pick its desired configuration. Follow the next remaining steps to perform this:

1. First, upload the Chef cookbook to our Chef server. Note that in the output of the following command, the `apt` and `docker` cookbooks that we depend on will also be automatically uploaded:

```
$ berks upload
Uploaded apt (2.7.0) to: 'https://api.opscode.../dockerorg'
Uploaded docker (0.40.3) to: 'https://api.ops.../dockerorg'
Uploaded dockerhost (0.1.0) to: 'https://api.opscode.com:443/
organizations/dockerorg'
```

2. Next, apply the `dockerhost` recipe we wrote earlier to the node (that is, the Docker host) by setting its `run_list` via the following command:

```
$ knife node run_list set dockerhost dockerhost
dockerhost:
  run_list: recipe[dockerhost]
```

3. Finally, run the `chef-client` in `dockerhost`. The `chef-client` will fetch the Docker host's node object and apply the desired state configuration we applied in the previous steps, as follows:

```
$ ssh dockerhost
dockerhost$ sudo chef-client
Starting Chef Client, version 12.3.0
resolving cookbooks for run list: ["dockerhost"]
Synchronizing Cookbooks:
- apt
- dockerhost
- docker
Compiling Cookbooks...
Converging 2 resources
Recipe: dockerhost::default
  * apt_repository[docker] action add
    * execute[install-key 58118E89F3A912897C...] action run
    ...
  * apt_package[docker-engine] action install
    - install version 1.7.1-0~j... of package docker-engine
```

**Running handlers:**

**Running handlers complete**

**Chef Cl... finished, 6/7 resources updated in 24.69 seconds**

Now, we have Docker installed and configured in our Docker host using Chef. Whenever we need to add another Docker host, we can just create another server in our cloud provider and bootstrap it to have the `dockerhost` Chef recipe written earlier. When we want to update how the Docker daemons are configured in all our Docker hosts, we will just update the Chef cookbook and rerun the `chef-client`.



In a production environment, the goal of having configuration management software installed in our Docker host is to never have the need to log in to it just to perform configuration updates. Running the `chef-client` manually is only half the automation.

We will want to run the `chef-client` as a daemon process so that we don't have to run it every time we perform an update. The `chef-client` daemon will poll the Chef server to check whether there are any updates to the node it is managing. By default, this polling interval is set to 30 minutes.

For more information on how to configure the `chef-client` as a daemon, refer to the Chef documentation at [https://docs.chef.io/chef\\_client.html](https://docs.chef.io/chef_client.html).

## Deploying Docker containers

The next step to manage Docker in scale is to deploy Docker containers to our pool of Docker hosts automatically. By now, we have built a few Docker applications already. We have a rough architectural sketch of how these containers communicate with and consume each other. Chef recipes can be used to represent this architectural topology in code, which is essential to managing our whole application and infrastructure in scale. We can identify which Docker containers need to run and know how each container connects with other containers. We can locate where our Docker containers should be deployed. Having the whole architecture in code allows us to place an orchestration strategy for our application.

In this section, we will create a Chef recipe to orchestrate the deployment of the Nginx Docker image to our Docker host. We will use the Chef resources provided by the `docker` cookbook we added in the previous section to configure our Docker host. Follow the next few steps to perform the deployment:

1. To begin, create the Chef recipe that we will work on. The following command will create the `recipes/containers.rb` recipe file in our `dockerhost/` cookbook:
2. Next, pull the official Nginx Docker image at [https://registry.hub.docker.com/\\_/nginx](https://registry.hub.docker.com/_/nginx) to our Docker host. Write the following code in `recipes/containers.rb`:
3. After downloading the Docker image, configure the Docker host to run the container. As of version 0.40.3 of the `docker` cookbook, we need to specify that our Debian Jessie Docker host deployment uses `systemd` as its init system. Add the following lines to the `recipes/containers.rb` as well:

```
$ chef generate recipe . containers
```

```
docker_image 'nginx' do
  tag '1.9.3'
end
```

```
node.set['docker']['container_init_type'] = 'systemd'
```

```
directory '/usr/lib/systemd/system'
```

```
docker_container 'nginx' do
  tag '1.9.3'
  container_name 'webserver'
  detach true
  port '80:80'
end
```



The `docker_container` and `docker_image` has several other options that we can tune to specify what we want to do with our container. The `docker` cookbook also has other resources to work with our Docker hosts. More information on options and further usage can be found on its project page at <https://github.com/bflad/chef-docker>.

4. Next, we will prepare the new version of our cookbook for release. Bump the version information in `metadata.rb` to do this, as follows:

```
name 'dockerhost'
maintainer 'The Authors'
```



```
maintainer_email 'you@example.com'
license 'all_rights'
description 'Installs/Configures dockerhost'
long_description 'Installs/Configures dockerhost'
version '0.2.0'

depends 'apt', '~> 2.7.0'
depends 'docker', '~> 0.40.3'
```

5. Update the `Berksfile.lock` file to pin the versions of all the cookbooks we will upload to the Chef server in the next step. Type the following command to perform the update:

```
$ berks install
Resolving cookbook dependencies...
Fetching 'dockerhost' from source at .
Fetching cookbook index from https://supermarket.chef.io...
Using dockerhost (0.2.0) from source at .
Using apt (2.7.0)
Using docker (0.40.3)
```

6. Now that all the artifacts for our new cookbook are ready, we will type the following command to upload the updated cookbook to our Chef server. Note how the berks upload command automatically recognizes that only the `dockerhost` cookbook needs updating as it skips uploading the `apt` and `docker` cookbooks:

```
$ berks upload
Skipping apt (2.7.0) (frozen)

Skipping docker (0.40.3) (frozen)
Uploaded dockerhost (0.2.0) to: 'https://ap.../dockerorg'
```

7. Next, add the `recipes/containers.rb` to the run list of our Docker host. Type the following command to update the node representing our Docker host:

```
$ knife node run_list add dockerhost dockerhost::containers
dockerhost:
  run_list:
    recipe[dockerhost]
    recipe[dockerhost::containers]
```

8. Finally, rerun the chef-client to pick up the new configuration for our Docker host. We can also wait for the rerun of the chef-client if we configure the chef-client to run as a daemon process. Execute the following command:

```
$ ssh dockerhost
dockerhost$ sudo chef-client
Starting Chef Client, version 12.3.0
resolving cookbooks for run list: ["dockerhost",
"dockerhost::containers"]
Synchronizing Cookbooks:
  - dockerhost
  - apt
  - docker
Compiling Cookbooks...
Converging 5 resources
Recipe: dockerhost::default
...
Recipe: dockerhost::containers
  * docker_image[nginx] action pull

  * directory[/usr/lib/systemd/system] action create
    - create new directory /usr/lib/systemd/system
  * docker_container[nginx] action run
    * template[/usr/lib/.../webserver.socket] action create
      ...
    * service[webserver] action enable (up to date)
    * service[webserver] action start
      - start service service[webserver]
  * template[webserver.socket] action nothing ...
  * template[webserver.service] action nothing ...
  * service[webserver] action nothing ...

Running handlers:
Running handlers complete
Chef Client finished, 6/10 resources updated in 42.83 seconds
```

We now have our Docker host running the nginx Docker container. We can confirm that it is working by going to `http://dockerhost`. We should be able to get the following page in the screenshot:



## Alternative methods

There are other general-purpose configuration management tools that allow us to configure our Docker host. The following is a short list of the other tools that we can use:

- **Puppet:** Refer to `http://puppetlabs.com`.
- **Ansible:** This can be found at `http://ansible.com`.
- **CFEngine:** This is available at `http://cfengine.com`.
- **SaltStack:** You can find more on this at `http://saltstack.com`.
- **The Docker machine:** This is a very specific configuration management tool that allows us to provision and configure Docker hosts in our infrastructure. More information about Docker machine can be found on the Docker documentation page at `https://docs.docker.com/machine`.

If we do not want to manage our Docker host infrastructure at all, we can use Docker hosting services. Popular cloud providers started offering Docker hosts as a preprovisioned cloud image that we can use. Others offer a more comprehensive solution that allows us to interact with all the Docker hosts in the cloud as a single virtual Docker host. The following is a list of links of the popular cloud providers describing their integration with the Docker ecosystem:

- Google Container Engine (<https://cloud.google.com/container-engine>)
- Amazon EC2 Container Service (<http://aws.amazon.com/documentation/ecs>)
- Azure Docker VM Extension (<https://github.com/Azure/azure-docker-extension>)
- Joyent Elastic Container Service (<https://www.joyent.com/public-cloud>)

In terms of deploying Docker containers, there are several container tools that allow you to do this. They provide APIs to run and deploy our Docker containers. Some of the offered APIs are even compatible with the Docker engine itself. This allows us to interact with our pool of Docker hosts as if it is a single virtual Docker host. The following is a list of a few tools that allow us to orchestrate the deployment of our containers to a pool of Docker hosts:

- Docker Swarm (<https://www.docker.com/docker-swarm>)
- Google Kubernetes (<http://kubernetes.io>)
- CoreOS fleet (<https://coreos.com/fleet>)
- Mesosphere Marathon (<https://mesosphere.github.io/marathon>)
- SmartDataCenter Docker Engine (<https://github.com/joyent/sdc-docker>)

However, we still need configuration management tools such as Chef to deploy and configure our orchestration systems at the top of our pool of Docker hosts.

## Summary

In this chapter, we learned how to automate the configuration of our Docker deployments. Using Chef allows us to configure and provision multiple Docker hosts in scale. It also enabled us to deploy and orchestrate Docker containers for our application to our pool of Docker hosts. From this point on, you can write Chef recipes to persist all the Docker optimization techniques you will learn in this book.

In the next chapter, we will introduce instrumentation to monitor our whole Docker infrastructure and application. This will give us further feedback on how to better optimize our Docker deployments for higher performance.



# 4

## Monitoring Docker Hosts and Containers

We now know some ways to optimize our Docker deployments. We also know how to scale to improve performance. But how do we know that our tuning assumptions were correct? Being able to monitor our Docker infrastructure and application is important to figure out why and when we need to optimize. Measuring how our system is performing allows us to identify its limits to scale and tune accordingly.

In addition to monitoring low-level information about Docker, it is also important to measure the business-related performance of our application. By tracing the value stream of our application, we can correlate business-related metrics to system-level ones. With this, our Docker development and operations teams can show their business colleagues how Docker saves their organization's costs and increases business value.

In this chapter, we will cover the following topics about being able to monitor our Docker infrastructure and applications at scale:

- The importance of monitoring
- Collecting monitored data in Graphite
- Monitoring Docker with collectd
- Consolidating logs in an ELK stack
- Sending logs from Docker

## The importance of monitoring

Monitoring is important as it provides a source of feedback on the Docker deployment that we built. It answers several questions about our application from low-level operating system performance to high-level business targets. Having proper instrumentation inserted in our Docker hosts allows us to identify our system's state. We can use this source of feedback to identify whether our application is behaving as originally planned.

If our initial hypothesis was incorrect, we can use the feedback data to revise our plan and change our system accordingly by tuning our Docker host and containers or updating our running Docker application. We can also use the same monitoring process to identify errors and bugs after our system is deployed to production.

Docker has built-in features to log and monitor. By default, a Docker host stores a Docker container's standard output and error streams to JSON files in `/var/lib/docker/<container_id>/<container_id>-json.log`. The `docker logs` command asks the Docker engine daemon to read the content of the files here.

Another monitoring facility is the `docker stats` command. This queries the Docker engine's remote API's `/containers/<container_id>/stats` endpoint to report runtime statistics about the running container's control group regarding its CPU, memory, and network usage. The following is an example output of the `docker stats` command reporting the said metrics:

```
dockerhost$ docker run --name running -d busybox \
    /bin/sh -c 'while true; do echo hello && sleep 1; done'
dockerhost$ docker stats running
```

CONTAINER	CPU %	MEM USAGE/LIMIT	MEM %	NET I/O
running	0.00%	0 B/518.5 MB	0.00%	17.06 MB/119.8 kB

The built-in `docker logs` and `docker stats` commands work well to monitor our Docker applications for development and small-scale deployments. When we get to a point in our production-grade Docker deployment where we manage tens, hundreds, or even thousands of Docker hosts, this approach will no longer be scalable. It is not feasible to log in to each of our thousand Docker hosts and type `docker logs` and `docker stats`.

Doing this one by one also makes it difficult to create a more holistic picture of our entire Docker deployment. Also, not everyone interested in our Docker application's performance can log in to our Docker hosts. Our colleagues dealing with only the business aspect of our application may want to ask certain questions on how our application's deployment in Docker improves what our organization wants. However, they do not necessarily want to learn how to log in and start typing Docker commands in our infrastructure.

Hence, it is important to be able to consolidate all of the events and metrics from our Docker deployments into a centralized monitoring infrastructure. It allows our operations to scale by having a single point to ask what is happening to our system. A centralized dashboard also enables people outside our development and operations team, such as our business colleagues, to have access to the feedback provided by our monitoring system. The remaining sections will show you how to consolidate messages from `docker logs` and collect statistics from data sources such as `docker stats`.

## Collecting metrics to Graphite

To begin monitoring our Docker deployments, we must first set up an endpoint to send our monitored values to. Graphite is a popular stack for collecting various metrics. Its plaintext protocol is very popular because of its simplicity. Many third-party tools understand this simple protocol. Later, we will show you how easy it is to send data to Graphite after we finish setting it up.

Another feature of graphite is that it can render the data it gathers into graphs. We can then consolidate these graphs to build a dashboard. The dashboard we crafted in the end will show the various kinds of information that we need to monitor our Docker application.

In this section, we will set up the following components of Graphite to create a minimal stack:

- **carbon-cache:** This is the Graphite component that receives metrics over the network. This implements the simple plaintext protocol described earlier. It can also listen to a binary-based protocol called the pickle protocol, which is a more advanced but smaller and optimized format to receive metrics.
- **whisper:** This is a file-based bounded time series database in which the carbon-cache persists the metrics it receives. Its bounded or fixed-size nature makes it an ideal solution for monitoring. Over time, the metrics that we monitor will accumulate. Hence, the size of our database will just keep increasing so that you will need to monitor it as well! However, in practice we are mostly interested in monitoring our application until a fixed point. Given this assumption, we can plan the resource requirements of our whisper database beforehand and not think about it this much as the operation of our Docker application continues.
- **graphite-web:** This reads the whisper database to render graphs and dashboards. It is also optimized to create such visualizations in real time by querying carbon-cache endpoints to display data that is yet to be persistent in the whisper database as well.





There are other components in carbon, such as carbon-aggregator and carbon-relay. These are the components that are needed in order to scale out Graphite effectively as the number of metrics you measure grows. More information about these components can be found at <https://github.com/graphite-project/carbon>. For now, we will focus on deploying just the carbon-cache to create a simple Graphite cluster.

The next few steps describe how to deploy the carbon-cache and a whisper database:

1. First, prepare a Docker image for carbon to dogfood our Docker host deployments. Create the following Dockerfile to prepare this image:

```
FROM debian:jessie

RUN apt-get update && \
    apt-get --no-install-recommends \
        install -y graphite-carbon

ENV GRAPHITE_ROOT /graphite

ADD carbon.conf /graphite/conf/carbon.conf

RUN mkdir -p $GRAPHITE_ROOT/conf && \
    mkdir -p $GRAPHITE_ROOT/storage && \
    touch $GRAPHITE_ROOT/conf/storage-aggregation.conf && \
    touch $GRAPHITE_ROOT/conf/storage-schemas.conf

VOLUME /whisper
EXPOSE 2003 2004 7002

ENTRYPOINT ["/usr/bin/twistd", "--nodaemon", \
    "--reactor=epoll", "--no_save"]
CMD ["carbon-cache"]
```

2. Next, build the Dockerfile we created earlier as the `hubuser/carbon` image, as follows:

```
dockerhost$ docker build -t hubuser/carbon .
```

3. Inside the `carbon.conf` configuration file, we will configure the carbon-cache to use the Docker volume `/whisper` as the whisper database. The following is the content describing this setting:

```
[cache]

CARBON_METRIC_INTERVAL = 0
LOCAL_DATA_DIR = /whisper
```

4. After building the `hubuser/carbon` image, we will prepare a whisper database by creating a data container. Type the following command to accomplish this:

```
dockerhost$ docker create --name whisper \
    --entrypoint='whisper database for graphite' \
    hubuser/carbon
```

5. Finally, run the carbon-cache endpoint attached to the data container we created earlier. We will use custom container names and publicly exposed ports so that we can send and read metrics from it, as follows:

```
dockerhost$ docker run --volumes-from whisper -p 2003:2003 \
    --name=carboncache hubuser/carbon
```

We now have a place to send all our Docker-related metrics that we will gather later. In order to make use of the metrics we stored, we need a way to read and visualize them. We will now deploy `graphite-web` to visualize what is going on with our Docker containers. The following are the steps to deploy `graphite-web` in our Docker hosts:

1. Build `Dockerfile` as `hubuser/graphite-web` to prepare a Docker image to deploy `graphite-web` via the following code:

```
FROM debian:jessie

RUN apt-get update && \
    apt-get --no-install-recommends install -y \
        graphite-web \
        apache2 \
        libapache2-mod-wsgi

ADD local_settings.py /etc/graphite/local_settings.py
RUN ln -sf /usr/share/graphite-web/apache2-graphite.conf \
    /etc/apache2/sites-available/100-graphite.conf && \
    a2dissite 000-default && a2ensite 100-graphite && \
    mkdir -p /graphite/storage && \
    graphite-manage syncdb --noinput && \
    chown -R _graphite:_graphite /graphite

EXPOSE 80
ENTRYPOINT ["apachectl", "-DFOREGROUND"]
```

2. The preceding Docker image refers to `local_settings.py` to configure `graphite-web`. Place the following annotations to link the carbon-cache container and whisper volume:

```
import os
# --link-from carboncache:carbon
```

```
CARBONLINK_HOSTS = ['carbon:7002']
# --volumes-from whisper
WHISPER_DIR = '/whisper'


GRAPHITE_ROOT = '/graphite'
SECRET_KEY = os.environ.get('SECRET_KEY', 'replacekey')
LOG_RENDERING_PERFORMANCE = False
LOG_CACHE_PERFORMANCE = False
LOG_METRIC_ACCESS = False
LOG_DIR = '/var/log/graphite'
```

3. After preparing the Dockerfile and local\_settings.py configuration files, build the hubuser/graphite-web Docker image with the following command:

```
dockerhost$ docker build -t hubuser/graphite-web .
```

4. Finally, run the hubuser/graphite-web Docker image linked with the carbon-cache container and whisper volume by typing the following command:

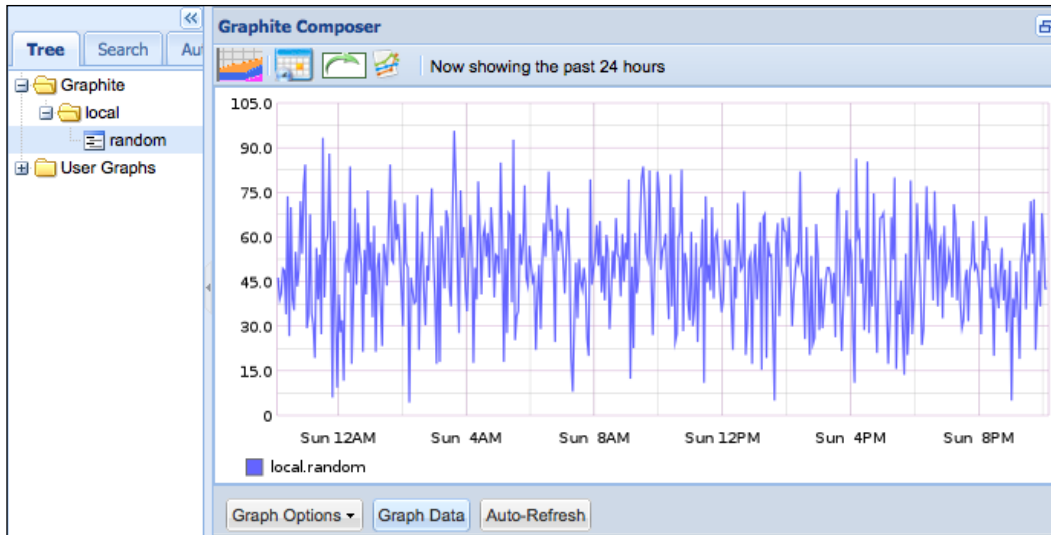
```
dockerhost$ docker run --rm --env SECRET_KEY=somestring \
--volumes-from whisper --link carboncache:carbon \
-p 80:80 hubuser/graphite-web
```

 The SECRET\_KEY environment variable is a necessary component to group together multiple graphite-web instances when you decide to scale out. More graphite-web settings can be found at <http://graphite.readthedocs.org/en/latest/config-local-settings.html>.

Now that we have a complete Graphite deployment, we can run some preliminary tests to see it in action. We will test this by populating the whisper database with random data. Type the following command to send random metrics called `local.random` to the carbon-cache endpoint:

```
dockerhost$ seq `date +%s` -60 $((`date +%s` - 24*60*60)) \
| perl -n -e \
'print "local.random ". int(rand(100)) . " " . $_' \
| docker run --link carboncache:carbon -i --rm \
busybox nc carbon 2003
```

Finally, confirm that the data is persistent by visiting our `hubuser/graphite-web`'s composer URL `http://dockerhost/compose`. Go to the **Tree** tab and then expand the `Graphite/local` folder to get the `random` metric. The following is a graph that we will see on our `graphite-web` deployment:



## Graphite in production

In production, this simple Graphite setup will reach its limits as we monitor more and more metrics in our Docker deployment. We need to scale this out in order to keep up with the increased number of metrics that we monitor. With this, you need to deploy Graphite with a cluster setup.

To scale out the metric processing capacity of `carbon-cache`, we need to augment it with a `carbon-relay` and `carbon-aggregator`. For `graphite-web` to be more responsive, we need to scale it out horizontally along with other caching components, such as `memcached`. We also need to add another `graphite-web` instance that connects to other `graphite-web` instances to create a unified view of all the metrics. The `whisper` databases will be co-located with a `carbon-cache` and `graphite-web`, so it will scale-out naturally along with them.



More information on how to scale out a Graphite cluster in production is found at <http://graphite.readthedocs.org>.

## Monitoring with collectd

We finished setting up a place to send all our Docker-related data to. Now, it is time to actually fetch all the data related to our Docker applications. In this section, we will use `collectd`, a popular system statistics collection daemon. It is a very lightweight and high-performance C program. This makes it a noninvasive monitoring software because it doesn't consume many resources from the system it monitors. Being lightweight, it is very simple to deploy as it requires minimum dependencies. It has a wide variety of plugins to monitor almost every component of our system.

Let's begin monitoring our Docker host. Follow the next few steps to install `collectd` and send the metrics to our Graphite deployment:

1. First, install `collectd` in our Docker host by typing the following command:

```
dockerhost$ apt-get install collectd-core
```

2. Next, create a minimum `collectd` configuration to send data to our Graphite deployment. You may recall from before that we exposed the carbon-cache's default plaintext protocol port (2003). Write the following configuration entry in `/etc/collectd/collectd.conf` to set this up:

```
LoadPlugin "write_graphite"
```

```
<Plugin write_graphite>
  <Node "carboncache">
    Host "dockerhost"
  </Node>
</Plugin>
```

3. Now, it is time to measure a few things from our Docker host. Load the following `collectd` plugins by placing the next few lines in `/etc/collectd/collectd.conf`:

```
LoadPlugin "cpu"
LoadPlugin "memory"
LoadPlugin "disk"
LoadPlugin "interface"
```

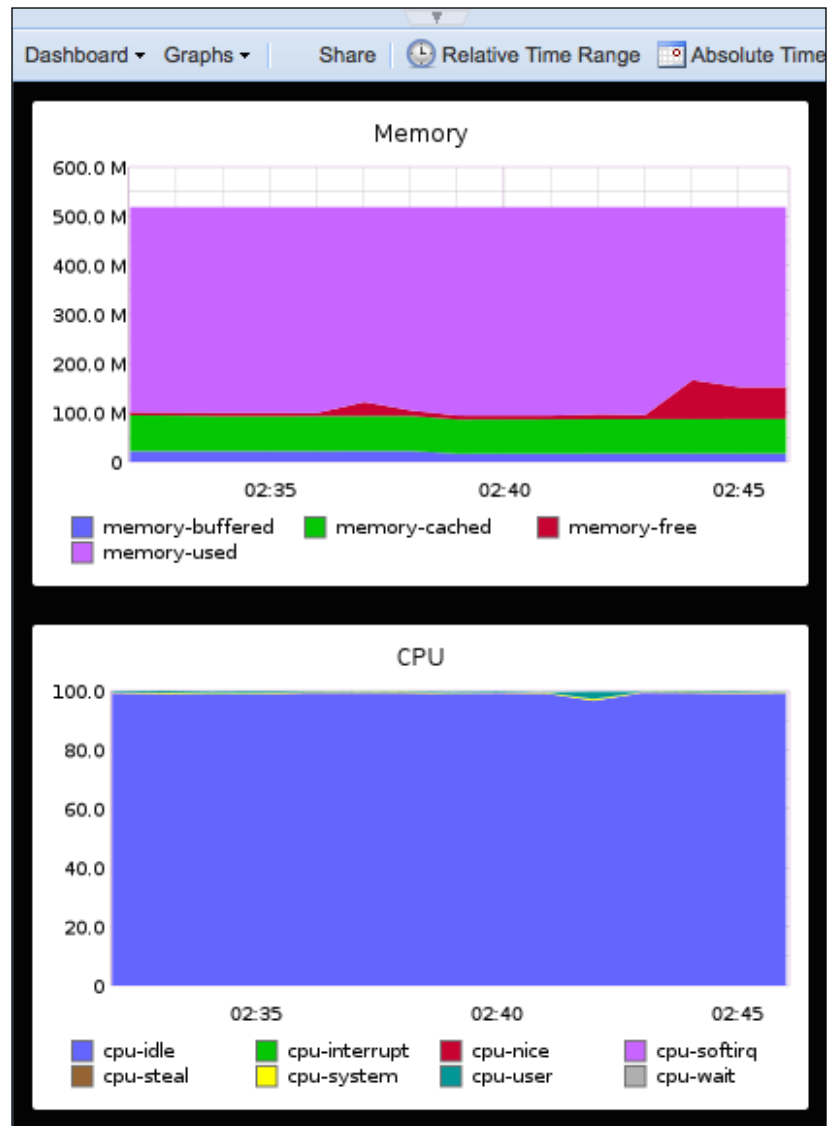
4. After finishing the configuration, restart `collectd` by typing the following command:

```
dockerhost$ systemctl restart collectd.service
```

5. Finally, let's create a visualization dashboard in our graphite-web deployment to look at the preceding metrics. Go to `http://dockerhost/dashboard`, click on **Dashboard**, and then on the **Edit Dashboard** link. It will prompt us with a text area to place a dashboard definition. Paste the following JSON text in this text area to create our preliminary dashboard:

```
[
  {
    "areaMode": "stacked",
    "yMin": "0",
    "target": [
      "aliasByMetric(dockerhost.memory.*) "
    ],
    "title": "Memory"
  },
  {
    "areaMode": "stacked",
    "yMin": "0",
    "target": [
      "aliasByMetric(dockerhost.cpu-0.*) "
    ],
    "title": "CPU"
  }
]
```

We now have a basic monitoring stack for our Docker host. The last step in the previous section will show a dashboard that looks something similar to the following screenshot:



## Collecting Docker-related data

Now, we will measure some basic metrics that govern the performance of our application. But how do we drill down to further details on the containers running in our Docker hosts? Inside our Debian Jessie Docker host, our containers run under the `docker-[container_id].scope` control group. This information is found at `/sys/fs/cgroup/cpu,cpuacct/system.slice` in our Docker host's sysfs. Fortunately, `collectd` has a `cgroups` plugin that interfaces with the sysfs information exposed earlier. The next few steps will show you how to use this plugin to measure the CPU performance of our running Docker containers:

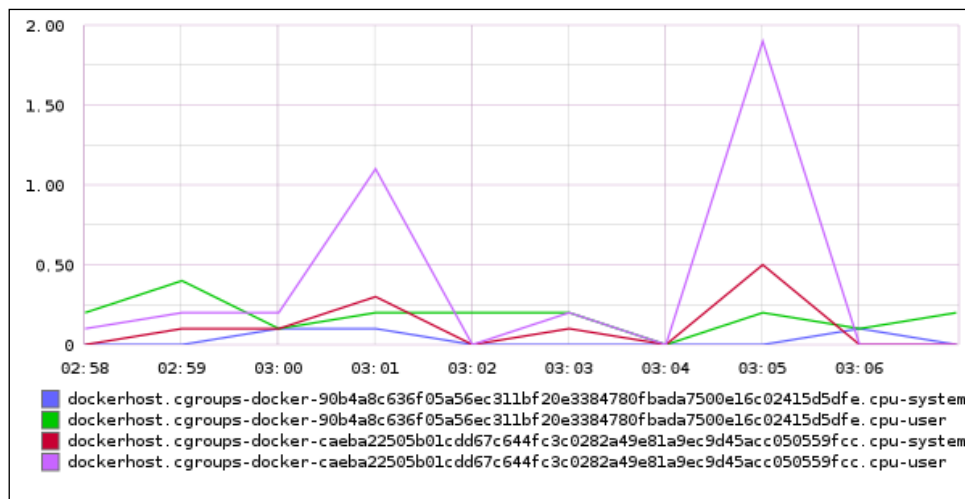
1. First, insert the following lines in `/etc/collectd/collectd.conf`:  

```
LoadPlugin "cgroups"

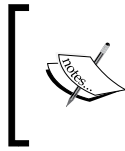
<Plugin cgroups>
    CGroup "/^docker.*.scope/"
</Plugin>
```
2. Next, restart `collectd` by typing the following command:  

```
dockerhost$ systemctl restart collectd.service
```
3. Finally, wait for a few minutes for Graphite to receive enough metrics from `collectd` so that we can get an initial feel to visualize our Docker container's CPU metrics.

We can now check the CPU metrics of our Docker containers by querying for the `dockerhost.cgroups-docker*. *` metrics on our Graphite deployment's render API. The following is the image produced by the render API URL `http://dockerhost/render/?target=dockerhost.cgroups-docker-*. *`:







More information about the `cgroups` plugin can be found in the `collectd` documentation page at [https://collectd.org/documentation/manpages/collectd.conf.5.shtml#plugin\\_cgrou](https://collectd.org/documentation/manpages/collectd.conf.5.shtml#plugin_cgrou)s.

Currently, the `cgroups` plugin only measures the CPU metrics of our running Docker containers. There is some work in progress, but it is not yet ready at the time of this book's writing. Fortunately, there is a Python-based `collectd` plugin that interfaces itself to `docker stats`. The following are the steps needed to set up this plugin:

1. First, download the following dependencies to be able to run the plugin:  
`dockerhost$ apt-get install python-pip libpython2.7`
2. Next, download and install the plugin from its GitHub page:  
`dockerhost$ cd /opt`  
`dockerhost$ git clone https://github.com/lebauce/docker-collectd-plugin.git`  
`dockerhost$ cd docker-collectd-plugin`  
`dockerhost$ pip install -r requirements.txt`
3. Add the following lines to `/etc/collectd/collectd.conf` to configure the plugin:

```
TypesDB "/opt/docker-collectd-plugin/dockerplugin.db"
LoadPlugin python
```

```
<Plugin python>
  ModulePath "/opt/docker-collectd-plugin"
  Import "dockerplugin"

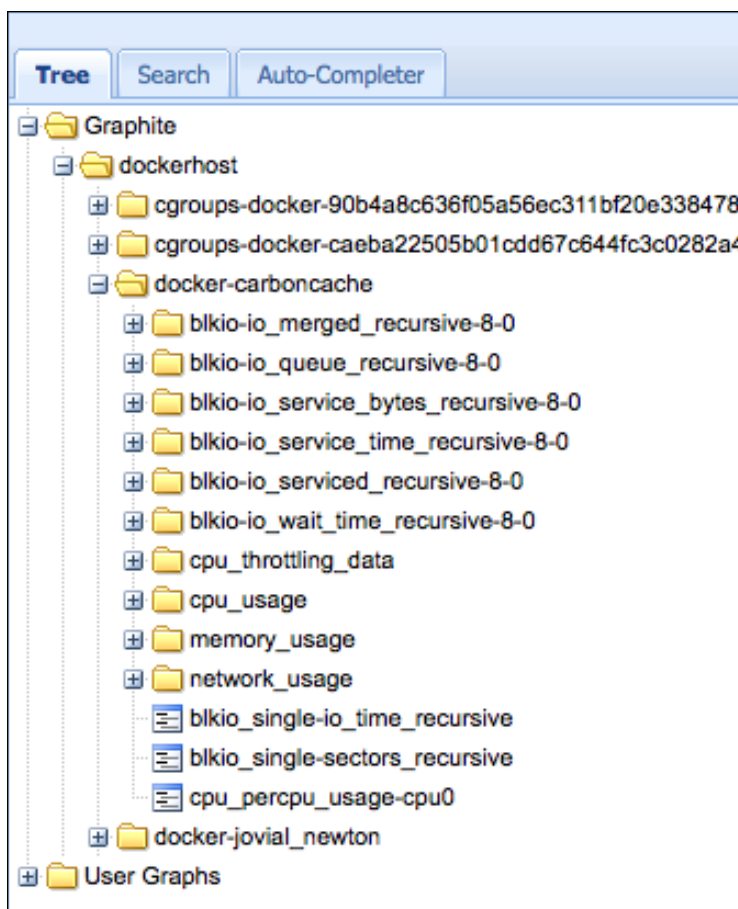
  <Module dockerplugin>
    BaseURL "unix://var/run/docker.sock"
    Timeout 3
  </Module>
</Plugin>
```

4. Finally, restart `collectd` to reflect the preceding configuration changes via the following command:  
`dockerhost$ systemctl restart collectd.service`



There is a case in which we don't want to install a whole stack of Python to just query the Docker container stat's endpoint. In this case, we can use the lower-level `curl_json` plugin of `collectd` to gather statistics about our container. We can configure it to make a request against the container stat's endpoint and parse the resulting JSON into a set of `collectd` metrics. More on how the plugin works can be found at [https://collectd.org/documentation/manpages/collectd.conf.5.shtml#plugin\\_curl\\_json](https://collectd.org/documentation/manpages/collectd.conf.5.shtml#plugin_curl_json).

In the following screenshot, we can explore the metrics given out by the `cgroups` plugin from our Graphite deployment at <http://docker/compose>:



## Running collectd inside Docker

If we want to deploy our `collectd` configuration similarly to our applications, we can run it inside Docker as well. The following is an initial `Dockerfile` that we can use to start with deploying `collectd` as a running Docker container:

```
FROM debian:jessie

RUN apt-get update && \
    apt-get --no-install-recommends install -y \
        collectd-core

ADD collectd.conf /etc/collectd/collectd.conf
ENTRYPOINT ["collectd", "-f"]
```



Most plugins look at the `/proc` and `/sys` filesystems. In order for `collectd` inside a Docker container to access these files, we need to mount them as Docker volumes, such as `--volume /proc:/host/proc`. However, most plugins currently read the hardwired `/proc` and `/sys` paths. There is ongoing discussion to make this configurable. Refer to this GitHub page to track its progress: <https://github.com/collectd/collectd/issues/1169>.

## Consolidating logs in an ELK stack

Not all statuses of our Docker hosts and containers are readily available to be queried with our monitoring solution in `collectd` and Graphite. Some events and metrics are only available as raw lines of text in log files. We need to transform these raw and unstructured logs to meaningful metrics. Similar to raw metrics, we can later ask higher-level questions on what is happening in our Docker-based application through analytics.

The ELK stack is a popular combination suite from Elastic that addresses these problems. Each letter in the acronym represents each of its components. The following is a description of each of them:

- **Logstash:** Logstash is the component that is used to collect and manage logs and events. It is the central point that we use to collect all the logs from different log sources, such as multiple Docker hosts and containers running in our deployment. We can also use Logstash to transform and annotate the logs we receive. This allows us to search and explore the richer features of our logs later.

- **Elasticsearch:** Elasticsearch is a distributed search engine that is highly scalable. Its sharding capabilities allow us to grow and scale our log storage as we continue to receive more and more logs from our Docker containers. Its database engine is document-oriented. This allows us to store and annotate logs as we see fit as we continue to discover more insights about the events we are managing in our large Docker deployments.
- **Kibana:** Kibana is an analytics and search dashboard for Elasticsearch. Its simplicity allows us to create dashboards for our Docker applications. However, Kibana is also very flexible to customize, so we can build dashboards that can provide valuable insights to people who want to understand our Docker-based applications, whether it is a low-level technical detail or higher-level business need.

In the remaining parts of this section, we will set up each of these components and send our Docker host and container logs to it. The next few steps describe how to build the ELK stack:

1. First, launch the official Elasticsearch image in our Docker host. We will put a container name so that we can link it easily in the later steps, as follows:

```
dockerhost$ docker run -d --name=elastic elasticsearch:1.7.1
```

2. Next, we will run Kibana's official Docker image by linking it against the Elasticsearch container we created in the previous step. Note that we publicly mapped the exposed port 5601 to port 80 in our Docker host so that the URL for Kibana is prettier, as follows:

```
dockerhost$ docker run -d --link elastic:elasticsearch \
  -p 80:5601 kibana:4.1.1
```

3. Now, prepare our Logstash Docker image and configuration. Prepare the following Dockerfile to create the Docker image:

```
FROM logstash:1.5.3
```

```
ADD logstash.conf /etc/logstash.conf
```

```
EXPOSE 1514/udp
```

4. In this Docker image, configure Logstash as a Syslog server. This explains the exposed UDP port in the preceding Dockerfile. As for the `logstash.conf` file, the following is the basic configuration to make it listen as a Syslog server. The latter part of the configuration shows that it sends logs to an Elasticsearch called `elasticsearch`. We will use this as the hostname when we link the Elasticsearch container we ran earlier:

```
input {
  syslog {
```

```
    port => 1514
    type => syslog
  }

output {
  elasticsearch {
    host => "elasticsearch"
  }
}
```



Logstash has a wealth of plugins so that it can read a wide variety of log data sources. In particular, it has a `collectd` codec plugin. With this, we can use an ELK stack instead of Graphite to monitor our metrics.

For more information on how to do this setup, visit <https://www.elastic.co/guide/en/logstash/current/plugins-codecs-collectd.html>.

5. Now that we have prepared all the files needed, type the following command to create it as the `hubuser/logstash` Docker image:

```
dockerhost$ docker build -t hubuser/logstash .
```

6. Run Logstash with the following command. Note that we are exposing port 1514 to the Docker host as the Syslog port. We also linked the Elasticsearch container named `elastic` that we created earlier. The target name is set to `elasticsearch` as it is the hostname of Elasticsearch that we configured earlier in `logstash.conf` to send the logs to:

```
dockerhost$ docker run --link elastic:elasticsearch -d \
  -p 1514:1514/udp hubuser/logstash -f /etc/logstash.conf
```

7. Next, let's configure our Docker host's Syslog service to forward it to our Logstash container. As a basic configuration, we can set up Rsyslog to forward all the logs. This will include the logs coming from the Docker engine daemon as well. To do this, create the `/etc/rsyslog.d/100-logstash.conf` file with the following content:

```
*.* @dockerhost:1514
```

8. Finally, restart Syslog to load the changes in the previous step by typing the following command:

```
dockerhost$ systemctl restart rsyslog.service
```

We now have a basic functioning ELK stack. Let's now test it by sending a message to Logstash and seeing it appear in our Kibana dashboard:

1. First, type the following command to send a test message:  

```
dockerhost$ logger -t test 'message to elasticsearch'
```
2. Next, go to our Kibana dashboard by visiting <http://dockerhost>. Kibana will now ask us to set the default index. Use the following default values and click on **Create** to start indexing:

Discover Visualize Dashboard **Settings**

Indices Advanced Objects About

Index Patterns

**Warning** No default index pattern. You must select or create one to continue.

## Configure an index pattern

In order to use Kibana you must configure at least one index pattern. Index patterns are used to identify the Elasticsearch index to run search and analytics against. They are also used to configure fields.

☒ Index contains time-based events  
☐ Use event times to create index names

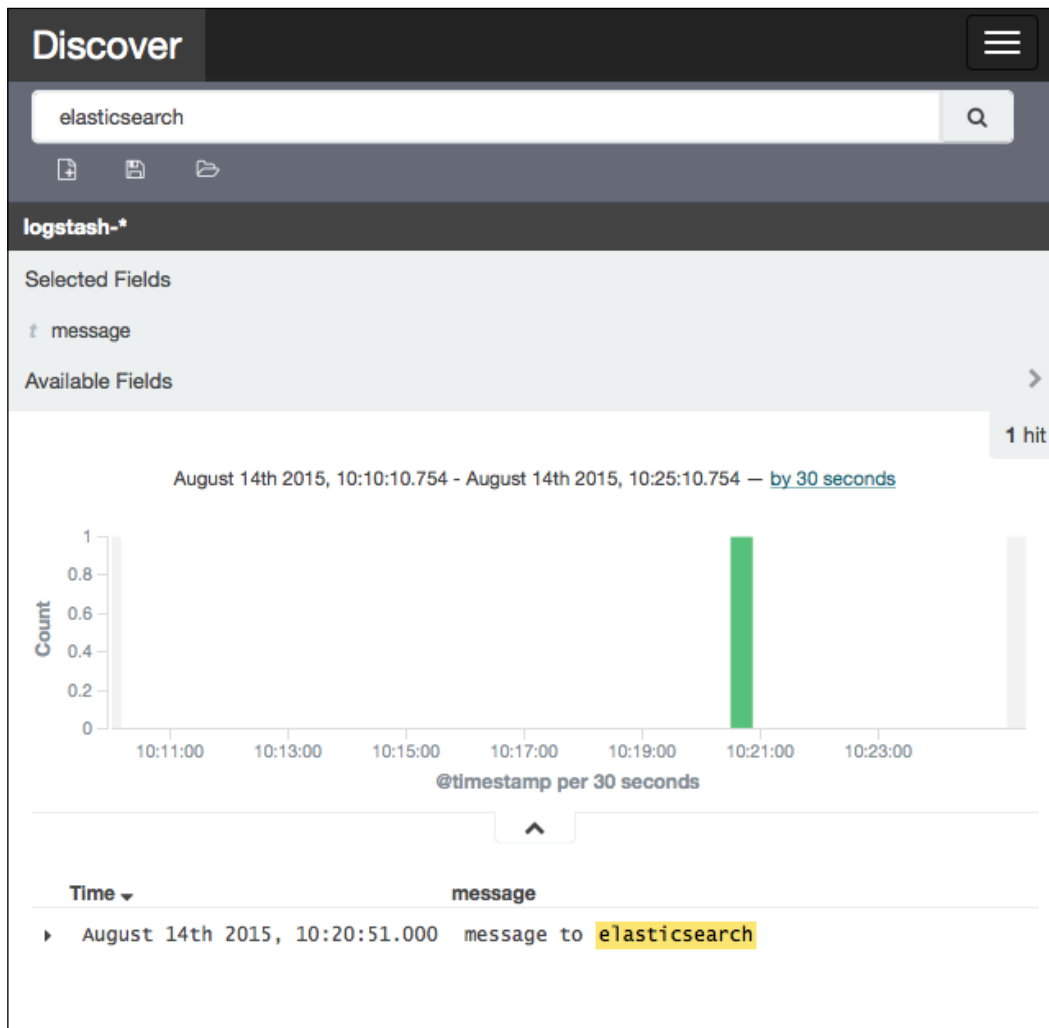
**Index name or pattern**  
Patterns allow you to define dynamic index names using \* as a wildcard. Example: logstash-\*

logstash-\*

**Time-field name** ⓘ refresh fields  
@timestamp

Create

- Go to `http://dockerhost/#discover` and type `elasticsearch` in the search. The following screenshot shows the Syslog message we generated earlier:



There are lot more things we can do on the ELK stack to optimize our logging infrastructure. We can add Logstash plugins and filters to annotate the logs we receive from our Docker hosts and containers. Elasticsearch can be scaled out and tuned to increase its capacity as our logging needs increase. We can create Kibana dashboards to share metrics. To find out more details on how to tune our ELK stack, visit Elastic's guides at <https://www.elastic.co/guide>.

## Forwarding Docker container logs

Now that we have a basic functional ELK stack, we can start forwarding our Docker logs to it. From Docker 1.7 onwards, support for custom logging drivers has been available. In this section, we will configure our Docker host to use the syslog driver. By default, Syslog events from Docker will go to the Docker host's Syslog service and since we configured Syslog to forward to our ELK stack, we will see the container logs there. Follow the next few steps to start receiving our container logs in the ELK stack:

1. The Docker engine service is configured via Systemd on our Debian Jessie host. To update how it runs in our Docker host, create a Systemd unit file called `/etc/systemd/system/docker.service.d/10-syslog.conf` with the following content:
 

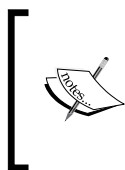
```
[Service]
ExecStart=
ExecStart=/usr/bin/docker daemon -H fd:// \
    --log-driver=syslog
```
2. Apply the changes on how we will run Docker in our host by reloading the Systemd configuration. The following command will do this:
 

```
dockerhost$ systemctl daemon-reload
```
3. Finally, restart the Docker engine daemon by issuing the following command:
 

```
dockerhost$ systemctl restart docker.service
```
4. Optionally, apply any Logstash filtering if we want to do custom annotations on our Docker container's logs.

Now, any standard output and error streams coming out from our Docker container should be captured to our ELK stack. We can do some preliminary tests to confirm that the setup works. Type the following command to create a test message from Docker:

```
dockerhost$ docker run --rm busybox echo message to elk
```



The `docker run` command also supports the `--log-driver` and `--log-opt=` command-line options to set up the logging driver only for the container we want to run. We can use it to further tune our logging policies for each Docker container running in our Docker host.

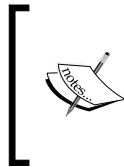


After typing the preceding command, our message should now be stored in Elasticsearch. Let's go to our Kibana endpoint in `http://dockerhost`, and search for the word `message to elk` in the textbox. It should give the Syslog entry for the message we sent earlier. The following screenshot is what the search result should look like in our Kibana results:



In the preceding screenshot, we can see the message we sent. There is also other information about Syslog. Docker's Syslog driver sets the default syslog annotations on facility and severity as **system** and **informational**, respectively. In addition, the preceding program is set to **docker/c469a2dfdc9a**.

The **c469a2dfdc9a** string is the container ID of the busybox image we ran earlier. The default program label for Docker containers is set in the `docker/<container-id>` format. All of the preceding default annotations can be configured by passing arguments to the `--log-opt=[]` option.



Aside from the Syslog and JSON file-logging drivers, Docker supports several other endpoints to send logs to. More information about all the logging drivers and their respective usage guides can be found in <https://docs.docker.com/reference/logging>.

## Other monitoring and logging solutions

There are several other solutions for us to deploy to monitor and log infrastructure to support our Docker-based application. Some of them already have built-in support for monitoring Docker containers. Others should be combined with other solutions, such as the ones we showed previously because they only focus on a specific part of monitoring or logging.

With others, we may have to do some workarounds. However, their benefits clearly outweigh the compromise we have to make. While the following list is not exhaustive, these are a few stacks we can explore to create our logging and monitoring solutions:

- cAdvisor (<http://github.com/google/cadvisor>)
- InfluxDB (<http://influxdb.com>)
- Sensu (<http://sensuapp.org>)
- Fluentd (<http://www.fluentd.org/>)
- Graylog (<http://www.graylog.org>)
- Splunk (<http://www.splunk.com>)

Sometimes, our operations staff and developers running and developing our Docker applications are not yet mature enough or do not want to focus on maintaining such monitoring and logging infrastructures. There are several hosted monitoring and logging platforms that we can use so that we can focus on actually writing and improving the performance of our Docker application.

Some of them work with existing monitoring and logging agents, such as Syslog and collectd. With others, we may have to download and deploy their agents to be able to forward the events and metrics to their hosted platform. The following is a nonexhaustive list of some solutions we may want to consider:

- New Relic (<http://www.newrelic.com>)
- Datadog (<http://www.datadoghq.com>)
- Librato (<http://www.librato.com>)
- Elastic's Found (<http://www.elastic.co/found>)
- Treasure Data (<http://www.treasuredata.com>)
- Splunk Cloud (<http://www.splunk.com>)

## Summary

We now know why it is important to monitor our Docker deployments in a scalable and accessible manner. We deployed collectd and Graphite to monitor our Docker container's metrics. We rolled out an ELK stack to consolidate the logs coming from various Docker hosts and containers.

In addition to raw metrics and events, it is also important to know what it means for our application. Graphite-web and Kibana allow us to create custom dashboards and analysis to provide insight in to our Docker applications. With these monitoring tools and skills in our arsenal, we should be able to operate and run our Docker deployments well in production.

In the next chapter, we will start doing performance tests and benchmark how our Docker applications fare well with a high load. We should be able to use the monitoring systems we deployed to observe and validate our performance testing activities there.

# 5

## Benchmarking

In optimizing our Docker applications, it is important to validate the parameters that we tuned. Benchmarking is an experimental way of identifying if the elements we modified in our Docker containers performed as expected. Our application will have a wide area of options to be optimized. The Docker hosts running them have their own set of parameters such as memory, networking, CPU, and storage as well. Depending on the nature of our application, one or more of these parameters can become a bottleneck. Having a series of tests to validate each component with benchmarks is important for guiding our optimization strategy.

Additionally, by creating proper performance tests, we can also identify the limits of the current configuration of our Docker-based application. With this information, we can start exploring infrastructure parameters such as scaling out our application by deploying them on more Docker hosts. We can also use this information to scale up the same application by moving our workload to a Docker host with higher storage, memory, or CPU. And when we have hybrid cloud deployments, we can use these measurements to identify which cloud provider gives our application its optimum performance.

Measuring how our application responds to these benchmarks is important when planning the capacity needed for our Docker infrastructure. By creating a test workload simulating peak and normal conditions, we can predict how our application will perform once it is released to production.

In this chapter, we will cover the following topics to benchmark a simple web application deployed in our Docker infrastructure:

- Setting up Apache JMeter for benchmarking
- Creating and designing a benchmark workload
- Analyzing application performance

## Setting up Apache JMeter

Apache JMeter is a popular application used to test the performance of web servers. Besides load testing web servers, the open source project grew to support testing other network protocols such as LDAP, FTP, and even raw TCP packets. It is highly configurable, and powerful enough to design complex workloads of different usage patterns. This feature can be used to simulate thousands of users suddenly visiting our web application thus inducing a spike in the load.

Another feature expected in any load-testing software is its data capture and analysis functions. JMeter has such a wide variety of data recording, plotting, and analysis features that we can explore the results of our benchmarks right away. Finally, it has a wide variety of plugins that may already have the load pattern, analysis, or network connection that we plan to use.



More information about the features and how to use Apache JMeter can be found on its website at <http://jmeter.apache.org>.

In this section, we will deploy an example application to benchmark, and prepare our workstation to run our first JMeter-based benchmark.

## Deploying a sample application

We can also bring our own web application we want to benchmark if we please. But for the rest of this chapter, we will benchmark the following application described in this section. The application is a simple Ruby web application deployed using Unicorn, a popular Ruby application server. It receives traffic via a Unix socket from Nginx. This setup is very typical for most Ruby applications found in the wild.

In this section, we will deploy this Ruby application in a Docker host called webapp. We will use separate Docker hosts for the application, benchmark tools, and monitoring. This separation is important so that the benchmark and monitoring instrumentation we run doesn't affect the benchmark results.

The next few steps show us how to build and deploy our simple Ruby web application stack:

1. First, create the Ruby application by creating the following Rack `config.ru` file:

```
app = proc do |env|
```

```
  Math.sqrt rand
```

```

    [200, {}, %w(hello world)]
  end
end
run app

```

2. Next, we package the application as a Docker container with the following Dockerfile:

```

FROM ruby:2.2.3

RUN gem install unicorn
WORKDIR /app

COPY . /app

VOLUME /var/run/unicorn

CMD unicorn -l /var/run/unicorn/unicorn.sock

```

3. Now we will create the Nginx configuration file `nginx.conf`. It will forward requests to our Unicorn application server through the Unix socket that we created in the previous step. In logging the request, we will record `$remote_addr` and `$response_time`. We will pay particular attention to these metrics later when we analyze our benchmark results:

```

events { }

http {
    log_format unicorn '$remote_addr [$time_local]'
        '"$request" $status'
        '$body_bytes_sent $request_time';
    access_log /var/log/nginx/access.log unicorn;

    upstream app_server {
        server unix:/var/run/unicorn/unicorn.sock;
    }
    server {
        location / {
            proxy_pass http://app_server;
        }
    }
}

```

4. The preceding Nginx configuration will then be packaged as a Docker container with the following Dockerfile:

```
FROM nginx:1.9.4
```

```
COPY nginx.conf /etc/nginx/nginx.conf
```

5. The last component will be a `docker-compose.yml` file to tie the two Docker containers together for deployment:

```
web:
```

```
  log_opt:
```

```
    syslog-tag: nginx
```

```
  build: ./nginx
```

```
  ports:
```

```
    - 80:80
```

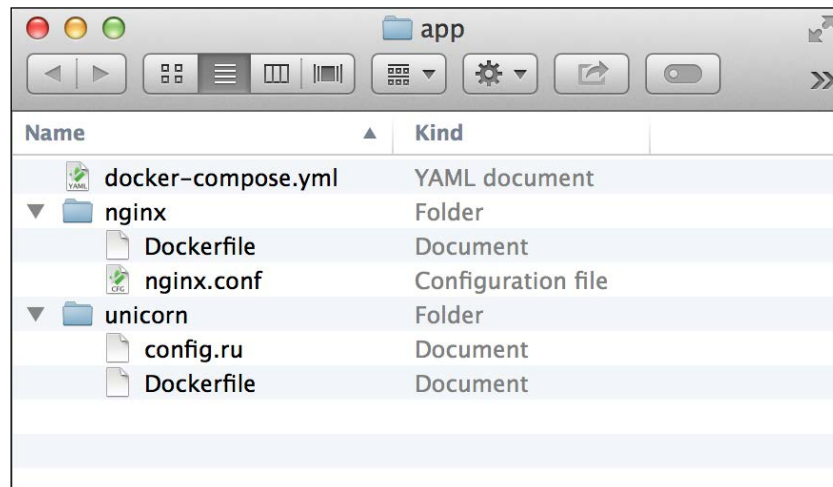
```
  volumes_from:
```

```
    - app
```

```
app:
```

```
  build: ./unicorn
```

In the end, we will have the files shown in the following screenshot in our code base:



After preparing our Dockerized web application, let us now deploy it to our Docker host by typing the following command:

```
webapp$ docker-compose up -d
```



Docker Compose is a tool for creating multi-container applications. It has a schema defined in YAML to describe how we want our Docker containers to run and link to each other.

Docker Compose supports a `curl | bash` type of installation. To quickly install it on our Docker host, type the following command:

```
dockerhost$ curl -L https://github.com/docker/compose/
releases/download/1.5.2/docker-compose-`uname -s`-`uname
-m` \
    > /usr/local/bin/docker-compose
```

We only covered Docker Compose in passing in this chapter. However, we can get more information about Docker Compose on the documentation website found at <http://docs.docker.com/compose>.

Finally, let us conduct a preliminary test to determine if our application works properly:

```
$ curl http://webapp.dev
hello world
```

Now we are done preparing the application that we want to benchmark. In the next section, we will prepare our workstation to perform the benchmarks by installing Apache JMeter.

## Installing JMeter

For the rest of this chapter, we will use Apache JMeter version 2.13 to perform our benchmarks. In this section, we will download and install it in our workstation. Follow the next few steps to set up JMeter properly:

1. To begin, go to JMeter's download web page at [http://jmeter.apache.org/download\\_jmeter.cgi](http://jmeter.apache.org/download_jmeter.cgi).
2. Select the link for **apache-jmeter-2.13.tgz** to begin downloading the binary.
3. When the download finishes, extract the tarball by typing the following command:  

```
$ tar -xzf apache-jmeter-2.13.tgz
```
4. Next, we will add the `bin/` directory to our `$PATH` so that JMeter can be easily launched from the command line. To do this, we will type the following command in our terminal:  

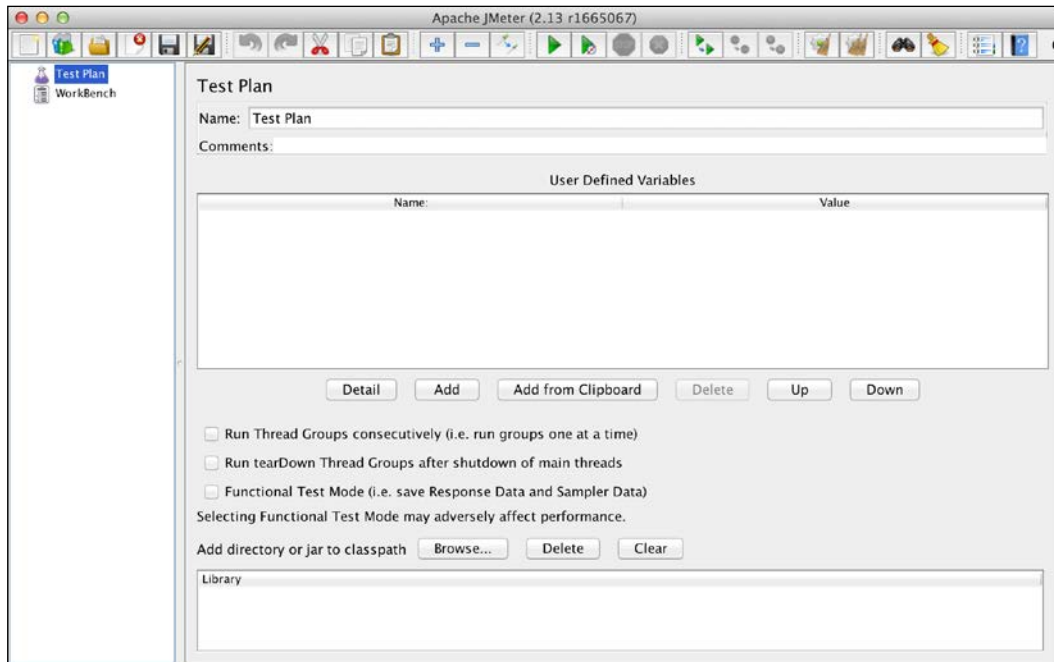
```
$ export PATH=$PATH:`pwd`/apache-jmeter-2.13/bin
```



5. Finally, launch JMeter by typing the following command:

```
$ jmeter
```

We will now see the JMeter UI just like the following screenshot. Now we are finally ready to write the benchmark for our application!:



Note that Apache JMeter is a Java application. According to the JMeter website, it requires at least Java 1.6 to work. Make sure you have a **Java Runtime Environment (JRE)** properly set up before installing JMeter.



If we were in a Mac OSX environment, we could use Homebrew and just type the following command:

```
$ brew install jmeter
```

For other platforms, the instructions described earlier should be sufficient to get started. More information on how to install JMeter can be found at <http://jmeter.apache.org/usermanual/get-started.html>.

## Building a benchmark workload

Writing benchmarks for an application is an open-ended area to explore. Apache JMeter can be overwhelming at first. It has several options to tune in order to write our benchmarks. To begin, we can use the "story" of our application as a start. The following are some of the questions we can ask ourselves:

- What does our application do?
- What is the persona of our users?
- How do they interact with our application?

Starting with these questions, we can then translate them into actual requests to our application.

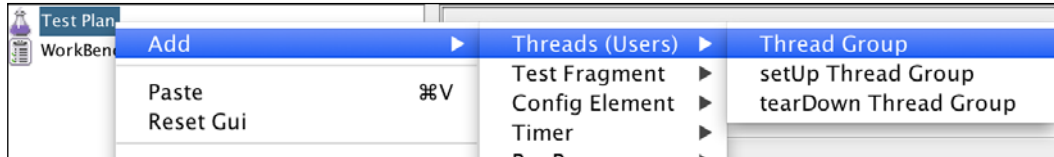
In the sample application that we wrote in the earlier section, we have a web application that displays `Hello World` to our users. In web applications, we are typically interested with the throughput and response time. Throughput refers to the number of users that can receive `Hello World` at a time. Response time describes the time lag before the user receives the `Hello World` message from the moment they requested it.

In this section, we will create a preliminary benchmark in Apache JMeter. Then we will begin analyzing our initial results with JMeter's analysis tools and the monitoring stack that we deployed in *Chapter 4, Monitoring Docker Hosts and Containers*. After that, we will iterate on the benchmarks we developed, and tune it. This way, we know that we are benchmarking our application properly.

## Creating a test plan in JMeter

A series of benchmarks in Apache JMeter is described in a test plan. A test plan describes a series of steps that JMeter will execute like performing requests to a web application. Each step in a test plan is called an element. These elements themselves can have one or more elements as well. In the end, our test plan will look like a tree—an hierarchy of elements to describe the benchmark we want for our application.

To add an element into our test plan, we simply right-click on the parent element that we want, and then select **Add**. This opens a context menu of elements that can be added to the selected parent element. In the following screenshot, we add a **Thread Group** element to the main element, **Test Plan**:



The next few steps show the way to create a test plan conducting the benchmark that we want:

1. First, let us rename the **Test Plan** to something more appropriate. Click on the **Test Plan** element. This will update the main JMeter window on the right. In the form field labeled **Name**;, set the value to **Unicorn Capacity**.
2. Under the **Unicorn Capacity** test plan, create a thread group. Name this **Application Users**. We will configure this thread group to send 10,000 requests to our application from a single thread in the beginning. Use the following parameters for filling out the form to achieve this setting:
  - **Number of Threads**: 1
  - **Ramp-up Period**: 0 seconds
  - **Loop Count**: 120,000 times

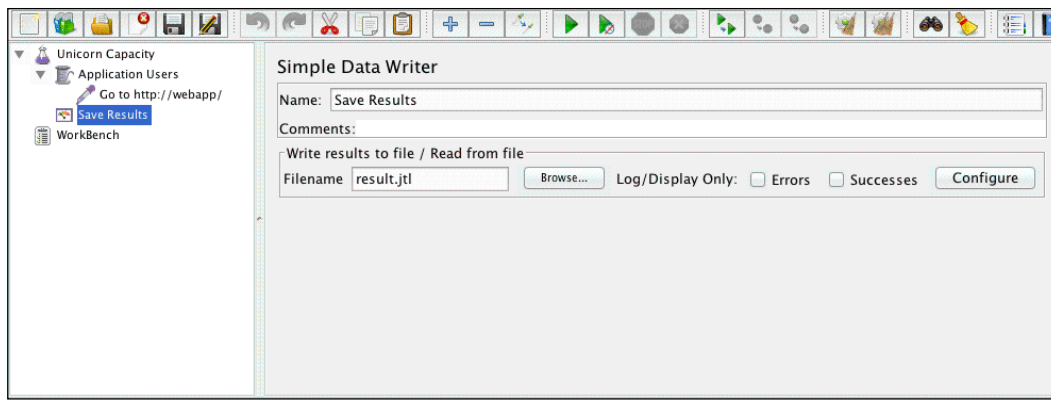


When we start developing our test plans, having a low loop count is useful. Instead of 120,000 loop counts, we can begin with 10,000 or even just 10 instead. Our benchmarks are shorter, but we get immediate feedback when developing it such as when we proceed to the next step. When we finish the whole test plan, we can always revert and tune it later to generate more requests.

3. Next, under the **Application Users** thread group, we create the actual request by adding **Sampler, HTTP Request**. This is the configuration where we set the details of how we make a request to our web application:
  - **Name**: Go to `http://webapp/`
  - **Server Name**: webapp

4. Finally, we configure how to save the test results by adding a listener under the **Unicorn Capacity** test plan. For this, we will add a **Simple Data Writer**, and name it **Save Result**. We set the **Filename** field to `result.jtl` to save our benchmark results in the said file. We will refer to this file later when we analyze the result of the benchmark.

Now we have a basic benchmark workload that generates 120,000 HTTP requests to `http://webapp/`. Then the test plan saves the result of each request in a file called `result.jtl`. The following is a screenshot of JMeter after the last step in creating the test plan:



Finally, it is time to run our benchmark. Go to the **Run** menu, then select **Start** to begin executing the test plan. While the benchmark is running, the **Start** button is grayed-out and disabled. When the execution finishes, it will be enabled again.

After running the benchmark, we will analyze the results by looking at the `result.jtl` file using JMeter's analysis tools in the next section.



There are various types of elements that can be placed in a JMeter test plan. Besides the three elements we used previously to create a basic benchmark for our application, there are several others that regulate requests, perform other network requests, and analyze data.

A comprehensive list of test plan elements and their description can be found on the JMeter page at [http://jmeter.apache.org/usermanual/component\\_reference.html](http://jmeter.apache.org/usermanual/component_reference.html).

## Analyzing benchmark results

In this section, we will analyze the benchmark results, and identify how the 120,000 requests affected our application. In creating web application benchmarks, there are typically two things we are usually interested in:

- How many requests can our application handle at a time?
- For how long is each request being processed by our application?

These two low-level web performance metrics can easily translate to the business implications of our application. For example, how many customers are using our application? Another one is, how are they perceiving the responsiveness of our application from a user experience perspective? We can correlate secondary metrics in our application such as CPU, memory, and network to determine our system capacity.

## Viewing the results of JMeter runs

Several listener elements of JMeter have features that render graphs. Enabling this when running the benchmark is useful when developing the test plan. But the time taken by the UI to render the results in real time, in addition to the actual benchmark requests, affects the performance of the test. Hence, it is better for us to separate the execution and analysis components of our benchmark. In this section, we will create a new test plan, and look at a few JMeter listener elements to analyze the data we acquired in `result.jtl`.

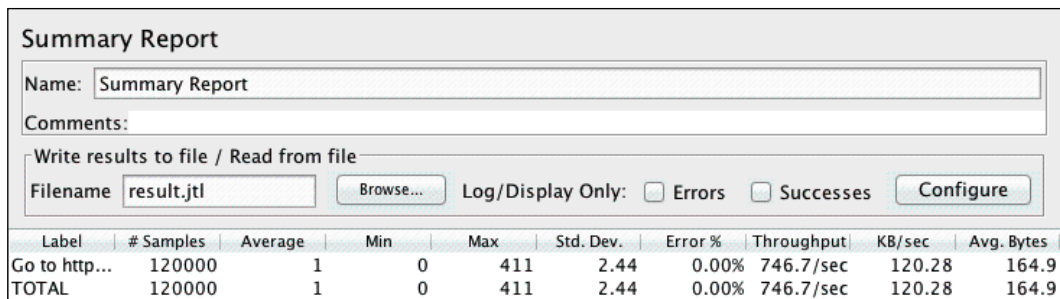
To begin our analysis, we first create a new test plan, and name this **Analyze Results**. We will add various listener elements under this test plan parent element. After this, follow the next few steps to add various JMeter listeners that can be used to analyze our benchmark result.

## Calculating throughput

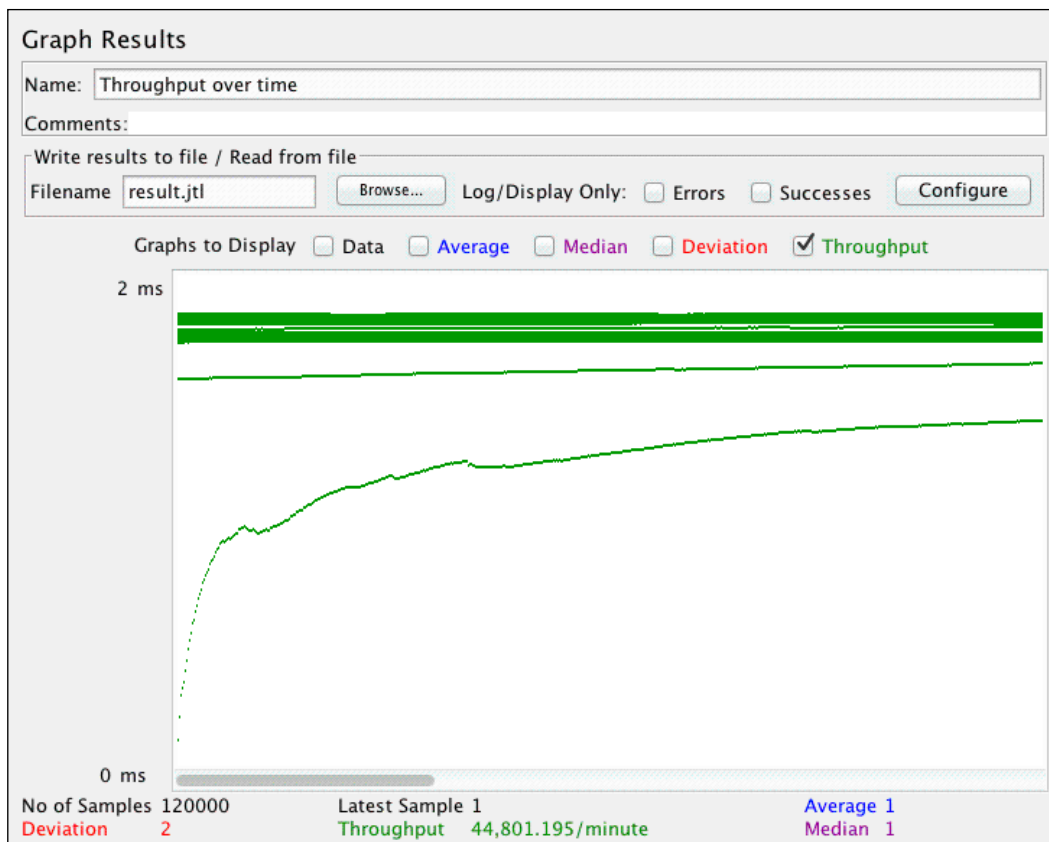
For our first analysis, we will use the **Summary Report** listener. This listener will show the throughput of our application. A measurement of throughput will show the number of transactions our application can handle per second.

To display the throughput, perform the following steps:

After loading the listener, fill out the **Filename** field by selecting the `result.jtl` file that we generated when we ran our benchmark. For the run we did earlier, the following screenshot shows that the 120,000 HTTP requests were sent to `http://webapp/` at a throughput of 746.7 requests per second:



We can also look at how throughput evolved over the course of our benchmark with the **Graph Results** listener. Create this listener under the **Analyze Results** test plan element and name it **Throughput over time**. Make sure that only the **Throughput** checkbox is marked (feel free to look at the other data points later though). After creating the listener, load our `result.jtl` test result again. The following screenshot shows how the throughput evolved over time:



As we can see in the preceding screenshot, the throughput started slow while JMeter tries to warm up its single-thread pool of requests. But after our benchmark continues to run, the throughput level settles at a stable level. By having a large number of loop counts earlier in our thread group, we were able to minimize the effect of the earlier ramp-up period.

This way, the throughput displayed in the **Summary Report** earlier is more or less a consistent result. Take note that the **Graph Results** listener wraps around its data points after several samples.

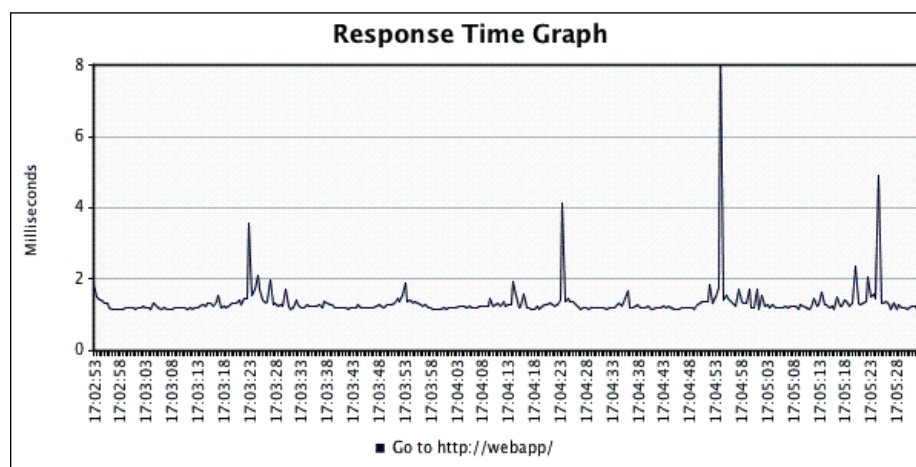


Remember that in benchmarking, the more samples we get, the more precise our observations can be!

## Plotting response time

Another metric we are interested in when we benchmark our application is the **response time**. The response time shows the duration for which JMeter has to wait before receiving the web page response from our application. In terms of real users, we can look at this as the time our users typed our web application's URL to the time everything got displayed in their web browser (it may not represent the real whole picture if our application renders some slow JavaScript, but for the application we made earlier, this analogy should suffice).

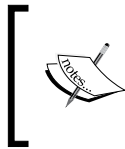
To view the response time of our application, we will use the **Response Time Graph** listener. As an initial setting, we can set the interval to 500 milliseconds. This will average some of the response times along 500 milliseconds in `result.jtl`. In the following image, you can see that our application's response time is mostly at around 1 millisecond:



If we want to display the response time in finer detail, we can decrease the interval to as low as 1 millisecond. Take note that this will take more time to display as the JMeter UI tries to plot more points in the application. Sometimes, when there are too many samples, JMeter may crash, because our workstation doesn't have enough memory to display the entire graph. In case of large benchmarks, we would be better off observing the results with our monitoring system. We will look at this data in the next section.

## Observing performance in Graphite and Kibana

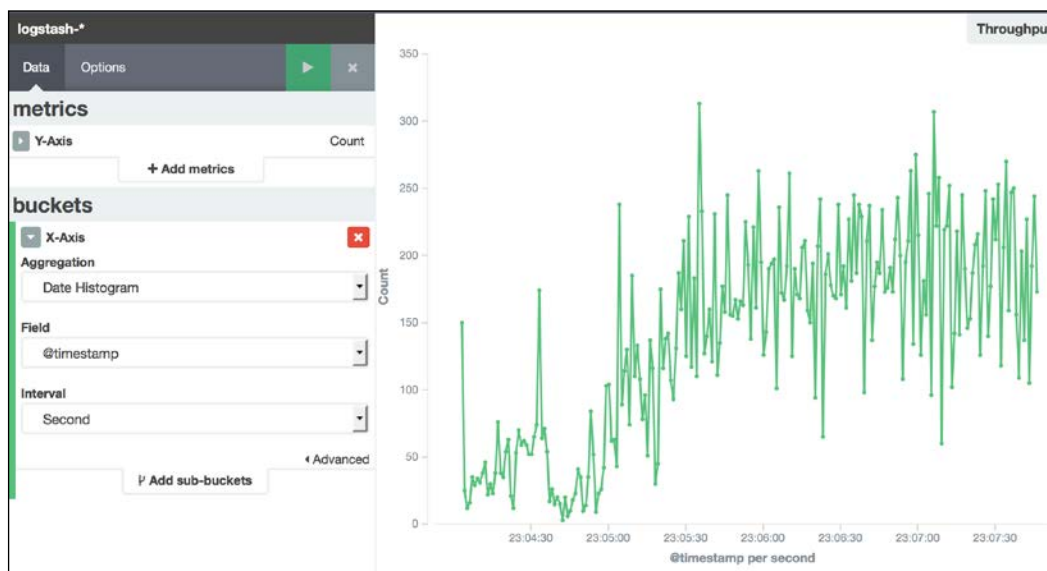
There might be a case when our workstation is so old that Java is not able to handle displaying 120,000 data points in its JMeter UI. To solve this, we can reduce the amount of data we have by either generating less requests in our benchmark or averaging out some of the data like we did earlier, when graphing response time. However, sometimes we want to see the full resolution of our data. This full view is useful when we want to inspect the finer details of how our application behaves. Fortunately, we already have a monitoring system in place for our Docker infrastructure that we built in *Chapter 4, Monitoring Docker Hosts and Containers*.



In this section, our monitoring and logging systems are deployed in a Docker host called `monitoring`. Our Docker host webapp that runs our application containers will have Collected and Rsyslog send events to the Docker host `monitoring`.



Remember the Nginx configuration mentioned when describing our benchmarks? The access log generated from the standard of the Nginx container is captured by Docker. If we use the same setup of our Docker daemon in *Chapter 4, Monitoring Docker Hosts and Containers*, these log events are captured by the local Rsyslog service. These Syslog entries will then be forwarded to the Logstash Syslog collector, and stored to Elasticsearch. We can then use the visualize feature of Kibana to look at the throughput of our application. The following analysis was made by counting the number of access log entries that Elasticsearch received per second:



We can also plot our application's response time during the course of the benchmark in Kibana. To do this, we first need to reconfigure our Logstash configuration to parse the data being received from the access log, and extract out the response time as a metric using filters. To do this, update `logstash.conf` from *Chapter 4, Monitoring Docker Hosts and Containers*, to add the `grok { }` filter as follows:

```
input {
  syslog {
    port => 1514
    type => syslog
  }
}

filter {
  if [program] == "docker/nginx" {
    grok {
```

```

    patterns_dir => ["/etc/logstash/patterns"]
    match => {
        "message" => "%{NGINXACCESS}"
    }
}
}
}

output {
    elasticsearch {
        host => "elasticsearch"
    }
}

```



Logstash's Filter plugins are used to intermediately process events before they reach our target storage endpoint such as Elasticsearch. It transforms raw data such as lines of text to a richer data schema in JSON that we can then use later for further analysis. More information about Logstash Filter plugins can be found at <https://www.elastic.co/guide/en/logstash/current/filter-plugins.html>.

The NGINXACCESS pattern being referred to in the preceding code is defined externally in what the `grok {}` filter calls a `patterns` file. Write the following as its content:

```

REQUESTPATH \"%{WORD:method} %{URIPATHPARAM} HTTP.*\"
HTTPREQUEST %{REQUESTPATH} %{NUMBER:response_code}
WEBMETRICS %{NUMBER:bytes_sent:int} %{NUMBER:response_time:float}
NGINXSOURCE %{IP:client} \[%{HTTPDATE:requested_at}\]
NGINXACCESS %{NGINXSOURCE} %{HTTPREQUEST} %{WEBMETRICS}

```

Finally, rebuild our `hubuser/logstash` Docker container from *Chapter 4, Monitoring Docker Hosts and Containers*. Don't forget to update the `Dockerfile` as follows to add the `patterns` file to our Docker context:

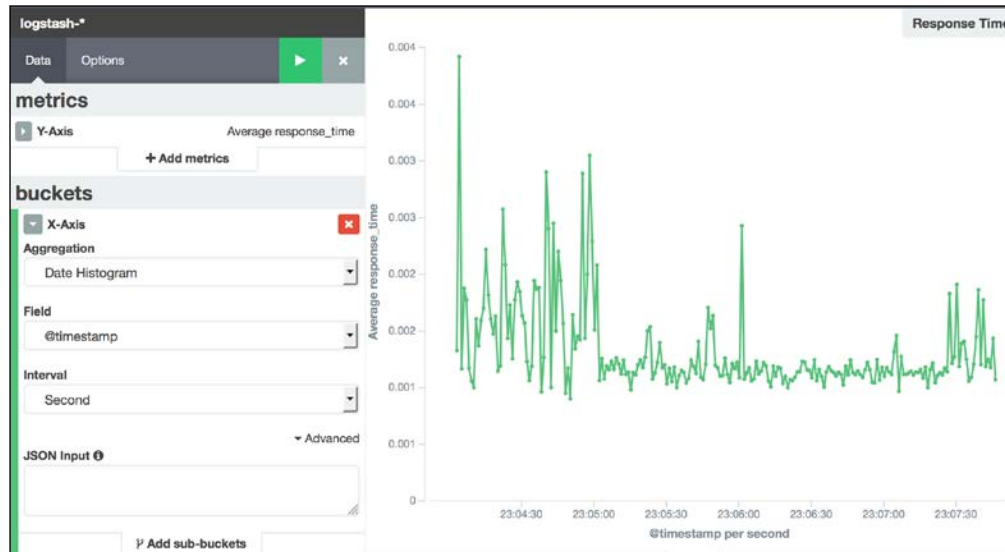
```

FROM logstash:1.5.3

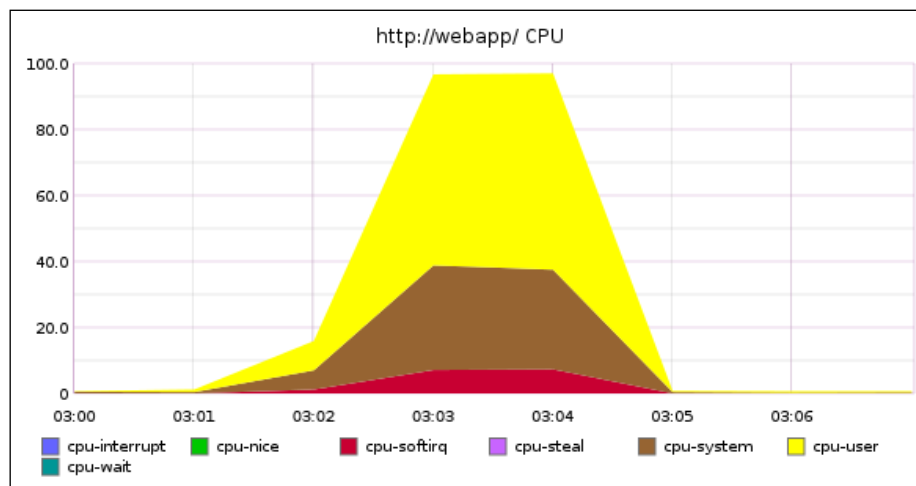
ADD logstash.conf /etc/logstash.conf
ADD patterns /etc/logstash/patterns/nginx
EXPOSE 1514/udp
EXPOSE 25826/udp

```

Now that we extracted the response times from the Nginx access logs, we can plot these data points in a Kibana visualization. The following is a screenshot of Kibana showing the average response time per second of the benchmark we ran earlier:



Another result that we can explore is the way our Docker host webapp responds with the load received from our benchmark. First we can check how our web application consumes the CPU of our Docker host. Let's log in to our monitoring system's graphite-web dashboard and plot the metrics `webapp.cpu-0.cpu-*` except `cpu-idle`. As we can see in the following image, the CPU of our Docker host goes to 100 percent usage the moment we start sending our application a lot of requests:



We can explore other system measurements of our Docker host to see how it is affected by the load of HTTP requests that it gets. The important point is that we use this data and correlate it to see how our web application behaved.



Apache JMeter version 2.13 and later include a backend listener that we can use to send JMeter data measurements in real time to external endpoints. By default, it ships with support for the Graphite wire protocol. We can use this feature to send benchmark results to the Graphite monitoring infrastructure that we built in *Chapter 4, Monitoring Docker Hosts and Containers*. More information on how to use this feature is available at <http://jmeter.apache.org/usermanual/realtime-results.html>.

## Tuning the benchmark

At this point, we already have a basic workflow of creating a test plan in Apache JMeter and analyzing the preliminary results. From here, there are several parameters we can adjust to achieve our benchmark objectives. In this section, we will iterate on our test plan to identify the limits of our Docker application.

## Increasing concurrency

The first parameter that we may want to tune is increasing the **Loop Count** of our test plan. Driving our test plan to generate more requests will allow us to see the effects of the load we induced to our application. This increases the precision of our benchmark experiments, because outlier events such as a slow network connection or hardware failure (unless we are testing that specifically!) affect our tests.

After having enough data points for our benchmarks, we may realize that the load being generated is not enough against our Docker application. For example, the current throughput we received from our first analysis may not simulate the behavior of real users. Let us say that we want to have 2000 requests per second. To increase the rate at which JMeter generates the requests, we can increase the number of threads in the thread group that we created earlier. This increases the number of concurrent requests that JMeter is creating at a time. If we want to simulate a gradual increase in the number of users, we can adjust the ramp-up period to be longer.



For workloads where we want to simulate a sudden increase of users, we can stick with a ramp-up period of 0 to start all the threads right away. In cases where we want to tune other behaviors such as a constant load and then a sudden spike, we can use the **Stepping Thread Group** plugin.

We may also want to limit it to precisely just 100 requests per second. Here, we can use `Timer` elements to control how our threads generate the request. To start limiting throughput, we can use the **Constant Throughput Timer**. This will make JMeter automatically slow down threads when it perceives that the throughput it is receiving from our web application is increasing too much.

Some of the benchmark techniques here are difficult to apply with the built-in Apache JMeter components. There are a variety of plugins that make it simpler to generate the load to drive our application. They are available as plugins. The Apache JMeter list of popularly used community plugins is found at <http://jmeter-plugins.org>.

## Running distributed tests

After tuning the concurrency parameters for a while, we realize that our result does not change. We may set JMeter to generate 10,000 requests at a time, but that will most likely crash our UI! In this case, we are already reaching the performance limits of our workstation while building the benchmarks. From this point, we can start exploring using a pool of servers that run JMeter to create distributed tests. Distributed tests are useful, because we can grab several servers from the cloud with higher performance to simulate spikes. It is also useful for creating load coming from several sources. This distributed setup is useful for simulating high-latency scenarios, where our users are accessing our Docker application from halfway across the world.

Execute the following steps for deploying Apache JMeter on several Docker hosts to perform a distributed benchmark:

1. First, create the following Dockerfile to create a Docker image called

hubuser/jmeter:

```
FROM java:8u66-jre
```

```
# Download URL for JMeter
```

```
RUN curl http://www.apache.org/dist/jmeter/binaries/apache-jmeter-2.13.tgz | tar xz
```

```
WORKDIR /apache-jmeter-2.13
```

```
EXPOSE 1099
```

```
EXPOSE 1100
```

```
ENTRYPOINT ["/bin/jmeter", "-j", "/dev/stdout", "-s", \
            "-Dserver_port=1099", "-Jserver.rmi.localport=1100"]
```

2. Next, provision the number of Docker hosts we want according to our cloud or server provider. Take note of the hostname or IP address of each Docker host. For our case, we created two Docker hosts called `dockerhost1` and `dockerhost2`.
3. Now, we will run the JMeter server on our Docker hosts. Log in to each of them, and type the following command:
 

```
dockerhost1$ docker run -p 1099:1099 -p 1100:1100 \
    hubuser/jmeter -Djava.rmi.server.hostname=dockerhost1
dockerhost2$ docker run -p 1099:1099 -p 1100:1100 \
    hubuser/jmeter -Djava.rmi.server.hostname=dockerhost2
```
4. To finalize our JMeter cluster, we will type the following command to launch the JMeter UI client connected to the JMeter servers:
 

```
$ jmeter -Jremote_hosts=dockerhost1,dockerhost2
```

With an Apache JMeter cluster at our disposal, we are now ready to run distributed tests. Note that the number of threads in the test plan specifies the thread count on each JMeter server. In the case of the test plan we made in the earlier section, our JMeter benchmark will generate 240,000 requests. We should adjust these counts according to the test workload we have in mind. Some of the guidelines we mentioned in the previous section can be used to tune our remote tests.

Finally, to start the remote tests, select **Remote Start All** from the **Run** menu. This will spawn the thread groups we created in our test plan to our JMeter servers in `dockerhost1` and `dockerhost2`. When we look at our access logs of Nginx, we can now see that the IP sources are coming from two different sources. The following IP addresses come from each of our Docker hosts:

```
172.16.132.216 [14/Sep/2015:16:...] "GET / HTTP/1.1" 200 20 0.003
172.16.132.187 [14/Sep/2015:16:...] "GET / HTTP/1.1" 200 20 0.003
172.16.132.216 [14/Sep/2015:16:...] "GET / HTTP/1.1" 200 20 0.003
172.16.132.187 [14/Sep/2015:16:...] "GET / HTTP/1.1" 200 20 0.002
172.16.132.216 [14/Sep/2015:16:...] "GET / HTTP/1.1" 200 20 0.002
172.16.132.187 [14/Sep/2015:16:...] "GET / HTTP/1.1" 200 20 0.003
```



More information on distributed and remote testing can be found at <http://jmeter.apache.org/usermanual/remote-test.html>.

## Other benchmarking tools

There are a few other benchmarking tools specifically for benchmarking web-based applications. The following is a short list of such tools with their links:

- **Apache Bench:** <http://httpd.apache.org/docs/2.4/en/programs/ab.html>
- **HP Lab's Httpperf:** <http://www.hpl.hp.com/research/linux/httpperf>
- **Siege:** <https://www.joedog.org/siege-home>

## Summary

In this chapter, we created benchmarks for gauging the performance of our Docker application. By using Apache JMeter and the monitoring system we set up in *Chapter 4, Monitoring Docker Hosts and Containers*, we analyzed how our application behaved under various conditions. We now have an idea about the limitations of our application, and will use it to further optimize it or to scale it out.

In the next chapter, we will talk about load balancers for scaling-out our application to increase its capacity.

# 6

## Load Balancing

No matter how we tune our Docker applications, we will reach our application's performance limits. Using the benchmarking techniques we discussed in the previous chapter, we should be able to identify the capacity of our application. In the near future, our Docker application's users will exceed this limit. We cannot turn these users away just because our Docker application cannot handle their requests anymore. We need to scale out our application so that it can serve our growing number of users.

In this chapter, we will talk about how to scale out our Docker applications to increase our capacity. We will use load balancers, which are a key component in the architecture of various web scale applications. Load balancers distribute our application's users to multiple Docker applications deployed in our farm of Docker hosts. The following steps covered in this chapter will help us accomplish this:

- Preparing a Docker host farm
- Balancing load with Nginx
- Scaling out our Docker applications
- Managing zero downtime releases with load balancers

### Preparing a Docker host farm

A key component in load balancing our Docker application is to have a farm of servers to send our application's requests to. In the case of our infrastructure, this involves preparing a farm of Docker hosts to deploy our application to. The scalable way to do this is to have a common base configuration that is managed by configuration management software, such as Chef, as we previously covered in *Chapter 3, Automating Docker Deployments with Chef*.



After preparing the farm of Docker hosts, it is time to prepare the application that we will run. In this chapter, we will scale a simple NodeJS application. The rest of this section will describe how this application works.

The web application is a small NodeJS application written in a file called `app.js`. For the purpose of visualizing how our application load balances, we will also log some information about our application and the Docker host it is running in. The `app.js` file will contain the following code:

```
var http = require('http');

var server = http.createServer(function (request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  var version = "1.0.0";
  var log = {};
  log.header = 'mywebapp';
  log.name = process.env.HOSTNAME;
  log.version = version;
  console.log(JSON.stringify(log));
  response.end(version + " Hello World  " + process.env.HOSTNAME);
});
server.listen(8000);
```

To deploy the preceding application code, we will package it in a Docker image called `hubuser/app:1.0.0` with the following Dockerfile:

```
FROM node:4.0.0

COPY app.js /app/app.js
EXPOSE 8000
CMD ["node", "/app/app.js"]
```

Make sure that our Docker image is built and available at Docker Hub. This way, we can easily deploy it. Run this with the following command:

```
dockerhost$ docker build -t hubuser/app:1.0.0 .
dockerhost$ docker push hubuser/app:1.0.0
```

As the final step in our preparation, we will deploy our Docker application to three Docker hosts: `greenhost00`, `greenhost01`, and `greenhost02`. Log in to each of the hosts and type the following command to start the container:

```
greenhost00$ docker run -d -p 8000:8000 hubuser/app:1.0.0
greenhost01$ docker run -d -p 8000:8000 hubuser/app:1.0.0
greenhost02$ docker run -d -p 8000:8000 hubuser/app:1.0.0
```



Better yet, we can write a Chef cookbook that will deploy the Docker application that we just wrote.

## Balancing load with Nginx

Now that we have a pool of Docker applications to forward traffic to, we can prepare our load balancer. In this section, we will briefly cover Nginx, a popular web server that has high concurrency and performance. It is commonly used as a reverse proxy to forward requests to more dynamic web applications, such as the NodeJS one we wrote earlier. By configuring Nginx to have multiple reverse proxy destinations, such as our pool of Docker applications, it will balance the load of requests coming to it across the pool.

In our load balancer deployment, we will deploy our Nginx Docker container in a Docker host called `dockerhost`. After deployment, the Nginx container will start forwarding to the pool of Docker hosts called `greenhost*`, which we provisioned in the earlier section.

The following is a simple configuration of Nginx that will forward traffic to the pool of Docker applications that we deployed earlier. Save this file in `/root/nginx.conf` inside the `dockerhost` Docker host, as follows:

```
events { }

http {
    upstream app_server {
        server greenhost00:8000;
        server greenhost01:8000;
        server greenhost02:8000;
    }
    server {
        location / {
            proxy_pass http://app_server;
        }
    }
}
```

The preceding Nginx configuration file is basically composed of directives. Each directive has a corresponding effect on Nginx's configuration. To define our pool of applications, we will use the `upstream` directive to define a group of servers. Next, we will place the list of servers in our pool using the `server` directive. A server in the pool is normally defined in the `<hostname-or-ip>:<port>` format.



The following are the references referring to the described directives mentioned earlier:

- `upstream`—[http://nginx.org/en/docs/http/nginx\\_http\\_upstream\\_module.html#upstream](http://nginx.org/en/docs/http/nginx_http_upstream_module.html#upstream)
- `server`—[http://nginx.org/en/docs/http/nginx\\_http\\_upstream\\_module.html#server](http://nginx.org/en/docs/http/nginx_http_upstream_module.html#server)
- `proxy_pass`—[http://nginx.org/en/docs/http/nginx\\_http\\_proxy\\_module.html#proxy\\_pass](http://nginx.org/en/docs/http/nginx_http_proxy_module.html#proxy_pass)

Introductory material discussing the basics of directives can be found at [http://nginx.org/en/docs/beginners\\_guide.html#conf\\_structure](http://nginx.org/en/docs/beginners_guide.html#conf_structure).

Now that we have prepared our `nginx.conf` file, we can deploy our Nginx container together with this configuration. To perform this deployment, let's run the following command in our `dockerhost` Docker host:

```
dockerhost$ docker run -p 80:80 -d --name=balancer \
    --volume=/root/nginx.conf:/etc/nginx/nginx.conf:ro nginx:1.9.4
```

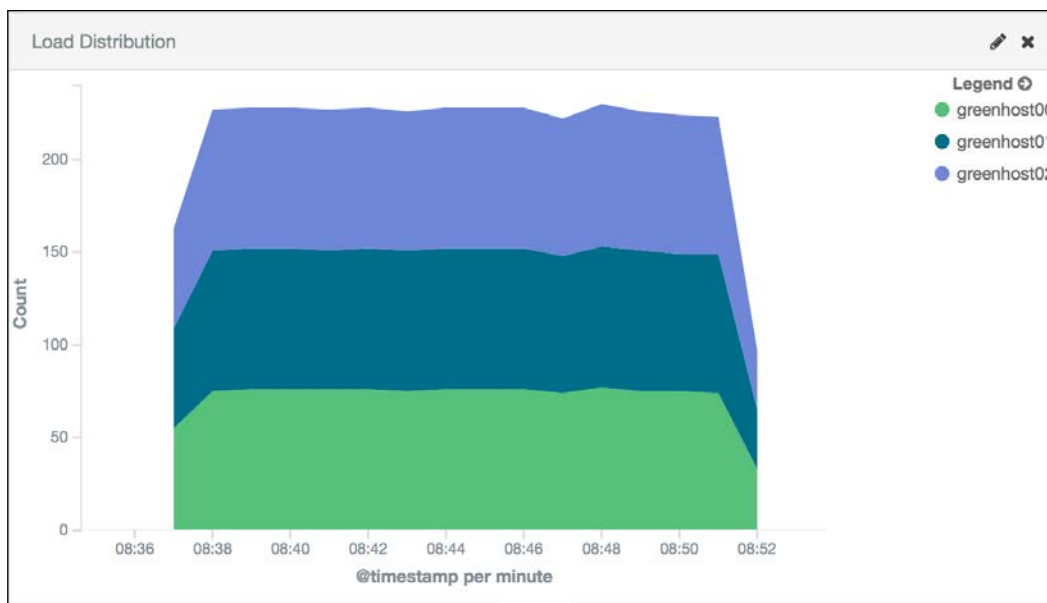
Our web application is now accessible via `http://dockerhost`. Each request will then be routed to one of the `hubuser/webapp:1.0.0` containers we deployed to our pool of Docker hosts.

To confirm our deployment, we can look at our Kibana visualization to show the distribution of traffic across our three hosts. To show the distribution of traffic, we must first generate load for our application. We can use our JMeter testing infrastructure described in *Chapter 5, Benchmarking*, to achieve this. For quick testing, we can also generate load using a long-running command similar to the following:

```
$ while true; do curl http://dockerhost && sleep 0.1; done
1.0.0 Hello World 56547aceb063
1.0.0 Hello World af272c6968f0
1.0.0 Hello World 7791edeefb8c
1.0.0 Hello World 56547aceb063
1.0.0 Hello World af272c6968f0
1.0.0 Hello World 7791edeefb8c
1.0.0 Hello World 56547aceb063
1.0.0 Hello World af272c6968f0
1.0.0 Hello World 7791edeefb8c
```

Recall that in the application we prepared earlier, we printed out `$HOSTNAME` as a part of the HTTP response. In the preceding case, the responses show the Docker container's hostname. Note that Docker containers get the short hash of their container IDs as their hostname by default. As we can note from the initial output of our test workload, we are getting responses from three containers.

We can visualize the response better in a Kibana visualization if we set up our logging infrastructure as we did in *Chapter 4, Monitoring Docker Hosts and Containers*. In the following screenshot, we can count the number of responses per minute according to the Docker host that the log entry came from:



We can note in the preceding figure that our workload gets distributed evenly by Nginx to our three Docker hosts: **greenhost00**, **greenhost01**, and **greenhost02**.



To properly visualize our deployment in Kibana we have to annotate our Docker containers and filter these log entries in Logstash so that they get properly annotated to Elasticsearch. We can do this via the following steps:

First, we will make sure that we use the `syslog-tag` option when deploying our Docker container. This makes our application easier to filter out later in Logstash. Run the following code:

```
greenhost01$ docker run -d -p 8000:8000 \
  --log-driver syslog \
  --log-opt syslogtag=webapp \
  hubuser/app:1.0.0
```

With this, Logstash will receive our Docker container's log entries with the `docker/webapp` tag. We can then use a Logstash filter as follows to get this information in Elasticsearch:

```
filter {
  if [program] == "docker/webapp" {
    json {
      source => "message"
    }
  }
}
```

## Scaling out our Docker applications

Now, suppose that the workload in the previous section starts to overload each of our three Docker hosts. Without a load balancer such as our preceding Nginx setup, our application's performance will start to degrade. This may mean a lower quality of service to our application's users or being paged in the middle of the night to perform heroic systems operations. However, with a load balancer managing the connections to our applications, it is very simple to add more capacity to scale out the performance of our application.

As our application is already designed to be load balanced, our scale-out process is very simple. The next few steps form a typical workflow on how to add capacity to a load-balanced application:

1. First, provision new Docker hosts with the same base configuration as the first three in our Docker host pool. In this section, we will create two new Docker hosts, named `greenhost03` and `greenhost04`.

2. The next step in our scale-out process is to then deploy our applications in these new Docker hosts. Type the same command before for deployment as the following one to each of the new Docker hosts:
 

```
greenhost03$ docker run -d -p 8000:8000 hubuser/app:1.0.0
greenhost04$ docker run -d -p 8000:8000 hubuser/app:1.0.0
```
3. At this point, new application servers in our pool are ready to accept connections. It is now time to add them as destinations to our Nginx-based load balancer. To add them to our pool of upstream servers, first update the `/root/nginx.conf` file, as follows:
 

```
events { }

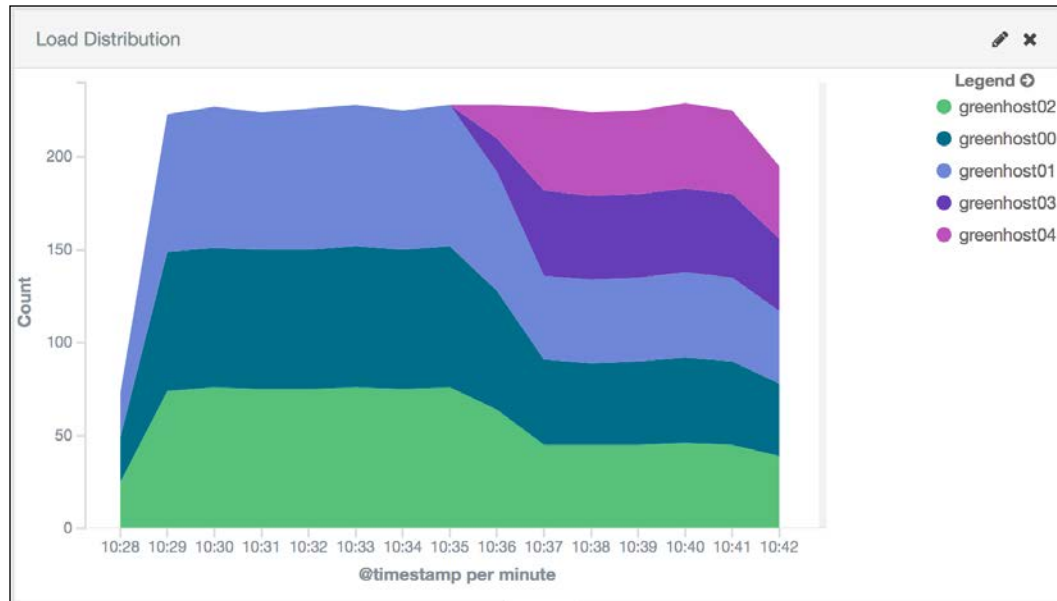
http {
    upstream app_server {
        server greenhost00:8000;
        server greenhost01:8000;
        server greenhost02:8000;
        server greenhost03:8000;
        server greenhost04:8000;
    }
    server {
        location / {
            proxy_pass http://app_server;
        }
    }
}
```
4. Finally, we will notify our running Nginx Docker container to reload its configuration. In Nginx, this is done by sending a `HUP` Unix signal to its master process. To send the signal to a master process inside the Docker container, type the following Docker command. Send the reload signal:
 

```
dockerhost$ docker kill -s HUP balancer
```



More information on how to control Nginx with various Unix signals is documented at <http://nginx.org/en/docs/control.html>.

Now that we are done scaling out our Docker application, let's look back at our Kibana visualization to observe the effect. The following screenshot shows the distribution of traffic across the five Docker hosts we currently have:



We can note in the preceding screenshot that after we reloaded Nginx, it started to distribute load across our new Docker containers. Before this, each Docker container received only a third of the traffic from Nginx. Now, each Docker application in the pool only receives a fifth of the traffic.

## Deploying with zero downtime

Another advantage of having our Docker application load balanced is that we can use the same load balancing techniques to update our application. Normally, operations engineers have to schedule downtime or a maintenance window in order to update an application deployed in production. However, as our application's traffic goes to a load balancer before it reaches our application, we can use this intermediate step to our advantage. In this section, we will employ a technique called blue-green deployments to update our running application with zero downtime.

Our current pool of `hubuser/app:1.0.0` Docker containers is called our *green* Docker host pool because it actively receives requests from our Nginx load balancer. We will update the application being served by our Nginx load balancer to pool of `hubuser/app:2.0.0` Docker containers. The following are the steps to perform the update:

1. First, let's update our application by changing the version string in our `app.js` file, as follows:

```
var http = require('http');

var server = http.createServer(function (request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  var version = "2.0.0";
  var log = {};
  log.header = 'mywebapp';
  log.name = process.env.HOSTNAME;
  log.version = version;
  console.log(JSON.stringify(log));
  response.end(version + " Hello World  " + process.env.HOSTNAME);
});

server.listen(8000);
```

2. After updating the content, we will prepare a new version of our Docker image called `hubuser/app:2.0.0` and publish it to Docker Hub via the following command:
 

```
dockerhost$ docker build -t hubuser/app:2.0.0 .
dockerhost$ docker push hubuser/app:2.0.0
```
3. Next, we will provision a set of Docker hosts called `bluehost01`, `bluehost02`, and `bluehost03`, either through our cloud provider or by buying actual hardware. This will become our *blue* Docker host pool.
4. Now that our Docker hosts are prepared, we will deploy our new Docker application on each of the new hosts. Type the following commands on each Docker host to perform the deployment:

```
bluehost00$ docker run -d -p 8000:8000 hubuser/app:2.0.0
bluehost01$ docker run -d -p 8000:8000 hubuser/app:2.0.0
bluehost02$ docker run -d -p 8000:8000 hubuser/app:2.0.0
```

Our *blue* Docker host pool is now prepared. It is called blue because although it is now live and running, it has yet to receive user traffic. At this point, we can do whatever is needed, such as performing preflight checks and tests before siphoning our users to the new version of our application.



After we are confident that our blue Docker host pool is fully functional and working, it will be time to send traffic to it. As in the scaling-out process of our Docker host pool, we will simply add our blue Docker hosts to the list of servers inside our `/root/nginx.conf` configuration, as follows:

```
events { }
```

```
http {  
    upstream app_server {  
        server greenhost00:8000;  
        server greenhost01:8000;  
        server greenhost02:8000;  
        server greenhost03:8000;  
        server greenhost04:8000;  
        server bluehost00:8000;  
        server bluehost01:8000;  
        server bluehost02:8000;  
    }  
    server {  
        location / {  
            proxy_pass http://app_server;  
        }  
    }  
}
```

To complete the activation, reload our Nginx load balancer by sending it the HUP signal through the following command:

```
dockerhost$ docker kill -s HUP balancer
```

At this point, Nginx sends traffic to both the old version (`hubuser/app:1.0.0`) and the new version (`hubuser/app:2.0.0`) of our Docker application. With this, we can completely verify that our new application is indeed working as expected because it now serves live traffic from our application's users. In the cases when it does not work properly, we can safely roll back by removing the `bluehost*` Docker hosts in the pool and resending the HUP signal to our Nginx container.

However, suppose we are already satisfied with our new application. We can then safely remove the old Docker application from our load balancer's configuration. In our `/root/nginx.conf` file, we can perform this by removing all the `greenhost*` lines, as follows:

```
http {  
    upstream app_server {  
        server bluehost00:8000;  
        server bluehost01:8000;
```

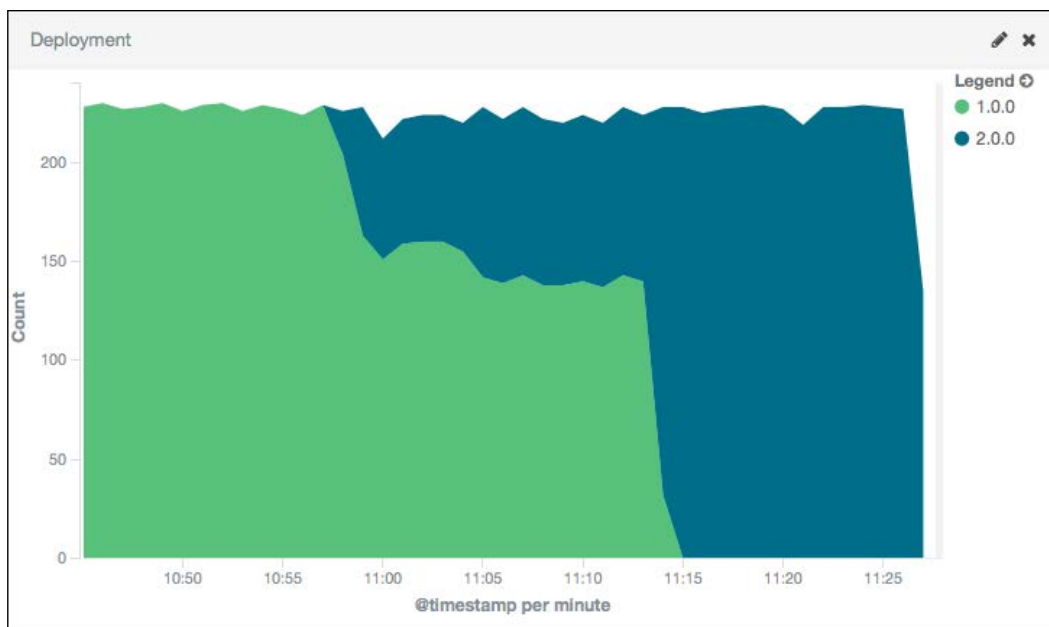
```

    server bluehost02:8000;
  }
  server {
    location / {
      proxy_pass http://app_server;
    }
  }
}

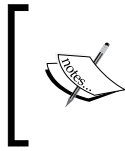
```

Now, we can complete our zero-downtime deployment with another `HUP` signal to Nginx. At this point, our blue Docker host pool serves all the production traffic of our application. This, therefore, becomes our new green Docker host pool. Optionally, we can deprovision our old green Docker host pool to save on resource usage.

The whole blue-green deployment process we did earlier can be summarized in the following Kibana visualization:



Note that in the preceding graph, our application still serves traffic even though we updated our application. Note also that before this, all of the traffic was distributed to our five **1.0.0** applications. After activating the blue Docker host pool, three-eighths of the traffic started going to version **2.0.0** of our application. In the end, we deactivated all the endpoints in our old green Docker host pool, and all of the application's traffic is now served by version **2.0.0** of our application.



More information about blue-green deployments and other types of zero-downtime release techniques can be found in a book called *Continuous Delivery* by Jez Humble and Dave Farley. The book's website can be found at <http://continuousdelivery.com>.

## Other load balancers

There are other tools that can be used to load balance applications. Some are similar to Nginx, where configuration is defined through external configuration files. Then, we can send a signal to the running process to reload the updated configuration. Some have their pool configurations stored in an outside store, such as Redis, etcd, and even regular databases, so that the list is dynamically loaded by the load balancer itself. Even Nginx has some of these functionalities with its commercial offering. There are also other open source projects that extend Nginx with third-party modules.

The following is a short list of load balancers that we can deploy as some form of Docker containers in our infrastructure:

- Redx (<https://github.com/rstudio/redx>)
- HAProxy (<http://www.haproxy.org>)
- Apache HTTP Server (<http://httpd.apache.org>)
- Vulcand (<http://vulcand.github.io/>)
- CloudFoundry's GoRouter (<https://github.com/cloudfoundry/gorouter>)
- dotCloud's Hipache (<https://github.com/hipache/hipache>)

There are also hardware-based load balancers that we can procure ourselves and configure via their own proprietary formats or APIs. If we use cloud providers, some of their own load balancer offerings would have their own cloud APIs that we can use as well.

## Summary

In this chapter, you learned the benefits of using load balancers and how to use them. We deployed and configured Nginx as a load balancer in a Docker container so that we can scale out our Docker application. We also used the load balancer to perform zero-downtime releases to update our application to a new version.

In the next chapter, we will continue to improve our Docker optimization skills by debugging inside the Docker containers we deploy.

# 7

## Troubleshooting Containers

Sometimes, instrumentation, such as the monitoring and logging system we set up in *Chapter 4, Monitoring Docker Hosts and Containers*, is not enough. Ideally, we should put in place a way to troubleshoot our Docker deployments in a scalable fashion. However, sometimes, we have no choice but to log in to the Docker host and look at the Docker containers themselves.

In this chapter, we will cover the following topics:

- Inspecting containers with `docker exec`
- Debugging from outside Docker
- Other debugging suites

### Inspecting containers

When troubleshooting servers, the traditional way to debug is to log in and poke around the machine. With Docker, this typical workflow is split into two steps: the first is logging in to the Docker host using standard remote access tools such as `ssh`, and the second is entering the desired running container's process namespace with `docker exec`. This is useful as a last resort to debug what is happening inside our application.

For most of this chapter, we will troubleshoot and debug a Docker container running HAProxy. To prepare this container, create a configuration file for HAProxy named `haproxy.cfg` with the following content:

```
defaults
  mode http
  timeout connect 5000ms
  timeout client 50000ms
```

```
timeout server 50000ms

frontend stats
  bind 127.0.0.1:80
  stats enable

listen http-in
  bind *:80
  server server1 www.debian.org:80
```

Next, using the official Docker image for HAProxy (`haproxy:1.5.14`), we will run the container together with the configuration we created earlier. Run the following command in our Docker host to start HAProxy with our prepared configuration:

```
dockerhost$ docker run -d -p 80:80 --name haproxy \
  -v `pwd`/haproxy.cfg:/usr/local/etc/haproxy/haproxy.cfg \
  haproxy:1.5.14
```

Now, we can begin inspecting our container and debugging it. A good first example is to confirm that the HAProxy container is listening to port 80. The `ss` program dumps a summary of sockets statistics available in most Linux distributions, such as our Debian Docker host. We can run the following command to display the statistics of the listening sockets inside our Docker container:

```
dockerhost$ docker exec haproxy /bin/ss -l
```

State	Recv-Q	Send-Q	Local Address:Port	Peer Address:Port
LISTEN	0	128	*:http	*:*
LISTEN	0	128	127.0.0.1:http	*:*

This approach with `docker exec` only worked because `ss` is included by default in the `debian:jessie` parent container of `haproxy:1.5.14`. We cannot use a similar tool that is not installed by default, such as `netstat`. Typing an equivalent `netstat` command will give the following error:

```
dockerhost$ docker exec haproxy /usr/bin/netstat -an
dockerhost$ echo $?
255
```

Let's investigate what happened by looking at the logs of Docker Engine Service. Typing the following command shows that the `netstat` program doesn't exist inside our container:

```
dockerhost$ journalctl -u docker.service -o cat
...
```

```

time="..." level=info msg="POST /v1.20/containers/haproxy/exec"
time="..." level=info msg="POST /v1.20/exec/c64fcf22b5c4.../start"
time="..." level=warning msg="signal: killed"
time="..." level=error msg="Error running command in existing..."
    " [8] System error: exec: \"/usr/bin/netstat\":"
    " stat /usr/bin/netstat: no such file or directory"
time="..." level=error msg="Handler for POST /exec/{n.../start..."
time="..." level=error msg="HTTP Error" err="Cannot run exec c..."
2015/11/18 17:58:12 http: response.WriteHeader on hijacked conn...
2015/11/18 17:58:12 http: response.Write on hijacked connect...
time="..." level=info msg="GET /v1.20/exec/c64fcf22b5c47be8278..."
...

```

An alternative way to find out whether `netstat` is installed in our system is to enter our container interactively. The `docker exec` command has the `-it` flags that we can use to spawn an interactive shell session to perform the debugging. Type the following command to use the `bash` shell to get inside our container:

```

dockerhost$ docker exec -it haproxy /bin/bash
root@b397ffb9df13:/#

```

Now that we are in a standard shell environment, we can debug with all the standard Linux utilities available inside our container. We will cover some of these commands in the next section. For now, let's take a look at why `netstat` doesn't work inside our container, as follows:

```

root@b397ffb9df13:/# netstat
bash: netstat: command not found
root@b397ffb9df13:/# /usr/bin/netstat -an
bash: /usr/bin/netstat: No such file or directory

```

As we can see here, `bash` is telling us that at this point, we have figured out that we don't have `netstat` installed through a more interactive debugging session.

We can provide a quick workaround by installing it inside our container, similar to what we do in a normal Debian environment. While we are still inside the container, we will type the following command to install `netstat`:

```

root@b397ffb9df13:/# apt-get update
root@b397ffb9df13:/# apt-get install -y net-tools

```

Now, we can run `netstat` successfully, as follows:

```
root@b397ffb9df13:/# netstat -an
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address   Foreign Address  State
tcp        0      0 0.0.0.0:80       0.0.0.0:*        LISTEN
tcp        0      0 127.0.0.1:80     0.0.0.0:*        LISTEN
Active UNIX domain sockets (servers and established)
Proto RefCnt Flags       Type        State         I-Node  Path
```

This approach of ad hoc container debugging is not recommended! We should have proper instrumentation and monitoring in place the next time we iterate on the design of our Docker infrastructure. Let's improve on what we initially built in *Chapter 4, Monitoring Docker Hosts and Containers*, next time! The following are some limitations of this last-resort approach:

1. When we stop and recreate the container, the `netstat` package we installed will not be available anymore. This is because the original HAProxy Docker image doesn't contain it in the first place. Installing ad hoc packages to run containers defeats the main feature of Docker, enabling an immutable infrastructure.
2. In case we want to package all the debugging tools inside our Docker image, its size will increase correspondingly. This means that our deployments will get larger and become slower. Remember that in the *Chapter 2, Optimizing Docker Images*, we optimized to reduce our container's size.
3. In the case of minimal containers with just the required binaries, we are now mostly blind. The `bash` shell is not even available! There is no way to enter our container; take a look at the following command:

```
dockerhost$ docker exec -it minimal_image /bin/bash
dockerhost$ echo $?
255
```

In summary, `docker exec` is a powerful tool to get inside our containers and debug by running various commands. Coupled with the `-it` flags, we can get an interactive shell to perform deeper debugging. This approach has limitations because it assumes that all the tools available inside our Docker container are ready to use.



More information about the `docker exec` command can be found in the official documentation at <https://docs.docker.com/reference/commandline/exec>.

The next section deals with how to go around this limitation by having tools from outside Docker inspect the state of our running container. We will provide a brief overview on how to use some of these tools.

## Debugging from the outside

Even though Docker isolates the network, memory, CPU, and storage resources inside containers, each individual container will still have to go to the Docker host's operating system to perform the actual command. We can take advantage of this trickling down of calls to the host operating system to intercept and debug our Docker containers from the outside. In this section, we will cover some selected tools and how to use them to interact with our Docker containers. We can perform the interaction from the Docker host itself or from inside a sibling container with elevated privileges to see some components of the Docker host.

## Tracing system calls

A **system call tracer** is one of the essential tools for server operations. It is a utility that intercepts and traces calls made by the application to the operating system. Each operating system has its own variation. Even if we run various applications and processes inside our Docker containers, it will eventually enter our Docker host's Linux operating system as a series of system calls.

On Linux systems, the `strace` program is used to trace these system calls. This interception and logging functionality of `strace` can be used to inspect our Docker containers from the outside. The list of system calls made throughout our container's lifetime can give a profile-level view on how it behaves.

To get started using `strace`, simply type the following command to install it inside our Debian Docker host:

```
dockerhost$ apt-get install strace
```



With the `--pid=host` option added to `docker run`, we can set a container's PID namespace to be of the Docker host's. This way, we'll be able to install and use `strace` inside a Docker container to inspect all the processes in the Docker host itself. We can also install `strace` from a different Linux distribution, such as CentOS or Ubuntu if we use the corresponding base image for our container.

More information describing this option is at <http://docs.docker.com/engine/reference/run/#pid-settings-pid>.



Now that we have `strace` installed in our Docker host, we can use it to inspect the system calls inside the HAProxy container we created in the previous section. Type the following commands to begin tracing the system calls from the `haproxy` container:

```
dockerhost$ PID=`docker inspect -f '{{.State.Pid}}' haproxy`
dockerhost$ strace -p $PID
epoll_wait(3, {}, 200, 1000)      = 0
epoll_wait(3, {}, 200, 1000)      = 0
epoll_wait(3, {}, 200, 1000)      = 0
epoll_wait(3, {}, 200, 1000)      = 0
...
```

As you can see, our HAProxy container makes `epoll_wait()` calls to wait for incoming network connections. Now, in a separate terminal, type the following command to make an HTTP request to our running container:

```
$ curl http://dockerhost
```

Now, let's go back to our running `strace` program earlier. We can see the following lines printed out:

```
...
epoll_wait(3, {}, 200, 1000)      = 0
epoll_wait(3, {{EPOLLIN, {u32=5, u64=5}}}, 200, 1000) = 1
accept4(5, {sa_family=AF_INET, sin_port=htons(56470), sin_addr...
setsockopt(6, SOL_TCP, TCP_NODELAY, [1], 4) = 0
accept4(5, 0x7ffc087a6a50, [128], SOCK_NONBLOCK) = -1 EAGAIN (...
recvfrom(6, "GET / HTTP/1.1\r\nUser-Agent: curl"..., 8192, 0, ...
socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) = 7
fcntl(7, F_SETFL, O_RDONLY|O_NONBLOCK) = 0
setsockopt(7, SOL_TCP, TCP_NODELAY, [1], 4) = 0
connect(7, {sa_family=AF_INET, sin_port=htons(80), sin_addr=in...
epoll_wait(3, {}, 200, 0)         = 0
sendto(7, "GET / HTTP/1.1\r\nUser-Agent: curl"..., 74, MSG_DON...
recvfrom(6, 0x18c488e, 8118, 0, 0, 0) = -1 EAGAIN (Resource ...
epoll_ctl(3, EPOLL_CTL_ADD, 7, {EPOLLOUT, {u32=7, u64=7}}) = 0...
epoll_ctl(3, EPOLL_CTL_ADD, 6, {EPOLLIN|EPOLLRDHUP, {u32=6, u6...
epoll_wait(3, {{EPOLLOUT, {u32=7, u64=7}}}, 200, 1000) = 1
sendto(7, "GET / HTTP/1.1\r\nUser-Agent: curl"..., 74, MSG_DON...
epoll_ctl(3, EPOLL_CTL_DEL, 7, 6bb1c) = 0
```

```

epoll_wait(3, {}, 200, 0) = 0
recvfrom(7, 0x18c88d4, 8192, 0, 0, 0) = -1 EAGAIN (Resource ...
epoll_ctl(3, EPOLL_CTL_ADD, 7, {EPOLLIN|EPOLLRDHUP, {u32=7, u6...
epoll_wait(3, {{EPOLLIN, {u32=7, u64=7}}}, 200, 1000) = 1
recvfrom(7, "HTTP/1.1 200 OK\r\nDate: Fri, 20 N"..., 8192, 0, ...
epoll_wait(3, {}, 200, 0) = 0
sendto(6, "HTTP/1.1 200 OK\r\nDate: Fri, 20 N"..., 742, MSG_DO...
epoll_wait(3, {{EPOLLIN|EPOLLRDHUP, {u32=6, u64=6}}}, 200, 100...
recvfrom(6, "", 8192, 0, NULL, NULL) = 0
shutdown(6, SHUT_WR) = 0
close(6) = 0
setsockopt(7, SOL_SOCKET, SO_LINGER, {onoff=1, linger=0}, 8) =...
close(7) = 0
epoll_wait(3, {}, 200, 1000) = 0
...

```

We can see here that HAProxy made standard BSD-style socket system calls, such as `accept4()`, `socket()`, and `close()`, to accept, process, and terminate network connections from our HTTP client. Finally, it goes back to `epoll_wait()` again to wait for the next connections. Also, take note that `epoll_wait()` calls are spread throughout the trace even while HAProxy processes a connection. This shows how HAProxy can handle concurrent connections.

Tracing system calls is a very useful technique to debug live production systems. People in operations sometimes get paged and don't have access to the source code right away. Alternatively, there are instances where we are only given compiled binaries (or plain Docker images) running in production where there is no source code (nor `Dockerfile`). The only clue we can get from a running application is to trap the system calls it makes to the Linux kernel.



The strace webpage can be found at <http://sourceforge.net/projects/strace/>. More information can be accessed through its man page as well by typing the following command:

```
dockerhost$ man 1 strace
```

For a more comprehensive list of system calls in Linux systems, refer to <http://man7.org/linux/man-pages/man2/syscalls.2.html>. This will be useful in understanding the various outputs given by strace.

## Analyzing network packets

Most Docker containers that we deploy revolve around providing some form of network service. In the example of HAProxy in this chapter, our container basically serves HTTP network traffic. No matter what kind of container we have running, the network packets will eventually have to get out of the Docker host for it to complete a request that we send it. By dumping and analyzing the content of these packets, we can gain some insight into the nature of our Docker container. In this section, we will use a packet analyzer called `tcpdump` to view the traffic of network packets being received and sent by our Docker containers.

To begin using `tcpdump`, we can issue the following command in our Debian Docker host to install it:

```
dockerhost$ apt-get install -y tcpdump
```



We can also expose the Docker host's network interfaces to a container. With this approach, we can install `tcpdump` in a container and not pollute our main Docker host with ad hoc debugging packages. This can be done by specifying the `--net=host` flag on `docker run`. With this, we can access the `docker0` interface from inside our Docker container with `tcpdump`.

The example of using `tcpdump` will be very specific to the Vagrant VMware Fusion provider for VMware Fusion 7.0. Assuming we have a Docker Debian host as a Vagrant VMware Fusion box, run the following command to suspend and unsuspend our Docker host's virtual machine:

```
$ vagrant suspend
$ vagrant up
$ vagrant ssh
dockerhost$
```

Now that we are back inside our Docker host, run the following command and note that we cannot resolve `www.google.com` anymore inside our interactive `debian:jessie` container, as follows:

```
dockerhost$ docker run -it debian:jessie /bin/bash
root@fce09c8c0e16:/# ping www.google.com
ping: unknown host
```

Now, let's run `tcpdump` in a separate terminal. While running the preceding ping command, we will notice the following output from our `tcpdump` terminal:

```
dockerhost$ tcpdump -i docker0
tcpdump: verbose output suppressed, use -v or -vv for full protocol
decode
listening on docker0, link-type EN10MB (Ethernet), capture size 262144
bytes
22:03:34.512942 ARP, Request who-has 172.17.42.1 tell 172.17.0.7, length 28
22:03:35.512931 ARP, Request who-has 172.17.42.1 tell 172.17.0.7, length 28
22:03:38.520681 ARP, Request who-has 172.17.42.1 tell 172.17.0.7, length 28
22:03:39.520099 ARP, Request who-has 172.17.42.1 tell 172.17.0.7, length 28
22:03:40.520927 ARP, Request who-has 172.17.42.1 tell 172.17.0.7, length 28
22:03:43.527069 ARP, Request who-has 172.17.42.1 tell 172.17.0.7, length 28
```

As we can see, the interactive `/bin/bash` container is looking for `172.17.42.1`, which is normally the IP address attached to the Docker Engine network device, `docker0`. With this figured out, take a look at `docker0` by typing the following command:

```
dockerhost$ ip addr show dev docker0
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:42:46:66:64:b8 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::42:46ff:fe66:64b8/64 scope link
        valid_lft forever preferred_lft forever
```

Now, we can view the problem. The `docker0` device doesn't have an IPv4 address attached to it. Somehow, VMware unsuspending our Docker host removes the mapped IP address in `docker0`. Fortunately, the solution is to simply restart the Docker Engine, and Docker will reinitialize the `docker0` network interface by itself. Restart Docker Engine by typing the following command in our Docker host:


```
dockerhost$ systemctl restart docker.service
```

Now, when we run the same command as earlier, will see that the IP address is attached, as follows:

```
dockerhost$ ip addr show dev docker0
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noque...
    link/ether 02:42:46:66:64:b8 brd ff:ff:ff:ff:ff:ff
    inet 172.17.42.1/16 scope global docker0
        valid_lft forever preferred_lft forever
    inet6 fe80::42:46ff:fe66:64b8/64 scope link
        valid_lft forever preferred_lft forever
```

Let's go back to our initial command showing the problem; we will see that it is now solved, as follows:

```
root@fce09c8c0e16:/# ping www.google.com
PING www.google.com (74.125.21.105): 56 data bytes
64 bytes from 74.125.21.105: icmp_seq=0 ttl=127 time=65.553 ms
64 bytes from 74.125.21.105: icmp_seq=1 ttl=127 time=38.270 ms
...
```

[  More information about the `tcpdump` packet dumper and analyzer can be found at <http://www.tcpdump.org>. We can also access the documentation from where we installed it by typing the following command:  
`dockerhost$ man 8 tcpdump` ]

## Observing block devices

Data being accessed from our Docker containers will mostly reside in physical storage devices, such as hard disks or solid state drives. Underneath Docker's copy-on-write filesystems is a physical device that is randomly accessed. These drives are grouped together as block devices. Data here is randomly accessed fixed-size data called *blocks*.

So, in case our Docker containers have peculiar I/O behavior and performance issues, we can trace and troubleshoot what is happening inside our block devices using a tool called `blktrace`. All events the kernel generates to interact with the block devices from processes are intercepted by this program. In this section, we will set up our Docker host to observe the block device supporting our containers underneath.

To use `blktrace`, let's prepare our Docker host by installing the `blktrace` program. Type the following command to install it inside our Docker host:

```
dockerhost$ apt-get install -y blktrace
```

In addition, we need to enable the debugging of the filesystem. We can do this by typing the following command in our Docker host:

```
dockerhost$ mount -t debugfs debugfs /sys/kernel/debug
```

After the preparations, we need to figure out how to tell `blktrace` where to listen for I/O events. To trace I/O events for our containers, we need to know where the root of the Docker runtime is. In the default configuration of our Docker host, the runtime points to `/var/lib/docker`. To figure out which partition it belongs to, type the following command:

```
dockerhost$ df -h
```

Filesystem	Size	Used	Avail	Use%	Mounted on
/dev/dm-0	9.0G	7.6G	966M	89%	/
udev	10M	0	10M	0%	/dev
tmpfs	99M	13M	87M	13%	/run
tmpfs	248M	52K	248M	1%	/dev/shm
tmpfs	5.0M	0	5.0M	0%	/run/lock
tmpfs	248M	0	248M	0%	/sys/fs/cgroup
/dev/sda1	236M	34M	190M	15%	/boot

As described in the preceding output, our Docker host's `/var/lib/docker` directory is under the `/` partition. This is where we will point `blktrace` to listen for events from. Type the following command to start listening for I/O events on this device:

```
dockerhost$ blktrace -d /dev/dm-0 -o dump
```



Using the `--privileged` flag in `docker run`, we can use `blktrace` within a container. Doing so will allow us to mount the debugged filesystem with the increased privileges.

More information on extended container privileges can be found at <https://docs.docker.com/engine/reference/run/#runtime-privilege-linux-capabilities-and-lxc-configuration>.

To create a simple workload that will generate I/O events in our disk, we will create an empty file from a container until the / partition runs out of free space. Type the following command to generate this workload:

```
dockerhost$ docker run -d --name dump debian:jessie \  
    /bin/dd if=/dev/zero of=/root/dump bs=65000
```

Depending on the free space available in our root partition, this command may finish quickly. Right away, let's get the PID of the container we just ran using the following command:

```
dockerhost$ docker inspect -f '{{.State.Pid}}' dump  
11099
```

Now that we know the PID of our Docker container that generated I/O events, we can look this up with the `blktrace` program's complementary tool, `blkparse`. The `blktrace` program only listens for the events in the Linux kernel's block I/O layer and dumps the results on a file. The `blkparse` program is the accompanying tool to view and analyze the events. In the workload we generated earlier, we can look for the I/O events that correspond to our Docker container's PID using the following command:

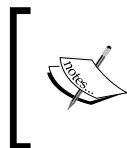
```
dockerhost$ blkparse -i dump.blktrace.0 | grep --color " $PID "  
...  
254,0    0      730    10.6267 11099  Q   R 13667072 + 24 [exe]  
254,0    0      732    10.6293 11099  Q   R 5042728 + 16 [exe]  
254,0    0      734    10.6299 11099  Q   R 13900768 + 152 [exe]  
254,0    0      736    10.6313 11099  Q  RM 4988776 + 8 [exe]  
254,0    0     1090    10.671 11099  C  W 11001856 + 1024 [0]  
254,0    0     1091    10.6712 11099  C  W 11002880 + 1024 [0]  
254,0    0     1092    10.6712 11099  C  W 11003904 + 1024 [0]  
254,0    0     1093    10.6712 11099  C  W 11004928 + 1024 [0]  
254,0    0     1094    10.6713 11099  C  W 11006976 + 1024 [0]  
254,0    0     1095    10.6714 11099  C  W 11005952 + 1024 [0]  
254,0    0     1138    10.6930 11099  C  W 11239424 + 1024 [0]  
254,0    0     1139    10.6931 11099  C  W 11240448 + 1024 [0]  
...
```

In the preceding highlighted output, we can see that the `/dev/dm-0` block offset the position of 11001856, and there was a writing (w) of 1024 bytes of data that just completed (c). To probe further, we can look at this offset position on the events that it generated. Type the following command to filter out this offset position:

```
dockerhost$ blkparse -i dump.blktrace.0 | grep 11001856
...
254,0 0 1066 10.667 8207 Q W 11001856 + 1024 [kworker/u2:2]
254,0 0 1090 10.671 11099 C W 11001856 + 1024 [0]
...
```

We can see the write (w) being queued (Q) to our device by the `kworker` process, which means the write was queued by the kernel. After 40 milliseconds, the write request registered was completed for our Docker container process.

The debugging walkthrough we just performed is just a small sample of what we can do by tracing block I/O events with `blktrace`. For example, we can also probe our Docker container's I/O behavior in greater detail and figure out the bottlenecks that are happening to our application. Are there a lot of writes being made? Are the reads so much that they need caching? Having the actual events rather than only the performance metrics provided by the built-in `docker stats` command is helpful in very deep troubleshooting scenarios.



More information on the different output values of `blkparse` and flags to capture I/O events in `blktrace` can be found in the user guide located at <http://www.cse.unsw.edu.au/~aaronc/iosched/doc/blktrace.html>.

## A stack of troubleshooting tools

Debugging applications inside Docker containers required a different approach from normal applications in Linux. However, the actual programs being used are the same because all the calls from inside the container will eventually go to the Docker host's kernel operating system. By knowing how calls go outside of our containers, we can use any other debugging tools we have to troubleshoot.



In addition to standard Linux tools, there are several container-specific utilities that package the preceding standard utilities to be more friendly for container usage. The following are some of these tools:

- Red Hat's `rhel-tools` Docker image is a huge container containing a combination of the tools we discussed earlier. Its documentation page at [https://access.redhat.com/documentation/en/red-hat-enterprise-linux-atomic-host/version-7/getting-started-with-containers/#using\\_the\\_atomic\\_tools\\_container\\_image](https://access.redhat.com/documentation/en/red-hat-enterprise-linux-atomic-host/version-7/getting-started-with-containers/#using_the_atomic_tools_container_image) shows how to run it with the proper Docker privileges for it to function correctly.
- The CoreOS `toolbox` program is a small script utility that creates a small Linux container using Systemd's `systemd-nspawn` program. By copying the root filesystem from popular Docker images, we can install any tool we want without polluting the Docker host's filesystem with ad hoc debugging tools. Its use is documented on its webpage at <https://coreos.com/os/docs/latest/install-debugging-tools.html>.
- The `nsenter` program is a utility to enter a Linux control group's process namespace. It is the predecessor to the `docker exec` program and is considered unmaintained. To get a history of how `docker exec` came to be, visit the `nsenter` program's project page at <https://github.com/jpetazzo/nsenter>.

## Summary

Remember that logging in to Docker hosts isn't scalable. Adding instrumentation at the application level, in addition to the ones given by our operating system, helps in faster and more efficient diagnosing of the problems that we may encounter in the future. Remember, nobody likes waking up at two in the morning to run `tcpdump` to debug a Docker container on fire!

In the next chapter, we will wrap up and look again at what it takes to get our Docker-based workloads to production.

# 8

## Onto Production

Docker came out of dotCloud's PaaS, where it fulfils the needs of IT to develop and deploy web applications in a fast and scalable manner. This is needed to keep up with the ever-accelerating pace of using the Web. Keeping everything running in our Docker container in production is no simple feat.

In this chapter, we will wrap up what you learned about optimizing Docker and illustrate how it relates to operating our web applications in production. It consists of the following topics:

- Performing web operations
- Supporting our application with Docker
- Deploying applications
- Scaling applications
- Further reading on web operations in general

### Performing web operations

Keeping a web application running 24/7 on the Internet poses challenges in both software development and systems administration. Docker positions itself as the glue that allows both disciplines to come together by creating Docker images that can be built and deployed in a consistent manner.

However, Docker is not a silver bullet to the Web. It is still important to know the fundamental concepts in software development and systems administration as web applications become more complex. The complexity naturally arises because these days, with Internet technologies in particular, the multitude of web applications is becoming more ubiquitous in people's lives.

Dealing with the complexity of keeping web applications up and running involves mastering the ins and outs of web operations, and like any road to mastery, Theo Schlossnagle boils it down to four basic pursuits: knowledge, tools, experience, and discipline. *Knowledge* refers to absorbing information about web operations available on the Internet, in conferences, and technology meetings like a sponge. Understanding them and knowing how to filter out the signal from the noise will aid us in designing our application's architecture when they burn in production. With Docker and Linux containers increasing in popularity, it is important to be aware of the different technologies that support it and dive into its basics. In *Chapter 7, Troubleshooting Containers*, we showed that regular Linux debugging tools are still useful in debugging running Docker containers. By knowing how containers interact with our Docker host's operating system, we were able to debug the problems occurring in Docker.

The second aspect is mastering our *tools*. This book basically revolved around mastering the use of Docker by looking at how it works and how to optimize its usage. In *Chapter 2, Optimizing Docker Images*, we learned how to optimize Docker images based on how Docker builds the images and runs the container using its copy-on-write filesystem underneath. This was guided by our knowledge of web operations on why optimized Docker images are important both from a scalability and deployability standpoint. Knowing how to use Docker effectively does not happen overnight. Its mastery can only be gained by a continuous practice of using Docker in production. Sure, we might be paged at 2 am for our first Docker deployment in production, but as time goes by, the experience we gain from continuous usage will make Docker an extension of our limbs and senses, as Schlossnagle puts it.

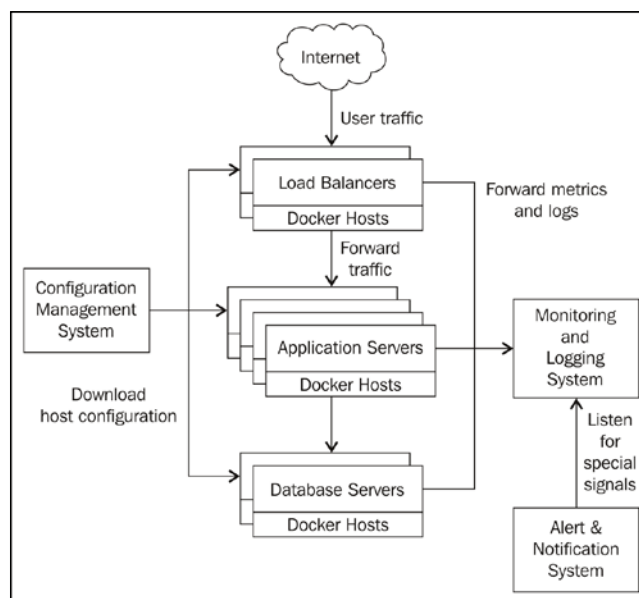
By applying the knowledge and continuously using our tools, we gain *experience* that we can draw upon in the future. This aids us in making good judgments based on bad decisions that we made in the past. It is the place where we can see the theory of container technology and the practice of running Docker in production collide. Schlossnagle mentioned the challenges of acquiring experience in web operations and how to survive the bad judgments and draw experiences from them. He suggests having limited environments in which a bad decision's impact is minimal. Docker is the best place to draw these types of experiences. Having a standard format of ready-to-deploy Docker images, junior web operations engineers can have their own environments that they can experiment with and learn from their mistakes in. Also, since Docker environments look very similar when they move forward to production, these engineers will already have their experience to draw upon.

The last part in the pursuit of mastering web operations is *discipline*. However, as it is a very young discipline, such processes are not well defined. Even with Docker, it took a few years for people to realize the best ways to use container technologies. Before this, the convenience of including the whole kitchen sink in Docker images was very common. However, as we can see in *Chapter 2, Optimizing Docker Images*, reducing the footprint of Docker images helps aid in managing the complexity of the applications that we have to debug. This makes the experience of debugging in *Chapter 7, Troubleshooting Containers*, much simpler because we have fewer components and factors to think about. These disciplines of using Docker do not come overnight just by reading Docker blogs (well, some do). It involves continuous exposure to the knowledge of the Docker community and the practice of using Docker in various settings for production use.

In the remaining sections, we will show how the theory and practice of using Docker's container technology can aid in the operation of our web applications.

## Supporting web applications with Docker

The following diagram shows the typical architecture of a web application. We have the load balancer tier that receives traffic from the Internet and then the traffic, which is typically composed of user requests, is relayed to a farm of web application servers in a load-balanced fashion. Depending on the nature of the request, some states will be grabbed by the web application from the persistent storage tier, similar to database servers:



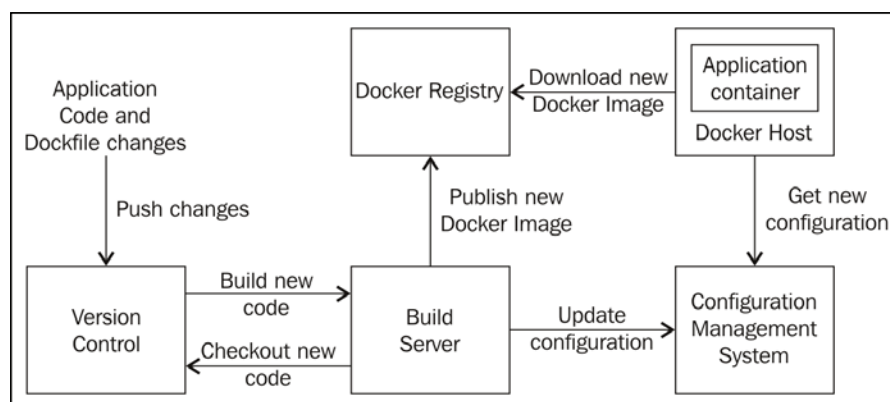
As we can see in the preceding diagram, each tier is run inside a Docker container on top of a Docker host. With this layout for each component, we can take advantage of Docker's uniform way of deploying load balancers, applications, and databases, as we did in *Chapter 2, Optimizing Docker Images*, and *Chapter 6, Load Balancing*. However, in addition to the Docker daemons in each Docker host, we need supporting infrastructure to manage and observe the whole stack of our web architecture in a scalable fashion. On the right-hand side, we can see that each of our Docker hosts sends diagnostic information—for example, application and system events such as log messages and metrics—to our centralized logging and monitoring system. We deployed such a system in *Chapter 4, Monitoring Docker Hosts and Containers*, where we rolled out Graphite and an ELK stack. In addition, there might be another system that listens for specific signals in the logs and metrics and sends alerts to the engineers responsible for the operation of our Docker-based web application stack. These events can relate to critical events, such as the availability and performance of our application, that we need to take action on to ensure that our application is fulfilling the needs of our business as expected. An internally managed system, such as Nagios, or a third-party one, such as PagerDuty, is used for our Docker deployments to call and wake us up at 2 am for deeper monitoring and troubleshooting sessions as in *Chapter 4, Monitoring Docker Hosts and Containers*, and *Chapter 7, Troubleshooting Containers*.

The left-hand side of the diagram contains the configuration management system. This is the place where each of the Docker hosts downloads all the settings it needs to function properly. In *Chapter 3, Automating Docker Deployments with Chef*, we used a Chef server to store the configuration of our Docker host. It contained information such as a Docker host's role in our architecture's stack. The Chef server stores information on which Docker containers to run in each tier and how to run them using the Chef recipes we wrote. Finally, the configuration management system also tells our Docker hosts where the Graphite and Logstash monitoring and logging endpoints are.

All in all, it takes various components to support our web application in production aside from Docker. Docker allows us to easily set up this infrastructure because of the speed and flexibility of deploying containers. Nonetheless, we shouldn't skip doing our homework about having these supporting infrastructures in place. In the next section, we will see the supporting infrastructure of deploying web applications in Docker using the skills you learned in the previous chapters.

## Deploying applications

An important component when tuning the performance of Docker containers is the feedback telling us that we were able to improve our web application correctly. The deployment of Graphite and the ELK stack in *Chapter 4, Monitoring Docker Hosts and Containers*, gave us visibility on the effects of what we changed in our Docker-based web application. As much as it is important to gather feedback, it is more important to gather feedback in a timely manner. Therefore, the deployment of our Docker containers needs to be in a fast and scalable manner. Being able to configure a Docker host automatically, as we did in *Chapter 3, Automating Docker Deployments with Chef*, is an important component for a fast and automated deployment system. The rest of the components are described in the following diagram:

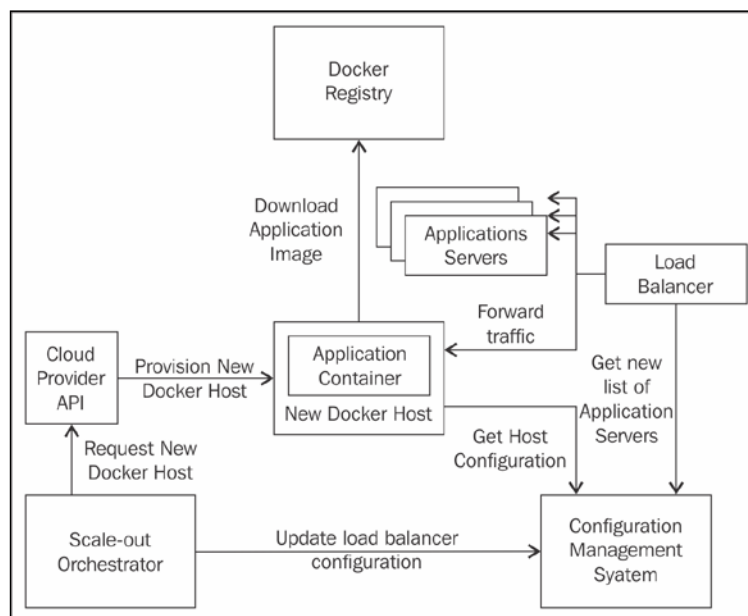


Whenever we submit changes to our application's code or the `Dockerfile` describing how it is run and built, we need supporting infrastructure to propagate this change all the way to our Docker hosts. In the preceding diagram, we can see that the changes we submit to our version control system, such as Git, generate a trigger to build the new version of our code. This is usually done through Git's postreceive hooks in the form of shell scripts. The triggers will be received by a build server, such as Jenkins. The steps to propagate the change will be similar to the blue-green deployment process we made in *Chapter 6, Load Balancing*. After receiving the trigger to build the new changes we submitted, Jenkins will take a look at the new version of our code and run `docker build` to create the Docker image. After the build, Jenkins will push the new Docker image to a Docker registry, such as Docker Hub, as we set up in *Chapter 2, Optimizing Docker Images*. In addition, it will update the target Docker hosts indirectly by updating the entry in the Chef server configuration management system we laid out in *Chapter 3, Automating Docker Deployments with Chef*. With the artifacts of changes available in the Chef server and Docker registry, our Docker host will now notice the new configuration and download, deploy, and run the new version of our web application inside a Docker container.

In the next section, we will discuss how a similar process is used to scale out our Docker application.

## Scaling applications

When we receive alerts from our monitoring system, as in *Chapter 4, Monitoring Docker Hosts and Containers*, that the pool of Docker containers running our web application is not loaded, it is time to scale out. We accomplished this using load balancers in *Chapter 6, Load Balancing*. The following diagram shows the high-level architecture of the commands we ran in *Chapter 6, Load Balancing*:



Once we decide to scale out and add an additional Docker host, we can automate the process with a scale-out orchestrator component. This can be a series of simple shell scripts that we will install inside a build server, such as Jenkins. The orchestrator will basically ask the cloud provider API to create a new Docker host. This request will then provision the Docker host and run the initial bootstrap script to download the configuration from our configuration management system in *Chapter 3, Automating Docker Deployments with Chef*. This will automatically set up the Docker host to download our application's Docker image from the Docker registry. After this whole provisioning process is finished, our scale-out orchestrator will then update the load balancer in our Chef server with the new list of application servers to forward traffic to. So, the next time the `chef-client` inside our load balancer Docker host polls the Chef Server, it will add the new Docker host and start forwarding traffic to it.

As we can note, learning the way to automate setting up our Docker host in *Chapter 3, Automating Docker Deployments with Chef*, is crucial to realizing the scalable load balancing architecture setup we did in *Chapter 6, Load Balancing*.

## Further reading

The supporting architecture to help our web applications use Docker is nothing but a scratch on the surface. The fundamental concepts in this chapter are described in greater detail in the following books:

- *Web Operations: Keeping the Data On Time*, which is edited by J. Allspaw and J. Robbins. 2010 O'Reilly Media.
- *Continuous Delivery*, by J. Humble and D. Farley. 2010 Addison-Wesley.
- *Jenkins: The Definitive Guide*, J. F. Smart. 2011 O'Reilly Media.
- *The Art of Capacity Planning: Scaling Web Resources*, J. Allspaw. 2008 O'Reilly Media.
- *Pro Git*, S. Chacon and B. Straub. 2014 Apress.

## Summary

You learned a lot about how Docker works throughout this book. In addition to the basics of Docker, we looked back at some fundamental concepts of web operations and how it helps us realize the full potential of Docker. You gained knowledge of key Docker and operating systems concepts to get a deeper understanding of what is happening behind the scenes. You now have an idea of how our application goes from our code down to the actual call in the operating system of our Docker host. You learned a lot about the tools to deploy and troubleshoot our Docker containers in production in a scalable and manageable fashion.

However, this should not stop you from continuing to develop and practice using Docker to run our web applications in production. We should not be afraid to make mistakes and gain further experience on the best ways to run Docker in production. As the Docker community evolves, so do these practices through the collective experience of the community. So, we should continue and be disciplined in learning the fundamentals we started to master little by little. Don't hesitate to run Docker in production!





# Index

## A

### **Amazon EC2 Container Service**

URL 59

### **Ansible**

URL 58

### **Apache HTTP Server**

URL 114

### **Apache JMeter**

installing 87, 88

page, URL 91

sample application, deploying 84-87

setting up 84

test plan, creating 89-91

test plan elements, URL 91

URL 84

### **applications**

deploying 133

scaling 134, 135

### **apt-cacher-ng 32**

### **Azure Docker VM Extension**

URL 59

## B

### **benchmark, results**

analyzing 92

JMeter runs, result viewing 92

response time, plotting 94, 95

throughput, calculating 92-94

### **benchmark, tools**

Apache Bench 102

HP Lab's Httperf 102

Siege 102

### **benchmark, tuning**

about 99

concurrency, increasing 99

distributed tests, running 100, 101

### **benchmark workload**

building 89

test plan, creating in JMeter 89-91

### **blktrace**

URL 127

### **block devices**

observing 124-126

### **blue-green deployments 110**

### **build context size**

reducing 28, 29

## C

### **cAdvisor**

URL 81

### **CFEngine**

URL 58

### **Chef**

Bootstrap nodes 48-50

cookbooks, URL 52

development kit, URL 47

recipe, URL 52

signing up, for server 44, 45

supermarket, URL 50

URL 43

using 43

workstation, setting up 46, 47

### **CloudFoundry's GoRouter**

URL 114

### **collectd**

monitoring with 68

running, inside Docker 74

URL 72

**configuration management** 41, 42

**Constant Throughput Timer** 100

**container privileges**

URL 125

**containers**

inspecting 115-118

**Continuous Delivery book** 114

**CoreOS fleet**

URL 59

**CoreOS toolbox program**

URL 128

## **D**

**dashboard**

URL 69

**Datadog**

URL 82

**debugging**

from outside 119

**Docker applications**

scaling out 108-110

zero downtime, deploying with 110-113

**docker build command**

URL 4

**Docker Compose**

URL 87

**Docker container logs**

forwarding 79-81

**Docker containers**

deploying 54-58

interactive containers 12-14

linking 11, 12

ports, exposing 7-9

ports, publishing 9

running 7

**Docker containers ports, publishing**

--publish-all flag 9, 10

--publish flag 10

**docker exec command**

URL 118

**Dockerfile**

URL 22

**Docker host**

configuring 50-54

preparing 1, 2

URL 2

**Docker host farm**

preparing 103, 104

**Docker Hub account**

URL 4

**dockerignore files**

URL 30

**Docker images**

building 3, 4

build time, improving 19

deployment time, reducing 16-18

pulling, from repository 6, 7

pushing, to repository 4-6

URL 3

working with 2

**Docker images, build time**

build context size, reducing 28, 29

caching proxies, using 30-32

image layers, reusing 22-28

improving 19

registry mirrors, using 19-21

**Docker images, size**

build and deployment images,

separating 35-38

commands, chaining 33, 34

reducing 33

**Docker machine**

URL 58

**Docker Swarm**

URL 59

**dotCloud's Hipache**

URL 114

## **E**

**Elastic's Found**

URL 82

**ELK stack**

Elasticsearch 75

Kibana 75

logs, consolidating 74-78

Logstash 74

## **F**

**Fluentd**

URL 81

## G

### Google Container Engine

URL 59

### Google Kubernetes

URL 59

### Graphite

carbon-cache component 63  
carbon-cache component, deploying 64-66  
graphite-web component 63  
in production 67  
metrics, collecting 63, 64  
performance, observing 95-99  
scaling, URL 67  
URL 64  
web settings, URL 66  
whisper component 63  
whisper database, deploying 64-66

### Graylog

URL 81

## H

### HAProxy

URL 114

## I

### image layers

reusing 22-28

### InfluxDB

URL 81

## J

### Java Runtime Environment (JRE) 88

### Joyent Elastic Container Service

URL 59

## K

### Kibana

performance, observing 95-99

### knife bootstrap

URL 49

## L

### Librato

URL 82

### linked containers

URL 11

### load balancers 114

### load balancing

with Nginx 105-108

### logging

solutions 81, 82

## M

### Mesosphere Marathon

URL 59

### monitoring

collectd, running inside Docker 74  
Docker-related data, collecting 71-73  
importance 62  
solutions 81, 82  
with collectd 68-70

## N

### network packets

analyzing 122-124

### New Relic

URL 82

### Nginx

load balancing with 105-107

### nsenter

URL 128

## P

### pid option

URL 119

### Polipo

URL 32

### proxy\_pass

URL 106

### Puppet

URL 58

## R

### Redx

URL 114

### registry

URL 18

### registry mirrors

URL 21

using 19-21

### remote testing

URL 101

### repository

Docker images, pulling from 6, 7

Docker images, pushing to 4-6

### rhel-tools

URL 128

## S

### SaltStack

URL 58

### Sensu

URL 81

### server

URL 106

### SmartDataCenter Docker Engine

URL 59

### Sonatype Nexus

URL 32

### Splunk

URL 81

### Splunk Cloud

URL 82

### Squid

URL 32

### strace

URL 121

### system calls

URL 121

### system call tracer 119-121

## T

### tcpdump packet dumper

URL 124

### Treasure Data

URL 82

### troubleshooting tools

stack 127

## U

### Unix signals

URL 109

### upstream

URL 106

## V

### Vulcand

URL 114

## W

### webapp

URL 3

### web applications

deploying 133

scaling 134, 135

supporting, with Docker 131, 132

### web operations

performing 129-131

web application, performing 129-131