



Community Experience Distilled

Build Your Own PaaS with Docker

Create, modify, and run your own PaaS with modularized containers using Docker

Oskar Hane

[PACKT]
PUBLISHING

Build Your Own PaaS with Docker

Create, modify, and run your own PaaS with
modularized containers using Docker

Oskar Hane



BIRMINGHAM - MUMBAI

Build Your Own PaaS with Docker

Copyright © 2015 Packt Publishing

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

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

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

First published: April 2015

Production reference: 1010415

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

ISBN 978-1-78439-394-6

www.packtpub.com

Credits

Author

Oskar Hane

Project Coordinator

Neha Bhatnagar

Reviewers

Donald Simpson

Lawrence Taylor

Proofreaders

Ting Baker

Simran Bhogal

Commissioning Editor

Sarah Crofton

Indexer

Mariammal Chettiar

Acquisition Editor

Rebecca Youe

Production Coordinator

Manu Joseph

Content Development Editor

Merwyn D'Souza

Cover Work

Manu Joseph

Technical Editors

Narsimha Pai

Mahesh Rao

Copy Editors

Dipti Kapadia

Vikrant Phadke

About the Author

Oskar Hane is a full stack developer, with 15 years of experience in the development and deployment of web applications. During this period, he mostly worked with start-ups and small, fast-moving companies. He is the cofounder of several companies and has been working as an independent contractor for the past few years. These days, Oskar works with Neo4j, the world's leading graph database, where he spends most of his time on the frontend, writing JavaScript.

He lives in Sweden with his wife and daughter. He enjoys programming as well as all kinds of sports and outdoor activities, such as hunting and fishing.

About the Reviewers

Donald Simpson is an experienced build manager, software developer, and information technology consultant based in Scotland, UK.

He specializes in helping organizations improve the quality and reduce the cost of software development through the adoption of continuous integration and continuous delivery best practices.

He has designed and implemented fully automated code and environment build solutions for a range of clients and Agile projects.

You can find out more about Donald on his website (www.donaldsimpson.co.uk).

Lawrence Taylor is armed with a PhD in mathematics. He has 7 years of experience in developing software in a variety of sectors, from finance to travel. Charred by his number-theoretic past, he is drawn to the abstractions and techniques required to design and build extensible software systems.

www.PacktPub.com

Support files, eBooks, discount offers, and more

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

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

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



<https://www2.packtpub.com/books/subscription/packtlib>

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

Why subscribe?

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

Free access for Packt account holders

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

Table of Contents

Preface	v
Chapter 1: Installing Docker	1
What is Docker?	1
Docker on Ubuntu Trusty 14.04 LTS	2
Upgrading Docker on Ubuntu Trusty 14.04 LTS	3
User permissions	3
Docker on Mac OS X	3
Installation	3
Upgrading Docker on Mac OS X	6
Docker on Windows	6
Installation	6
Upgrading Docker on Windows	7
Docker on Amazon EC2	7
Installation	8
Open ports	12
Upgrading Docker on Amazon EC2	12
User permissions	12
Displaying Hello World	13
Summary	14
Chapter 2: Exploring Docker	15
The Docker image	15
The Docker container	16
The Docker command-line interface	17
The Docker Registry Hub	19
Browsing repositories	19
Exploring published images	21
Summary	27

Table of Contents

Chapter 3: Creating Our First PaaS Image	29
The WordPress image	29
Moving from the defaults	31
Our objective	32
Preparing for caching	32
Raising the upload limit	34
Plugin installation	36
Making our changes persist	43
Hosting image sources on GitHub	44
Publishing an image on the Docker Registry Hub	46
Automated builds	47
Summary	50
Chapter 4: Giving Containers Data and Parameters	51
Data volumes	51
Mounting a host directory as a data volume	52
Mounting a data volume container	52
Backing up and restoring data volumes	53
Creating a data volume image	53
Data volume images	54
Exposing mount points	54
The Dockerfile	55
Hosting on GitHub	55
Publishing on the Docker Registry Hub	57
Running a data volume container	58
Passing parameters to containers	59
Creating a parameterized image	59
Summary	62
Chapter 5: Connecting Containers	63
Manually connecting containers	63
Exploring the contents of a data volume container	65
Connecting containers using Docker Compose	67
Installing Docker Compose	67
Basic Docker Compose commands	68
Service	68
Using the run command	69
Using the scale command	69
Setting up our PaaS with Docker Compose	69
Connecting containers using Crane	70
Installing Crane	71

Table of Contents

Usage	71
Configuration	71
Summary	75
Chapter 6: Reverse Proxy Requests	77
Explaining the problem	78
Finding a solution	78
Implementing the solution	80
Implementation with HAProxy	81
Installing HAProxy	81
Configuring HAProxy	82
Adding more domains to HAProxy	85
Implementation with Nginx	86
Installing Nginx	87
Configuring Nginx	87
Adding more domains to Nginx	89
Automating the process of mapping domains	90
Summary	91
Chapter 7: Deployment on Our PaaS	93
The problem with our current setup	93
The tools/services available	94
Dokku – Docker-powered mini-Heroku	96
Installation	96
Creating a sample Dokku app	97
How Dokku works	100
The receive process	100
Dokku plugins	103
Dokku domains plugin	103
Dokku-docker-options	103
Volume plugin for Dokku	103
Dokku-link	104
MariaDB plugin for Dokku	104
Setting up a WordPress app with Dokku	104
Starting multiple apps	107
Adding a domain to Dokku	108
More notes on Dokku	109
Summary	110
Chapter 8: What's Next?	111
What is a Twelve-Factor app?	111
Flynn	113
Deis	114

Table of Contents

Rocket	115
Orchestration tools	116
Summary	116
Index	117

Preface

Docker is an open source project with a high-level API that provides software containers to run processes in isolation. Packaging an app in a container that can run on any Linux server (as well as on OS X and Windows) helps developers focus on developing the app instead of server setups and other DevOps operations.

What this book covers

Chapter 1, Installing Docker, takes you through the Docker installation process to start a container.

Chapter 2, Exploring Docker, gives you an insight into how Docker works and the terminology used and introduces public images.

Chapter 3, Creating Our First PaaS Image, shows you how to create your own custom Docker image that will be a part of your PaaS.

Chapter 4, Giving Containers Data and Parameters, teaches you about the data storing alternatives available and how to pass parameters to your PaaS containers.

Chapter 5, Connecting Containers, shows you how to manually connect containers in order to form a complete platform, and introduces two tools that give you more control over multicontainer platforms.

Chapter 6, Reverse Proxy Requests, explains the problem and provides a solution to having multiple containers on the same host, where more than one host should be reachable on the same port.

Chapter 7, Deployment on Our PaaS, takes you through the process of deploying code to your PaaS. Here, you learn how to create your own mini-Heroku with Dokku.

Chapter 8, What's Next?, introduces a few projects that are in their early stages and look promising for the future of a Docker based PaaS.

What you need for this book

- A PC/laptop running OS X, Linux, or Windows
- Internet connection

Who this book is for

This book is intended for those who want to learn how to take full advantage of separating services into module containers and connecting them to form a complete platform. You may have, perhaps, heard of Docker but never installed or used it; or, you may have installed it and run a full stack container, not separating services in module containers that connect. In either case, this book will give you all the insights and knowledge required to run your own PaaS.

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:
"After some dependent images are downloaded, we should be able to see our running container when we execute `docker ps`."

A block of code is set as follows:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Hello</title>
</head>
<body>
<h1>First edit!</h1>
</body>
</html>
```

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

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

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: "Open the **Finder** window and navigate to your Applications folder; locate **boot2docker** and double-click on it."



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, 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.

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 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, and we will do our best to address the problem.

1

Installing Docker

In this chapter, we will find out where to download and how to install Docker on various operating systems. Some basic Docker commands will be used so that we can verify whether the installation was successful and to interact with Docker for the very first time.

The following topics are covered in this chapter:

- What is Docker?
- Docker on Ubuntu Trusty 14.04 LTS
- Docker on Mac OS X
- Docker on Windows
- Docker on Amazon EC2

This book will take you through all the steps, from installing Docker to running your own **Platform as a Service (PaaS)** so that you can push your code without having to think about infrastructure or server provisioning.

The theme of this book will be to create a modular web application using an isolated web server and a database.

What is Docker?

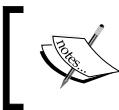
On Docker's website, <http://www.docker.com>, the following definition is provided for Docker:

"Docker is an open platform for developers and sysadmins to build, ship, and run distributed applications."

What this means in a more practical sense is that Docker is a way of enclosing services in isolated environments, called **containers**, so that they can be packaged with all they need in terms of libraries and dependencies and the developer can be certain that the service will run wherever Docker runs.

Docker on Ubuntu Trusty 14.04 LTS

The OS, flavor and version, where it's easiest to install Docker is in Ubuntu Trusty 14.04 LTS. This is a pretty quick task since we can use the built-in package manager `apt-get`.



Note that Docker is called `docker.io` here and just `docker` on other platforms since Ubuntu (and Debian) already has a package named `docker`.



First we open a terminal and execute these commands one by one:

```
sudo apt-get update  
sudo apt-get install docker.io  
source /etc/bash_completion.d/docker.io
```

Here, we first update the lists of the packet manager `apt-get` in order to get information about all the packages, versions, and dependencies that are available. The next line actually installs Docker, and after that, we enable Ubuntu to tab-complete our Docker commands.

When you've done this without errors, run `sudo docker.io version` just to verify that it works as expected.



Note that this installs the latest released Ubuntu package version, which might not necessarily be the latest released Docker version.



In order to have the latest version from an alternative Docker-maintained repository, we can execute the following command:

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

This adds an alternative repository maintained by the Docker team and installs Docker for you as a much more updated version than the one that comes via the Ubuntu repository. Note that the Docker package is named `lxc-docker` when it is installed this way. The command used to run Docker commands is still `docker`.

Upgrading Docker on Ubuntu Trusty 14.04 LTS

To check and download upgrades, all you have to do is to execute this command in a terminal:

```
sudo apt-get update && sudo apt-get upgrade
```

User permissions

For convenience, it's preferred to add our user to the system's Docker user group so that we can control Docker without using `sudo`. This gives our user permission to execute Docker commands.

Replace `USER` with your username before you run the code:

```
sudo gpasswd -a USER docker
```

You might have to log out and log in again for it to work. When you are logged back in, run `docker ps` to verify that there are no permission problems.



More detailed information can be found in the official installation guide at <https://docs.docker.com/installation/ubuntu/>.



Docker on Mac OS X

To be able to use Docker on Mac OS X, we have to run the Docker service inside a virtual machine (VM) since Docker uses Linux-specific features to run. We don't have to get frightened by this since the installation process is very short and straightforward.

Installation

There is an OS X installer that installs everything we need, that is, VirtualBox, boot2docker, and Docker.

VirtualBox is a virtualizer in which we can run the lightweight Linux distribution, and boot2docker is a virtual machine that runs completely in the RAM and occupies just about 27 MB of space.

[ The latest released version of the OS X installer can be found at <https://github.com/boot2docker/osx-installer/releases/latest>.]

Now, let's take a look at how the installation should be done with the following steps:

1. Download the installer by clicking on the button named **Boot2Docker-1.3.0.pkg** to get the .pkg file, as shown in the following screenshot:

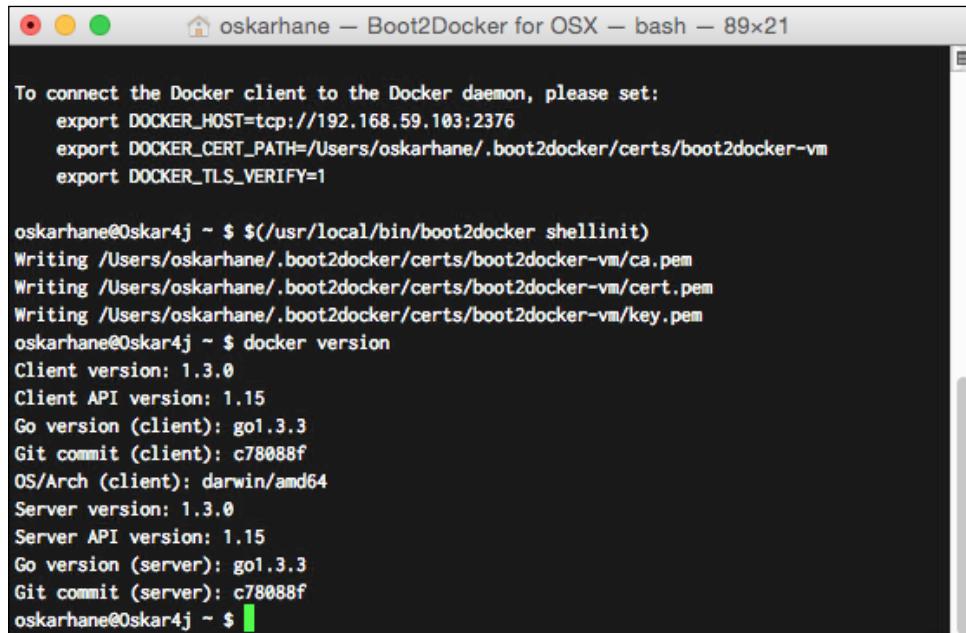


2. Double-click on the downloaded .pkg file and go through with the installation process.
3. Open the **Finder** window and navigate to your Applications folder; locate **boot2docker** and double-click on it. A terminal window will open and issue a few commands, as shown here:



This runs a Linux VM, named `boot2docker-vm`, that has Docker pre-installed in VirtualBox. The Docker service in the VM runs in daemon (background) mode, and a Docker client is installed in OS X, which communicates directly with the Docker daemon inside the VM via the Docker Remote API.

4. You will see a screen similar to the following screenshot, which tells you to set some environment variables:



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

oskarhane@Oskar4j ~ $ $(/usr/local/bin/boot2docker shellinit)
Writing /Users/oskarhane/.boot2docker/certs/boot2docker-vm/ca.pem
Writing /Users/oskarhane/.boot2docker/certs/boot2docker-vm/cert.pem
Writing /Users/oskarhane/.boot2docker/certs/boot2docker-vm/key.pem
oskarhane@Oskar4j ~ $ docker version
Client version: 1.3.0
Client API version: 1.15
Go version (client): go1.3.3
Git commit (client): c78088f
OS/Arch (client): darwin/amd64
Server version: 1.3.0
Server API version: 1.15
Go version (server): go1.3.3
Git commit (server): c78088f
oskarhane@Oskar4j ~ $
```

We open up the `~/.bash_profile` file and paste three lines from our output, as follows, at the end of this file:

```
export DOCKER_HOST=tcp://192.168.59.103:2376
export DOCKER_CERT_PATH=/Users/xxx/.boot2docker/certs/
boot2docker-vm
export DOCKER_TLS_VERIFY=1
```

The reason why we do this is so that our Docker client knows where to find the Docker daemon. If you want to find the IP in the future, you can do so by executing the `boot2docker ip` command. Adding the preceding lines will set these variables every time a new terminal session starts. When you're done, close the terminal window. Then, open a new window and type `echo $DOCKER_HOST` to verify that the environment variable is set as it should be. You should see the IP and port your boot2docker VM printed.

5. Type `docker version` to verify that you can use the Docker command.
An output that looks similar to the last few lines of the preceding screenshot will mean that we have succeeded.

Upgrading Docker on Mac OS X

Since Docker is relatively new, there could be a lot happening in every update, so make sure that you check for updates on a regular basis. From time to time, go to the Mac OS X installer download page and check whether there is an upgrade available. If there is, execute these commands to update it:

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

Docker on Windows

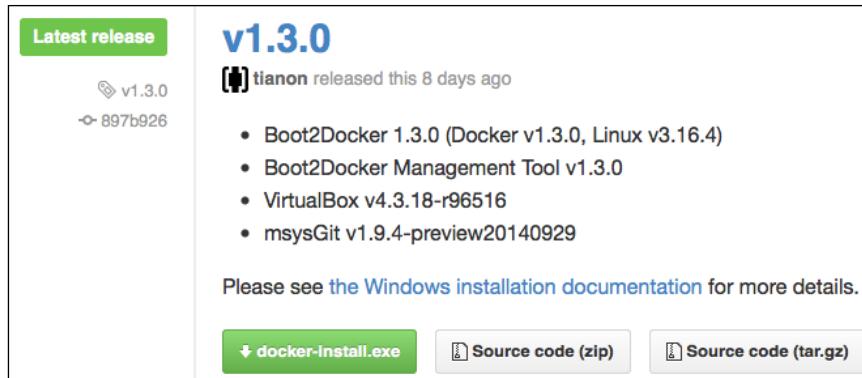
Just as we have to install a Linux virtual machine when installing Docker in OS X, we have to do the same in Windows in order to run Docker because of the Linux kernel features that Docker builds on. OS X has a native Docker client that directly communicates with the Docker daemon inside the virtual machine, but there isn't one available for Windows yet. There is a native Windows version of the Docker client coming, but it will not be available by the time this book is published.

Installation

There is a Windows installer that installs everything we need in order to run Docker. For this, go to <https://github.com/boot2docker/windows-installer/releases/latest>.

Now, let's take a look at how the installation should be done with the help of the following steps:

1. Click on the **docker-install.exe** button to download the .exe file, as shown in the following screenshot:



2. When the download is complete, run the downloaded installer. Follow through the installation process, and you will get VirtualBox, msysGit, and boot2docker installed.
3. Go to your `Program Files` folder and click on the newly installed boot2docker to start using Docker. If you are prompted to enter a passphrase, just press *Enter*.
4. Type `docker version` to verify that you can use the Docker command.

Upgrading Docker on Windows

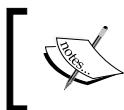
A new software changes often and to keep boot2docker updated, invoke these commands:

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

Docker on Amazon EC2

Throughout this book, I will use an Amazon EC2 instance, and since it is a superb place to host your PaaS, I will recommend that you do the same.

EC2 stands for Elastic Compute Cloud, and it is an infrastructure type of service. Amazon EC2 offers virtual servers that are created and available within a minute of ordering them.



Amazon has instances named t [x] .micro that you can use for free for 750 hours per month. You can read more about them at <http://aws.amazon.com/free>.

Amazon has its own Linux named Amazon Linux AMI that can be used to run Docker.

Installation

Let's see how the installation is done with the following steps:

1. Create an account at <http://aws.amazon.com> and go to Amazon's **Create EC2 Instance Wizard** at <https://console.aws.amazon.com/ec2/v2/home?#LaunchInstanceWizard>.

The next steps are shown in the screenshot as follows:

Step 1: Choose an Amazon Machine Image (AMI)

An AMI is a template that contains the software configuration (operating system, application server, and applications) required to launch your instance. You can select an AMI provided by AWS, our user community, or the AWS Marketplace; or you can select one of your own AMIs.

Name	Description	Type
amzn-ami-hvm-2014.09.2.x86_64-ebs - ami-dfc39aef	Amazon Linux AMI 2014.09.2 x86_64 HVM EBS	64-bit
RHEL-7.1_HVM_GA-20150225-x86_64-1-Hourly2-GP2 - ami-4dbf9e7d	Provided by Red Hat, Inc.	64-bit
suse-sles-12-v20141023-hvm-ssd-x86_64 - ami-d7450be7	SUSE Linux Enterprise Server 12 (HVM, 64-bit, SSD-Backed)	64-bit
ubuntu/images/hvm-ssd/ubuntu-trusty-14.04-amd64-server-20150123 - ami-29ebb519	Root device type: ebs Virtualization type: hvm	64-bit

2. Click on **Community AMIs** in the menu on the left-hand side and select the latest amzn-ami-pv. Make sure that you select the pv version and not the hvm version so that you have a virtualization that is more stable and has less overhead, as shown here:

amzn-ami-pv-2014.09.1.x86_64-ebs - ami-55a7ea65

Amazon Linux AMI 2014.09.1 x86_64 PV EBS

Root device type: ebs Virtualization type: paravirtual

Select

64-bit

3. When it's time to choose an instance type, you can choose **t1.micro** or **t2.micro** for now if they are available. The micro instances are very limited in their performance, but since they are available in the free usage tier in some regions and this is not for a live site at the moment, we can use them. Click on **Next: Configure Instance Details** and then click on the **Review and Launch** button, as shown in the following screenshot:

Step 2: Choose an Instance Type

Amazon EC2 provides a wide selection of instance types optimized to fit different use cases. Instances are virtual servers that can run applications. They have varying combinations of CPU, memory, storage, and networking capacity, and give you the flexibility to choose the appropriate mix of resources for your applications. [Learn more](#) about instance types and how they can meet your computing needs.

Filter by:	All instance types	Current generation	Show/Hide Columns			
Currently selected: t2.micro (Variable ECUs, 1 vCPUs, 2.5 GHz, Intel Xeon Family, 1 GiB memory, EBS only)						
	Family	Type	vCPUs	Memory (GiB)	Instance Storage (GB)	EBS-Optimized Available
<input checked="" type="checkbox"/>	General purpose	t2.micro <small>Free tier eligible</small>	1	1	EBS only	-
<input type="checkbox"/>	General purpose	t2.small	1	2	EBS only	-
<input type="checkbox"/>	General purpose	t2.medium	2	4	EBS only	-
<input type="checkbox"/>	General purpose	m3.medium	1	3.75	1 x 4 (SSD)	-
<input type="checkbox"/>	General purpose	m3.large	2	7.5	1 x 32 (SSD)	-

4. Verify all the details on the **summary** page and click on the **Launch Instance** button.
5. You will be prompted whether you want to use an existing key-pair or create a new one. If this is your first time creating an Amazon EC2 instance, you will want to create a new key-pair. This makes it easy to securely connect to your instances.
6. Download the new key-pair, move it to your `~/.ssh/` folder, and remove the `.txt` extension.
7. It's also important to set the correct user permissions on the file or SSH will refuse to use it.

In Linux or on a Mac, this is how the terminal command to do this looks:

```
mv ~/Downloads/amz.pem.txt ~/.ssh/amz.pem
chmod 600 ~/.ssh/amz.pem
```

On Windows, save the key anywhere and use a tool such as PuTTYgen to convert it to a `.ppk` file, so you can use it when connecting using PuTTY.

8. You will be prompted to choose a security group for your instance. Pick the default one since this won't be a production server. When it's time to use a production server, we might want to add more security to our instance.
9. Now we're up and running! Let's connect to it. Click on the **View Instances** button and select your newly created instance in the list, as shown here:

The screenshot shows the 'Launch Status' page for an AWS instance. At the top, a green box says 'Your instance is now launching' with a checkmark icon. Below it, a link to 'View launch log'. A blue box below says 'Get notified of estimated charges' with a speech bubble icon, and a link to 'Create billing alerts'. Under 'How to connect to your instance', it says the instance is launching and may take a few minutes to reach a 'running' state. It also says to click 'View Instances' to monitor status. A section titled 'Here are some helpful resources to get you started' lists links to 'How to connect to your Linux instance', 'Learn about AWS Free Usage Tier', 'Amazon EC2: User Guide', and 'Amazon EC2: Discussion Forum'. At the bottom, it says instances are launching and lists links to 'Create status check alarms', 'Create and attach additional EBS volumes', and 'Manage security groups'. A blue 'View Instances' button is at the bottom right.

10. In the bottom of the screen, you can see some information about the instance. You should be looking for the public DNS information. This is how it should look:

`ec2-54-187-234-27.us-west-2.compute.amazonaws.com`

11. On a Linux or Mac, open a terminal and connect to it:

```
ssh ec2-user@ec2-54-187-234-27.us-west-2.compute.amazonaws.com -i  
~/.ssh/amz.pem
```

The screenshot is displayed as follows:

```
oskarhane@oskar4j ~ $ ssh ec2-user@ec2-54-187-234-27.us-west-2.compute.amazonaws.com -i ~/.ssh/amz.pem
The authenticity of host 'ec2-54-187-234-27.us-west-2.compute.amazonaws.com (54.187.234.27)' can't be established.
RSA key fingerprint is 61:59:ce:8a:45:51:67:59:d5:bc:ec:e0:81:d1:69:7a.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'ec2-54-187-234-27.us-west-2.compute.amazonaws.com,54.187.234.27' (RSA) to the list of known hosts.
=====
@           WARNING: UNPROTECTED PRIVATE KEY FILE!           @
=====
Permissions 0644 for '/Users/oskarhane/.ssh/amz.pem' are too open.
It is required that your private key files are NOT accessible by others.
This private key will be ignored.
bad permissions: ignore key: /Users/oskarhane/.ssh/amz.pem
Permission denied (publickey).
oskarhane@oskar4j ~ $ chmod 600 ~/.ssh/amz.pem
oskarhane@oskar4j ~ $ ssh ec2-user@ec2-54-187-234-27.us-west-2.compute.amazonaws.com -i ~/.ssh/amz.pem

 _I_ _|-_
 -| (   /   Amazon Linux AMI
 ___\_\_|__|_|
 
https://aws.amazon.com/amazon-linux-ami/2014.09-release-notes/
No packages needed for security; 4 packages available
Run "sudo yum update" to apply all updates.
-bash: warning: setlocale: LC_CTYPE: cannot change locale (UTF-8): No such file or directory
[ec2-user@ip-172-31-32-58 ~]$
```

We use the `ec2-user` user that is the default user for Amazon's Linux instances, and `amz.pem` is the key we downloaded earlier. Replace the URL with your public DNS information from the last step.

When asked whether you want to continue because of an unknown host, type yes.

On Windows, use PuTTY and make sure that you have specified the converted private key from step 4 in the PuTTY Auth tab.

12. Once you are connected to the instance, install Docker:

```
sudo yum update  
sudo yum install -y docker  
sudo service docker start
```

- To test whether it's working as expected, type `docker version` and make sure there's no error. You should see a few lines with the client version, API version, and so on.

Open ports

Amazon's default security policy is to block the default ports used to expose services from Docker, so we have to change this.

- We go back to the EC2 dashboard and click on the **Security Groups** option in the menu
- Select the security group that your EC2 instance uses and select the **Inbound** tab
- Docker uses ports in a range from **49000 - 50000**, so we add a rule for this, as shown in the following screenshot:

The screenshot shows the AWS Management Console interface for managing security groups. The top navigation bar has tabs for 'Description', 'Inbound' (which is highlighted in orange), 'Outbound', and 'Tags'. Below the tabs is an 'Edit' button. The main area displays a table with four columns: 'Type', 'Protocol', 'Port Range', and 'Source'. There are three existing rules: one for SSH (TCP port 22) from 0.0.0.0/0, one for HTTP (TCP port 80) from 0.0.0.0/0, and a third row for a 'Custom TCP Rule' (TCP port range 49000 - 50000) from 0.0.0.0/0. A new row is being added at the bottom of the table.

Type	Protocol	Port Range	Source
SSH	TCP	22	0.0.0.0/0
HTTP	TCP	80	0.0.0.0/0
Custom TCP Rule	TCP	49000 - 50000	0.0.0.0/0

Upgrading Docker on Amazon EC2

Upgrading an Amazon Linux AMI instance is as easy as it is for Ubuntu. Type `sudo yum update` and confirm whether there's an update waiting. This command will list all the available updates and upon your confirmation, install them.

User permissions

Docker requires commands to be run by users in the `docker` user group. For convenience, we add our user to the Docker group so that we can control Docker without using `sudo`:

```
sudo gpasswd -a ec2-user docker
```

You might have to log out and log in again for it to work. When you are logged back in, run `docker ps` to verify that there are no permission problems. You should see a row of capitalized words, such as **CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES**.

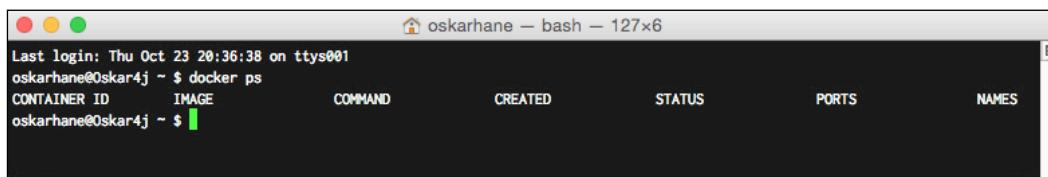
Displaying Hello World

Now that we have Docker running on a machine of our choice, it's time to make Docker work for us. Here are a few very basic commands that we can use for some basic interaction with the Docker daemon.

In the next chapter, all the concepts and phrases used in Docker will be explained:

- `docker ps`: This lists the running containers
- `docker ps -a`: This lists all the containers, both running and exited
- `docker images`: This lists local (downloaded and locally created) images
- `docker run`: This will launch a new instance container from an image
- `docker stop`: This is used to stop a container

Let's try the first one in the screenshot shown below:

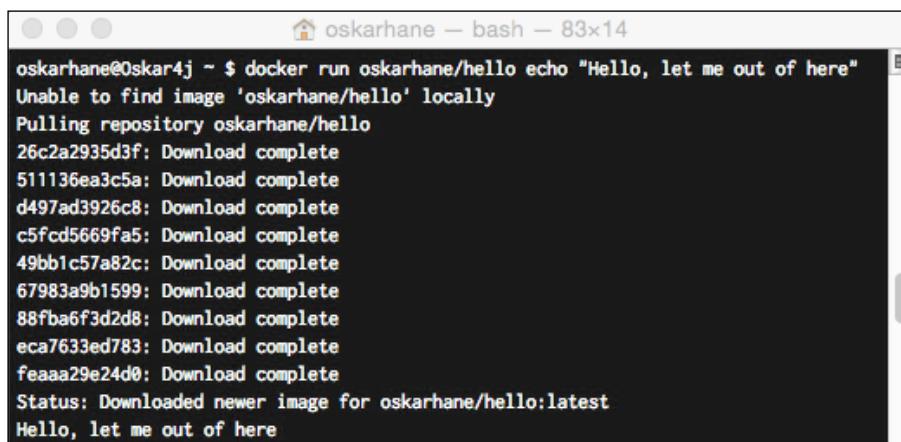


```
Last login: Thu Oct 23 20:36:38 on ttys001
oskarhane@Oskar4j ~ $ docker ps
CONTAINER ID        IMAGE               COMMAND       CREATED          STATUS          PORTS          NAMES
oskarhane@Oskar4j ~ $
```

As expected, we have nothing running yet.

Launching a container is as easy as `docker run [image] [command]`. If the image doesn't exist locally, Docker will download it from the Docker Registry Hub and launch your container when it's downloaded.

The following steps are displayed as follows:



```
oskarhane@Oskar4j ~ $ docker run oskarhane/hello echo "Hello, let me out of here"
Unable to find image 'oskarhane/hello' locally
Pulling repository oskarhane/hello
26c2a2935d3f: Download complete
511136ea3c5a: Download complete
d497ad3926c8: Download complete
c5fcfd5669fa5: Download complete
49bb1c57a82c: Download complete
67983a9b1599: Download complete
88fba6f3d2d8: Download complete
eca7633ed783: Download complete
feaaa29e24d0: Download complete
Status: Downloaded newer image for oskarhane/hello:latest
Hello, let me out of here
```

Type the following command in a terminal to launch a container that prints the string **Hello, let me out of here** and then exits:

```
docker run oskarhane/hello echo "Hello, let me out of here"
```

This is not very useful, but we just ran a command in Ubuntu inside the container.

If we type `docker ps` again, we can see that we still have no running containers since we exited the one we just started straightaway. Try using `docker ps -a` instead, and try `docker images`.

Summary

In this chapter, we learned that Docker can be used on most operating systems and that the installation process varies a lot depending on the OS. We had our first interaction with the Docker daemon and launched our first container in Docker. Even though all the container did was write a command, that's how easy it is to start and run something inside a guest operating system.

We have also introduced the theme that shows what this book is all about, running a multicontainer web app of a web server container and a MySQL container: your own PaaS.

In the next chapter, we will further explore Docker, its terminology, and the community around it.

2

Exploring Docker

After reading this chapter, you will find yourself more comfortable talking about and using Docker. The following topics will be covered here:

- The Docker image
- The Docker container
- The Docker command-line interface
- The Docker Registry Hub

You will find these topics important when building your PaaS, and you will use and interact with all of them throughout this book.

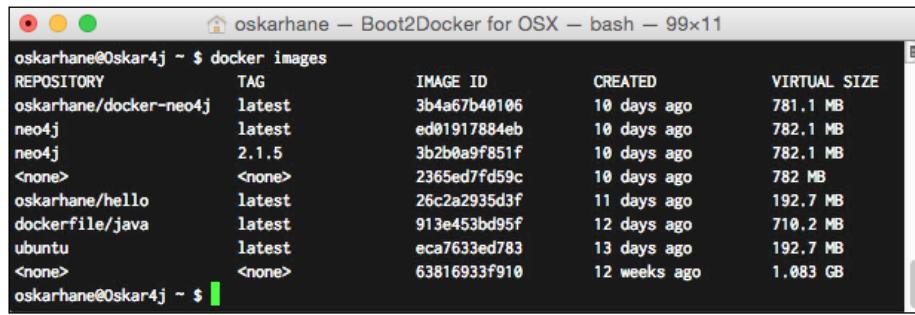
The Docker image

In the beginning, it can be hard to understand the difference between a Docker image and a Docker (or Linux) container.

Imagine that our Linux kernel is layer zero. Whenever we run a Docker image, a layer is put on top of our kernel layer. This image, layer one, is a read-only image and cannot be changed or cannot hold a state.

A Docker image can build on top of another Docker image that builds on top of another Docker image and so on. The first image layer is called a **base image**, and all other layers except the last image layer are called **parent images**. They inherit all the properties and settings of their parent images and add their own configuration in the Dockerfile.

Docker images are identified by an image ID, which is a 64-character long hexadecimal string, but when working with images, we will almost never reference an image by this ID but use the image names instead. To list all our locally available Docker images, we use the `docker images` command. Take a look at the following image to see how the images are listed:



The screenshot shows a terminal window titled "oskarhane — Boot2Docker for OSX — bash — 99x11". The command `oskarhane@oskar4j ~ $ docker images` is run, displaying a table of images. The columns are REPOSITORY, TAG, IMAGE ID, CREATED, and VIRTUAL SIZE.

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
oskarhane/docker-neo4j	latest	3b4a67b40106	10 days ago	781.1 MB
neo4j	latest	ed01917884eb	10 days ago	782.1 MB
neo4j	2.1.5	3b2b0a9f851f	10 days ago	782.1 MB
<none>	<none>	2365ed7fd59c	10 days ago	782 MB
oskarhane/hello	latest	26c2a2935d3f	11 days ago	192.7 MB
dockerfile/java	latest	913e453bd95f	12 days ago	710.2 MB
ubuntu	latest	eca7633ed783	13 days ago	192.7 MB
<none>	<none>	63816933f910	12 weeks ago	1.083 GB

oskarhane@oskar4j ~ \$

Images can be distributed with different versions for us to choose from, and the mechanism for this is called **tags**. The preceding screenshot illustrates this with the **neo4j** image that has a **latest** and a **2.1.5** tag. This is how the command used to pull a specific tag looks:

```
docker pull ubuntu:14.04
docker pull ubuntu:12.02
```

The Docker container

A Docker container is created the moment we execute `docker run imagename`. A writeable layer is added on top of all the image layers. This layer has processes running on the CPU and can have two different states: running or exited. This is the container. When we start a container with the Docker run command, it enters the running state until it, for some reason, stops by itself or is stopped by us and then enters the exited state.

When we have a container running, all the changes we make to its filesystem are permanent between start and stop. Remember that changes made to the container's filesystem are not written to the underlying Docker image.

We can start as many instances of running containers as we want from the same image; they will all live side by side, totally separated by each other. All the changes we make to a container are limited to that container only.

If changes are made to the container's underlying image, the running container is unaffected and there is no autoupdate happening. If we want to update our container to a newer version of its image, we have to be careful and make sure that we have set up the data structure in a correct way, otherwise we have the risk of losing all the data in the container. Later in this book, I will show you where to keep important data without the risk of losing it.

The corresponding screenshot is shown as follows:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
6a2356a84a67	neo4j:latest	/bin/bash -c /launc	10 days ago	Exited (-1) 10 days ago		thirsty_yonath
b63b99a49ef	63816933f910	/bin/bash -c /launc	10 days ago	Exited (-1) 10 days ago		sleepy_hawking
36c9eb86d307	neo4j:2.1.5	/bin/bash -c /launc	10 days ago	Exited (0) 10 days ago		tender_hawking
04702a60d226	2365ed7fd59c	/bin/bash -c /launc	10 days ago	Exited (-1) 10 days ago		hopeful_leakey
1bcc0926d0e2	2365ed7fd59c	/bin/bash -c /launc	10 days ago	Exited (-1) 10 days ago		nostalgic_curie
eddf388cdadd	oskarhane/hello:latest	echo 'Hello, let me	11 days ago	Exited (0) 11 days ago		desperate_franklin

A 64-character long hexadecimal string called **container ID** identifies Docker containers. This ID can be used when interacting with the container, and depending on how many containers we have running, we will usually only have to type the first four characters of the container ID. We can use the container name as well, but it's often easier to type the beginning of the ID.

The Docker command-line interface

The command line interface is where we communicate with the daemon using the Docker command. The Docker daemon is the background process that receives the commands that are typed by us.

In the previous chapter, we ran a few Docker commands to start and stop containers as well as to list containers and images. Now, we are going to learn a few more that will help us when handling containers for our PaaS, as follows:

- `docker logs <container-ID|name>`: Everything that is written to the STDOUT containers will end up in the file that can be accessed via this command. This is a very handy way to output information from within a container, as shown here:

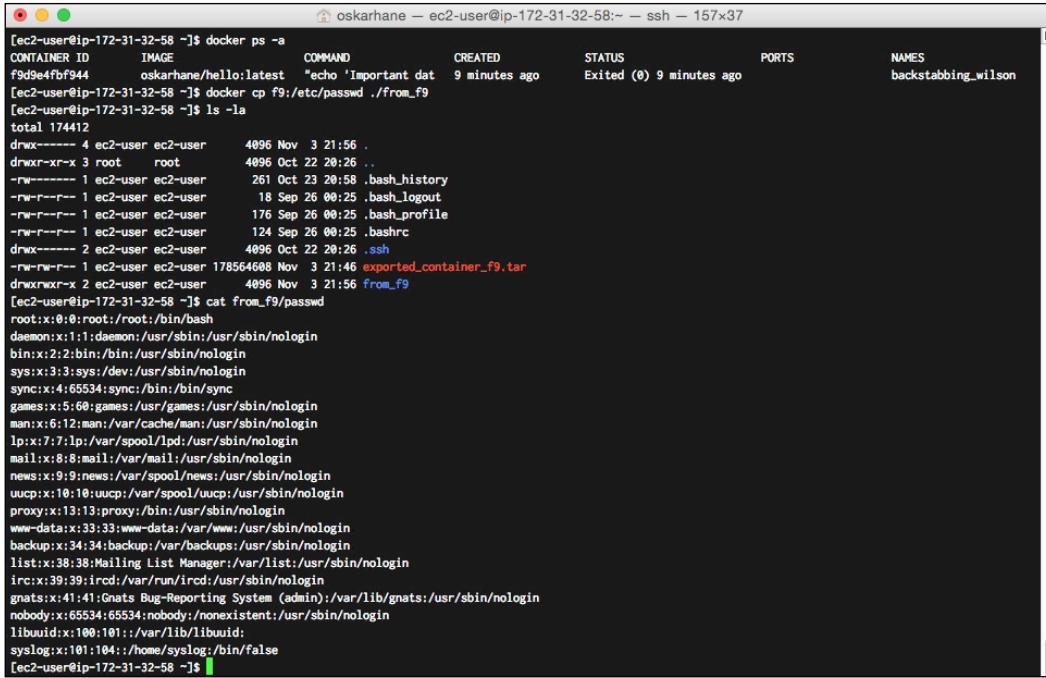
```
oskarhane@Oskar4j ~ $ docker run oskarhane/hello echo "Let's go for this container again. This is written to the stdout from inside the container."
Let's go for this container again. This is written to the stdout from inside the container.
oskarhane@Oskar4j ~ $ docker logs 9ad7
Let's go for this container again. This is written to the stdout from inside the container.
oskarhane@Oskar4j ~ $
```

- `docker export <container-ID|name>`: If you have a container that holds data that you want to export, this is the command to be used. This creates a tar archive and sends it to STDOUT:

```
[ec2-user@ip-172-31-32-58 ~]$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
f9d9e4fb944        oskarhane/hello:latest   "echo 'Important dat"  6 seconds ago    Exited (0) 6 seconds ago          backstabbing_wilson

[ec2-user@ip-172-31-32-58 ~]$ docker export f9 > exported_container_f9.tar
[ec2-user@ip-172-31-32-58 ~]$ ls -la
total 174408
drwx----- 3 ec2-user ec2-user      4096 Nov  3 21:46 .
drwxr-xr-x  3 root     root       4096 Oct 22 20:26 ..
-rw-----  1 ec2-user ec2-user      261 Oct 23 20:58 .bash_history
-rw-r--r--  1 ec2-user ec2-user      18 Sep 26 00:25 .bash_logout
-rw-r--r--  1 ec2-user ec2-user      176 Sep 26 00:25 .bash_profile
-rw-r--r--  1 ec2-user ec2-user      124 Sep 26 00:25 .bashrc
drwx----- 2 ec2-user ec2-user      4096 Oct 22 20:26 .ssh
-rw-rw-r--  1 ec2-user ec2-user 178564608 Nov  3 21:46 exported_container_f9.tar
[ec2-user@ip-172-31-32-58 ~]$
```

- `docker cp CONTAINER:PATH HOSTPATH`: If you don't want the whole file system from a container but just one directory or a file, you can use `docker cp` instead of `export`, as shown in the following screenshot:



```
[ec2-user@ip-172-31-32-58 ~]$ docker ps -a
CONTAINER ID        IMAGE               COMMAND       CREATED          STATUS           PORTS     NAMES
f9d9e4fb944        oskarhane/hello:latest "echo 'Important dat" 9 minutes ago   Exited (0) 9 minutes ago
[ec2-user@ip-172-31-32-58 ~]$ docker cp f9:/etc/passwd ./from_f9
[ec2-user@ip-172-31-32-58 ~]$ ls -la
total 174412
drwx----- 4 ec2-user ec2-user    4096 Nov  3 21:56 .
drwxr-x--x  3 root   root      4096 Oct 22 20:26 ..
-rw-----  1 ec2-user ec2-user    261 Oct 23 20:58 .bash_history
-rw-r--r--  1 ec2-user ec2-user    18 Sep 26 00:25 .bash_logout
-rw-r--r--  1 ec2-user ec2-user   176 Sep 26 00:25 .bash_profile
-rw-r--r--  1 ec2-user ec2-user   124 Sep 26 00:25 .bashrc
drwx----- 2 ec2-user ec2-user    4096 Oct 22 20:26 .ssh
-rw-rw-r--  1 ec2-user ec2-user 178564608 Nov  3 21:46 exported_container_f9.tar
drwxr-x--x  2 ec2-user ec2-user    4096 Nov  3 21:56 from_f9
[ec2-user@ip-172-31-32-58 ~]$ cat from_f9/passwd
root:x:0:root:root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin/nologin
bin:x:2:2:bin:/bin/nologin
sys:x:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
libuuid:x:100:101:/var/lib/libuuid:
syslog:x:101:104::/home/syslog/bin/false
[ec2-user@ip-172-31-32-58 ~]$
```

The Docker Registry Hub

One important part of Docker's popularity is its community and the ease with which you can share, find, and extend Docker images. The central place for this is the Docker Registry Hub that can be found at <https://hub.docker.com/>.

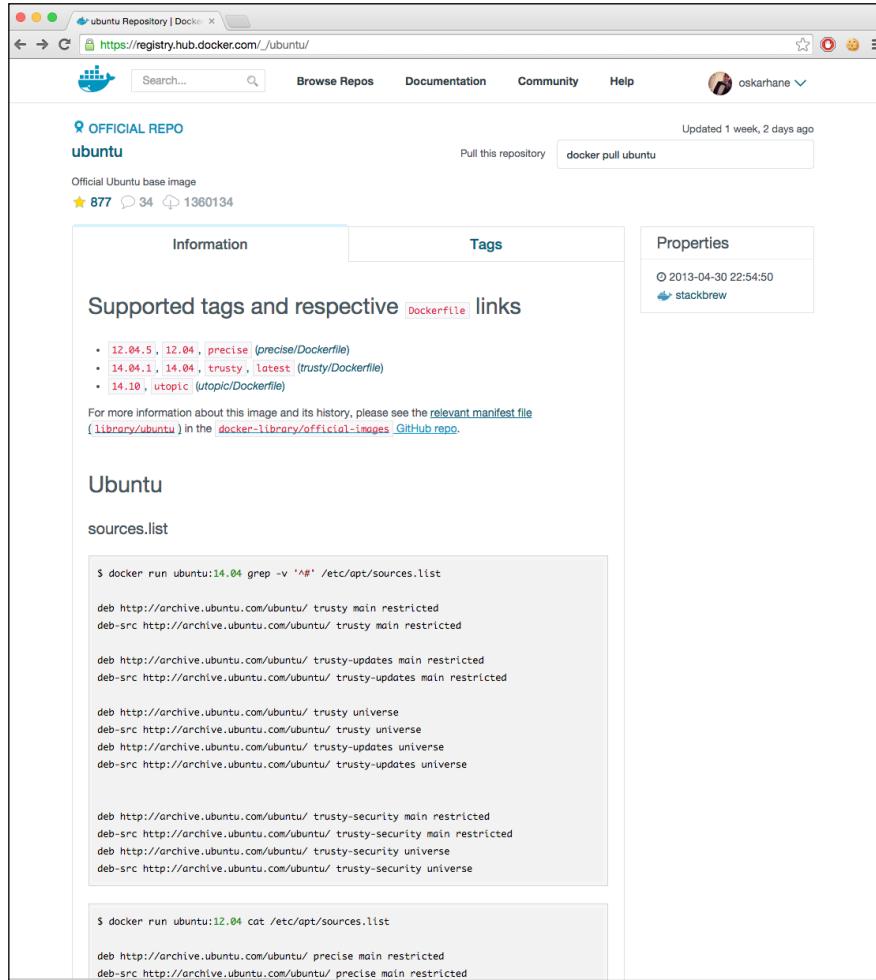
Browsing repositories

Here, we can search and, in many ways, browse for image repositories to find exactly what we're after. If we take a look at the popular ones, we will see what others are using the most.

If we click on the Ubuntu repository, we will see lots of information about the image, the tags that are available, users' comments, the number of stars it has, and when it was updated.

Exploring Docker

The screenshot is displayed as follows:



If we click on a tag in the main view, we'll see something called the **Dockerfile**. This is the image description that runs when an image is being created. Further in this book, we'll write our own.

If you're interested in an image in the Docker hub, I recommend that you read the Information/README as well as the other users' comments. Often, you will find valuable information there that will help you to choose the right image and show you how to run it in the way the maintaining developer intended to.

Often, you will find images that almost fit your needs since most images are quite general, but as a developer, you might need specific settings or services installed.

Exploring published images

Take the official WordPress Docker image, for example (https://registry.hub.docker.com/_/wordpress/). You'll find it on the Docker hub's browse page or you can search for it.

Let's forget about these shortcomings for now and see what the information page says:



The screenshot shows the Docker Hub page for the official WordPress Docker image. At the top is the image logo, followed by the heading "How to use this image". Below this is a code block containing the command `docker run --name some-wordpress --link some-mysql:mysql -d wordpress`. A note below the command states: "The following environment variables are also honored for configuring your WordPress instance:" followed by a bulleted list of variables. Another note explains that if `WORDPRESS_DB_NAME` does not exist, it will be created automatically. A final note suggests using port mapping with `-p 8080:80`.

```
docker run --name some-wordpress --link some-mysql:mysql -d wordpress
```

The following environment variables are also honored for configuring your WordPress instance:

- `-e WORDPRESS_DB_USER=...` (defaults to "root")
- `-e WORDPRESS_DB_PASSWORD=...` (defaults to the value of the `MYSQL_ROOT_PASSWORD` environment variable from the linked mysql container)
- `-e WORDPRESS_DB_NAME=...` (defaults to "wordpress")
- `-e WORDPRESS_AUTH_KEY=..., -e WORDPRESS_SECURE_AUTH_KEY=..., -e WORDPRESS_LOGGED_IN_KEY=..., -e WORDPRESS_NONCE_KEY=..., -e WORDPRESS_AUTH_SALT=..., -e WORDPRESS_SECURE_AUTH_SALT=..., -e WORDPRESS_LOGGED_IN_SALT=..., -e WORDPRESS_NONCE_SALT=...` (default to unique random SHA1s)

If the `WORDPRESS_DB_NAME` specified does not already exist in the given MySQL container, it will be created automatically upon container startup, provided that the `WORDPRESS_DB_USER` specified has the necessary permissions to create it.

If you'd like to be able to access the instance from the host without the container's IP, standard port mappings can be used:

```
docker run --name some-wordpress --link some-mysql:mysql -p 8080:80 -d wordpress
```

Then, access it via <http://localhost:8080> or <http://host-ip:8080> in a browser.

This image reads the settings from the Docker container's environment variables. This means that image has to be started with the environment variables injected using the `docker run -e` command, or you can `--link` another container to it that injects these variables. We'll discuss container linking more later in this book.

Let's see what we'll get if we were to pull this image. Click on the link to the Dockerfile in the apache directory:

```
FROM php:5.6-apache

RUN a2enmod rewrite

# install the PHP extensions we need
RUN apt-get update && apt-get install -y libpng12-dev libjpeg-dev && rm
-rf /var/lib/apt/lists/* \
&& docker-php-ext-configure gd --with-png-dir=/usr --with-jpeg-dir=/
usr \
&& docker-php-ext-install gd
RUN docker-php-ext-install mysqli

VOLUME /var/www/html

ENV WORDPRESS_VERSION 4.1.1
ENV WORDPRESS_UPSTREAM_VERSION 4.1.1
ENV WORDPRESS_SHA1 15d38fe6c73121a20e63cccd8070153b89b2de6a9

# upstream tarballs include ./wordpress/ so this gives us /usr/src/
wordpress
RUN curl -o wordpress.tar.gz -SL https://wordpress.org/wordpress-
${WORDPRESS_UPSTREAM_VERSION}.tar.gz \
&& echo "$WORDPRESS_SHA1 *wordpress.tar.gz" | shasum -c - \
&& tar -xzf wordpress.tar.gz -C /usr/src/ \
&& rm wordpress.tar.gz

COPY docker-entrypoint.sh /entrypoint.sh

# grr, ENTRYPOINT resets CMD now
ENTRYPOINT ["/entrypoint.sh"]
CMD ["apache2-foreground"]
```

Ok, we see that it builds on Debian Wheezy and installs Apache2, PHP5, and some other stuff. After that, it sets a bunch of environment variables and then downloads WordPress.

We see a few lines starting with the command `COPY`. This means that files are shipped with the Docker image and are copied to the inside of the container when it's started. This is how the `docker-apache.conf` file shipped with the WordPress image looks:

```
<VirtualHost *:80>
    DocumentRoot /var/www/html
    <Directory /var/www/html>
        AllowOverride all
    </Directory>
</VirtualHost>
# vim: syntax=apache ts=4 sw=4 sts=4 sr noet
```

The preceding line of code tells Apache where to look for files.

What about the `docker-entrypoint.sh` file?

The `ENTRYPOINT` keyword tells the Docker daemon that if nothing else is specified, this file should be executed whenever the container is run. It is as if the whole container is an executable file.

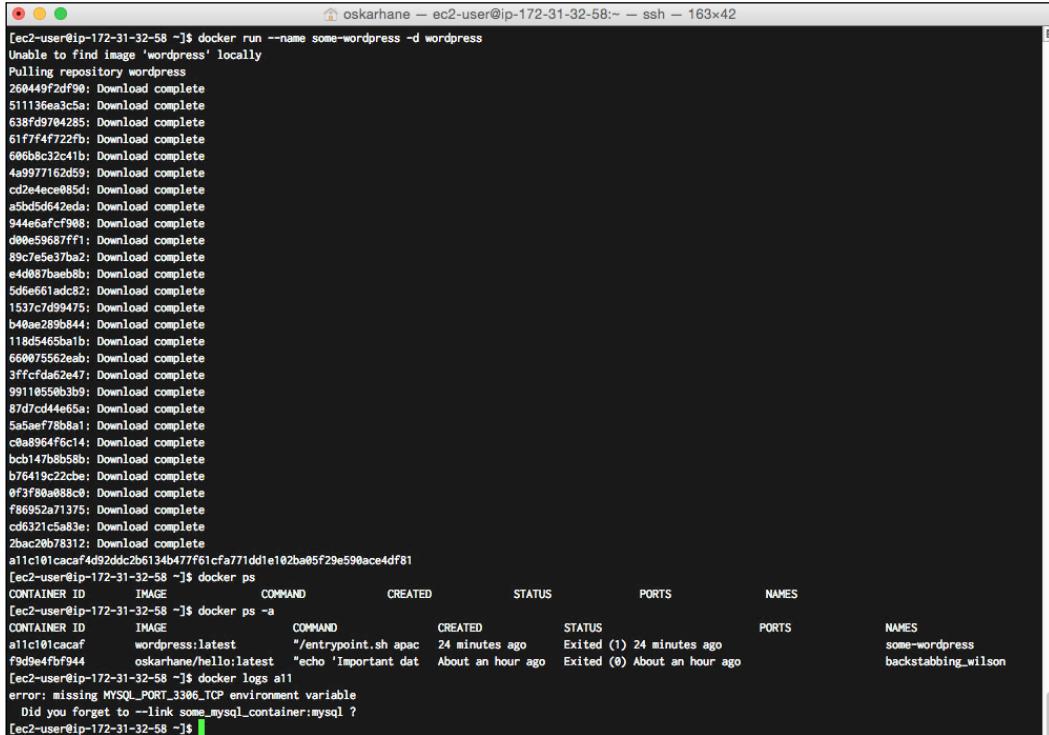
If we take a look at what is present inside this file, we'll see that it basically sets up the connection to the MySQL database and configures `.htaccess` and WordPress:

```
#!/bin/bash
set -e
if [ -z "$MYSQL_PORT_3306_TCP" ]; then
    echo >&2 'error: missing MYSQL_PORT_3306_TCP environment variable'
    echo >&2 ' Did you forget to --link some_mysql_container:mysql ?'
    exit 1
fi
```

The first thing that is done is to check whether the user has set environment variables for the MySQL connection. If not, it exits and writes some info to STDERR.

Why don't you try and see whether you can trigger the MySQL error that writes **error: missing MYSQL_PORT_3306_TCP environment variable** to the STDERR, as follows:

```
docker run --name some-wordpress -d wordpress
```



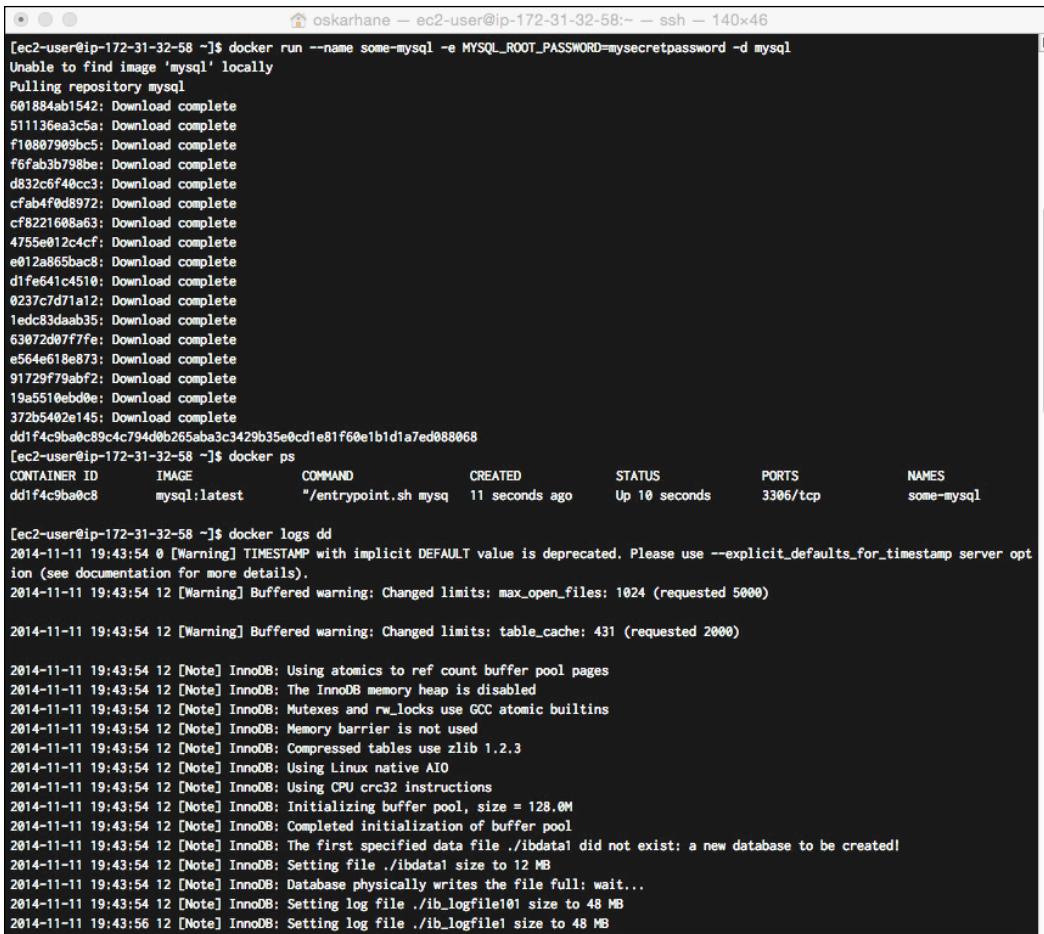
The screenshot shows a terminal window with the following session:

```
[ec2-user@ip-172-31-32-58 ~]$ docker run --name some-wordpress -d wordpress
Unable to find image 'wordpress' locally
Pulling repository wordpress
260449f2df90: Download complete
511136ea3c5e: Download complete
638fd9784285: Download complete
61f774f722fb: Download complete
6065bc32c41b: Download complete
4a9977162d59: Download complete
cdze4ec0e85d: Download complete
a5b5d6d42eda: Download complete
944e6acf9088: Download complete
d00e59687ff1: Download complete
89c7e5e37ba2: Download complete
e4d887baeb88: Download complete
5d6e61adac82: Download complete
1537c7d99475: Download complete
b408ae289b844: Download complete
118d5465ba1b: Download complete
660075562eab: Download complete
3ffcfcd462e47: Download complete
99118550b3b9: Download complete
87d7cd44e65a: Download complete
5a5aef78b8a1: Download complete
c3a8964f6c14: Download complete
bcb14768b58b: Download complete
b76419c22cbe: Download complete
ef5f80a088c80: Download complete
f86952a71375: Download complete
cd6321c5e63a: Download complete
2bac20b78312: Download complete
a11c101cacaf4d92ddc2b6134b477f61cfa771dd1e102ba05f29e59ace4df81
[ec2-user@ip-172-31-32-58 ~]$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
[ec2-user@ip-172-31-32-58 ~]$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
a11c101cacaf        wordpress:latest     "/entrypoint.sh apac"   24 minutes ago    Exited (1) 24 minutes ago          some-wordpress
f9d9e4fb944         oskarhane/hello:latest   "echo 'Important dat"   About an hour ago   Exited (0) About an hour ago          backstabbing_wilson
[ec2-user@ip-172-31-32-58 ~]$ docker logs a11
error: missing MYSQL_PORT_3306_TCP environment variable
Did you forget to --link some_mysql_container:mysql ?
[ec2-user@ip-172-31-32-58 ~]$
```

The `--name` `some-wordpress` command names the container, so we can reference it by this name later. Also, the `-d` argument tells the container to run in detached mode, which means that it does not listen to commands from where we started it anymore. The last `wordpress` argument is the name of the Docker image we want to run.

If we check the log for our new container, we'll see what the screenshot shows us: the expected error message.

Let's run a MySQL container and see whether we can get it to work. Navigate to https://registry.hub.docker.com/_/mysql/ in order to get to the official MySQL docker repository on the Docker registry hub. Here, it states that in order to start a MySQL instance, we need to invoke **docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=mysecretpassword -d mysql** in the shell. Since we are doing this for educational purposes at the moment, we don't have to choose a strong root user password. After some dependent images are downloaded, we should be able to see our running container when we execute `docker ps`. If we do, have a look at the installation log by running `docker logs some-mysql`, as shown here:



```

oskarhane ~ ec2-user@ip-172-31-32-58:~ ssh 140x46
[ec2-user@ip-172-31-32-58 ~]$ docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=mysecretpassword -d mysql
Unable to find image 'mysql' locally
Pulling repository mysql
601884ab1542: Download complete
511136ea3c5a: Download complete
f10807999bc5: Download complete
f6fbab5b798be: Download complete
d832c6f40cc3: Download complete
cfab4f0d8972: Download complete
cf8221608a63: Download complete
4755e012c4cf: Download complete
e012a865bac8: Download complete
d1fe641c4510: Download complete
0237c7d71a12: Download complete
1edc83daab35: Download complete
63072d0f77fe: Download complete
e564e618e873: Download complete
91729f79abf2: Download complete
19a510eb00: Download complete
372b5402e145: Download complete
dd1f4c9ba0c89c4c794d0b265aba3c3429b35e0cd1e81f60e1b1d1a7ed088068
[ec2-user@ip-172-31-32-58 ~]$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS          NAMES
dd1f4c9ba0c8        mysql:latest      "/entrypoint.sh mysq"   11 seconds ago    Up 10 seconds     3306/tcp       some-mysql

[ec2-user@ip-172-31-32-58 ~]$ docker logs dd
2014-11-11 19:43:54 [Warning] TIMESTAMP with implicit DEFAULT value is deprecated. Please use --explicit_defaults_for_timestamp server option (see documentation for more details).
2014-11-11 19:43:54 12 [Warning] Buffered warning: Changed limits: max_open_files: 1024 (requested 5000)

2014-11-11 19:43:54 12 [Warning] Buffered warning: Changed limits: table_cache: 431 (requested 2000)

2014-11-11 19:43:54 12 [Note] InnoDB: Using atomicity to ref count buffer pool pages
2014-11-11 19:43:54 12 [Note] InnoDB: The InnoDB memory heap is disabled
2014-11-11 19:43:54 12 [Note] InnoDB: Mutexes and rw_locks use GCC atomic builtins
2014-11-11 19:43:54 12 [Note] InnoDB: Memory barrier is not used
2014-11-11 19:43:54 12 [Note] InnoDB: Compressed tables use zlib 1.2.3
2014-11-11 19:43:54 12 [Note] InnoDB: Using Linux native AIO
2014-11-11 19:43:54 12 [Note] InnoDB: Using CPU crc32 instructions
2014-11-11 19:43:54 12 [Note] InnoDB: Initializing buffer pool, size = 128.0M
2014-11-11 19:43:54 12 [Note] InnoDB: Completed initialization of buffer pool
2014-11-11 19:43:54 12 [Note] InnoDB: The first specified data file ./ibdata1 did not exist: a new database to be created!
2014-11-11 19:43:54 12 [Note] InnoDB: Setting file ./ibdata1 size to 12 MB
2014-11-11 19:43:54 12 [Note] InnoDB: Database physically writes the file full: wait...
2014-11-11 19:43:54 12 [Note] InnoDB: Setting log file ./ib_logfile01 size to 48 MB
2014-11-11 19:43:56 12 [Note] InnoDB: Setting log file ./ib_logfile1 size to 48 MB

```

Great, now we have a running MySQL container that is needed to start a WordPress instance. Let's start a new WordPress instance with the MySQL link in place:

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

The `--link` parameter exposes the `some-mysql` containers' environment variables, interface, and exposed ports via the environment variables injected to the `some-wordpress` container.

To open a port that can be reached from the outside, port 80 is exposed via the `-p 80` parameter.

If you get an error message saying **Error response from daemon: Conflict, The name some-wordpress is already assigned to a11c101cacaf.**, you have to delete (or rename) that container to be able to assign `some-wordpress` to a container again. You need to give the new container a new name or delete the old (failing) WordPress container. Invoke `docker rm some-wordpress` to delete the old container using the desired name.

When you have the container running, invoke `docker ps` command to find out which of our ports was assigned to the container's private port 80.

We can either look at the ports column in the container list, or we can invoke `docker port some-wordpress 80` to explicitly find it, as shown here:



A screenshot of a terminal window titled "oskarhane — ec2-user@ip-172-31-32-58:~ — ssh — 161x9". The terminal shows the following command and its output:

```
[ec2-user@ip-172-31-32-58 ~]$ docker run --name some-wordpress --link some-mysql:mysql -p 80 -d wordpress
737125e61b9218fa0f89c7be98a1505e596cc961be8c004c3a0a5138a667d8a
[ec2-user@ip-172-31-32-58 ~]$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
737125e61b92        wordpress:latest   "/entrypoint.sh apac"   4 seconds ago     Up 3 seconds       0.0.0.0:49155->80/tcp   some-wordpress
dd1f4c9ba0c8        mysql:latest       "/entrypoint.sh myso"  About an hour ago  Up About an hour   3306/tcp           some-mysql,some-wordpress/mysql
[ec2-user@ip-172-31-32-58 ~]$ docker port 7371 80
0.0.0.0:49155
[ec2-user@ip-172-31-32-58 ~]$
```

In my case, it was port **49155**.

Enter your Docker hosts' **ip:port** in your web browser to see whether you can reach it. If you're on your local computer running Windows or OS X, you can find your Docker IP by invoking `boot2docker ip`. If you're on a local Linux, **127.0.0.1** should be fine.

I'm doing this on Amazon EC2, so I have to go to the EC2 Management console to get my public IP or public DNS.

Point your web browser to `http://yourip:yourport` (in my case, `http://myamazon-dns.com:49155`) and you should be presented with this:



The default Amazon AWS security policy is to block the default Docker public ports, so we have to change this in the **Security Groups** section in the EC2 dashboard. See the *Docker on Amazon EC2* section in *Chapter 1, Installing Docker*, for how to do this.

Wonderful, it works!

Summary

The Docker image can be seen as a read-only template for containers, specifying what's supposed to be installed, copied, configured, and exposed when a container is started.

We learned more about how we can interact with the Docker daemon and with individual Docker containers to read logs, copy files, and export the complete filesystem.

The Docker hub was introduced and we looked at what the official WordPress Docker image consisted of and how they configured the OS in the Dockerfile as well as in an `ENTRYPOINT` file to some extent.

We downloaded and ran the WordPress image that failed as expected, and we fixed it by linking the required MySQL container to it.

In the next chapter, we will create a Dockerfile and publish a Docker image to the Docker registry hub so that we have a way to get our customized Docker images to wherever we decide to place our PaaS.

3

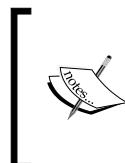
Creating Our First PaaS Image

You are now ready to write your own Dockerfiles, publish them to the Docker Registry Hub, and create containers for them. In this chapter you will:

- Build your own image on top of another
- Host your Dockerfiles in your GitHub account
- Publish an image on the Docker Registry Hub

The WordPress image

For this project, we are going to use the official WordPress Docker image as a base, which has Apache2 as its web server.



If you plan to host sites with a lot of traffic, I would recommend using an image based on Nginx instead of Apache2 as the web server. I have had great success running WordPress sites with Nginx and the memcached plugin, WP-FFPC. It can be a bit tricky to set up, and that's why it's out of the scope of this book.



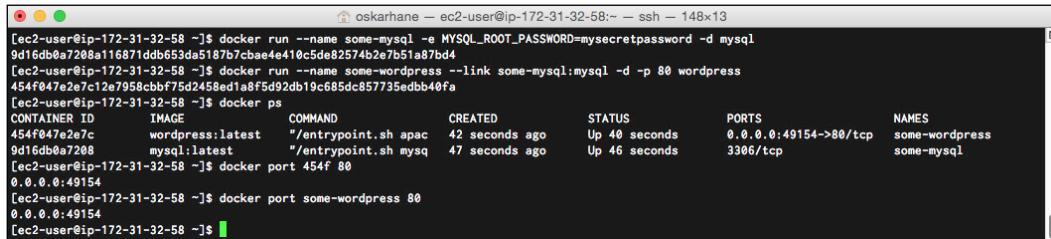
First of all, let's run a MySQL container and a WordPress container and link to them to see what happens:

```
docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=mysecretpassword -d mysql
docker run --name some-wordpress --link some-mysql:mysql -d -p 80 wordpress
```

Creating Our First PaaS Image

The `-p 80` option tells Docker to expose the private port 80 to the outer world. To find out which public port is bound to the private port 80, run `docker ps` command and look in the ports column or invoke the `docker port <container-ID | name> 80` command.

The screenshot is shown below:



```
oskarhan -- ec2-user@ip-172-31-32-58:~ - ssh - 148x13
[ec2-user@ip-172-31-32-58 ~]$ docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=myscretpassword -d mysql
9d16db0a7208a116871dd653da5187b7c8ae4410c5de82574b2e7b51a87bd4
[ec2-user@ip-172-31-32-58 ~]$ docker run --name some-wordpress --link some-mysql:mysql -d -p 80 wordpress
454f047e2e7c12e7958ccb7f5d2458ed1aef5d92db19c685dc857735edb40fa
[ec2-user@ip-172-31-32-58 ~]$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
454f047e2e7c        wordpress:latest   "/entrypoint.sh apac"   42 seconds ago    Up 40 seconds      0.0.0.0:49154->80/tcp   some-wordpress
9d16db0a7208        mysql:latest       "/entrypoint.sh mysq"  47 seconds ago    Up 46 seconds      3306/tcp           some-mysql
[ec2-user@ip-172-31-32-58 ~]$ docker port some-wordpress 80
0.0.0.0:49154
[ec2-user@ip-172-31-32-58 ~]$ docker port some-mysql 80
0.0.0.0:49154
[ec2-user@ip-172-31-32-58 ~]$
```

In my case, the public port is **49154**. Enter the full URL in the form of `http://public_ip:public_port` in your web browser. I'm doing this on an Amazon EC2 instance. I get a public domain, which is `http://ec2-54-187-234-27.us-west-2.compute.amazonaws.com:49154` in my case.

The screenshot is displayed below:



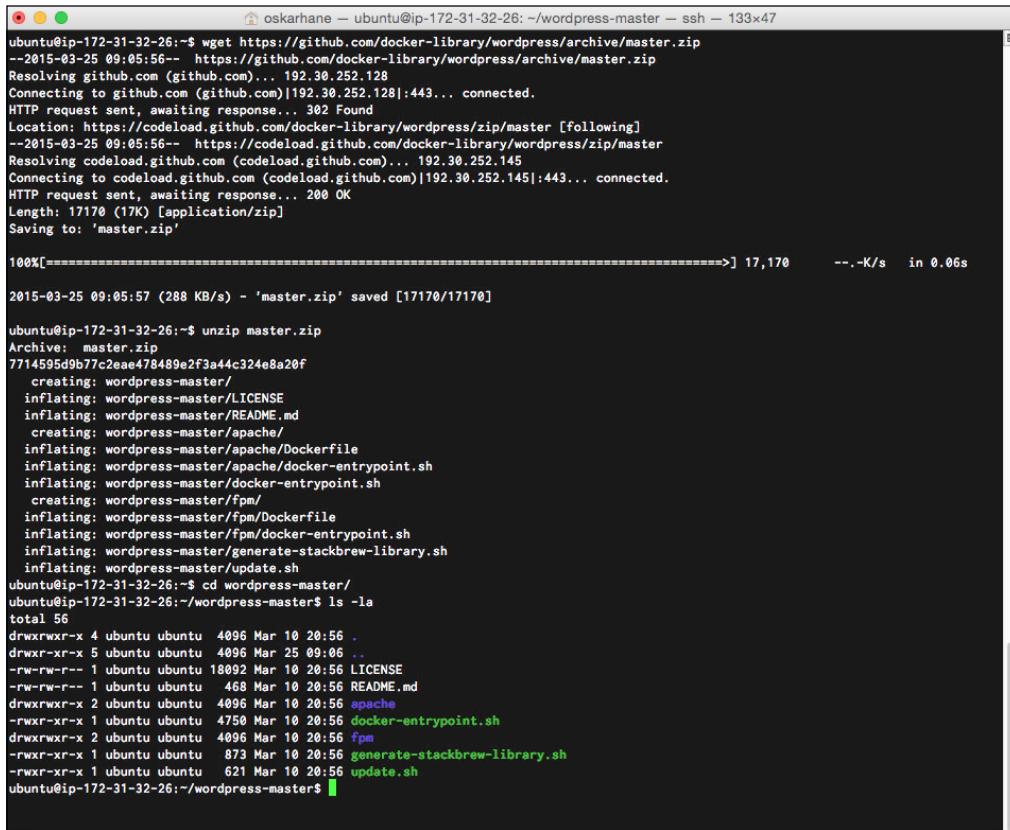
The WordPress installation page welcomes us, which means that the WordPress and the MySQL containers are working properly.

Moving from the defaults

Now we have a default installation of WordPress run on Apache2. Some WordPress plugins require you to make changes to the web server's configuration. How can we do that? What if we want to edit some of the files in the WordPress directory?

The first thing we need to do is to get our own copy of the official WordPress repository so that we can explore the Dockerfile. The current URL that is used to get the repository is <https://github.com/docker-library/wordpress>. Click on this link from the WordPress repo page on the Docker Registry Hub.

You can clone, fork, or just download the source for this Docker image. It doesn't matter how you get it because we're not going to use it later on. This image is for testing and exploring purposes. I used my EC2 instance to do this.



```

oskarhane@ip-172-31-32-26: ~/wordpress-master$ ssh - 133x47
ubuntu@ip-172-31-32-26:~$ wget https://github.com/docker-library/wordpress/archive/master.zip
--2015-03-25 09:05:56-- https://github.com/docker-library/wordpress/archive/master.zip
Resolving github.com (github.com)... 192.30.252.128
Connecting to github.com (github.com)|192.30.252.128|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://code.load.github.com/docker-library/wordpress/zip/master [following]
--2015-03-25 09:05:56-- https://code.load.github.com/docker-library/wordpress/zip/master
Resolving code.load.github.com (code.load.github.com)... 192.30.252.145
Connecting to code.load.github.com (code.load.github.com)|192.30.252.145|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 17170 (17K) [application/zip]
Saving to: "master.zip"

100%[=====] 17,170      --.-K/s   in 0.06s

2015-03-25 09:05:57 (288 KB/s) - 'master.zip' saved [17170/17170]

ubuntu@ip-172-31-32-26:~$ unzip master.zip
Archive: master.zip
7714595d9b77c2eae478489e2f3a44c324e8a20f
  creating: wordpress-master/
  inflating: wordpress-master/LICENSE
  inflating: wordpress-master/README.md
  creating: wordpress-master/apache/
  inflating: wordpress-master/apache/Dockerfile
  inflating: wordpress-master/apache/docker-entrypoint.sh
  inflating: wordpress-master/docker-entrypoint.sh
  creating: wordpress-master/fpm/
  inflating: wordpress-master/fpm/Dockerfile
  inflating: wordpress-master/fpm/docker-entrypoint.sh
  inflating: wordpress-master/generate-stackbrew-library.sh
  inflating: wordpress-master/update.sh
ubuntu@ip-172-31-32-26:~$ cd wordpress-master/
ubuntu@ip-172-31-32-26:~/wordpress-master$ ls -la
total 56
drwxrwxr-x 4 ubuntu ubuntu 4096 Mar 10 20:56 .
drwxr-xr-x 5 ubuntu ubuntu 4096 Mar 25 09:06 ..
-rw-rw-r-- 1 ubuntu ubuntu 18092 Mar 10 20:56 LICENSE
-rw-rw-r-- 1 ubuntu ubuntu 468 Mar 10 20:56 README.md
drwxrwxr-x 2 ubuntu ubuntu 4096 Mar 10 20:56 apache
-rw-rxr-- 1 ubuntu ubuntu 4750 Mar 10 20:56 docker-entrypoint.sh
drwxrwxr-x 2 ubuntu ubuntu 4096 Mar 10 20:56 fpm
-rwxr-xr-x 1 ubuntu ubuntu 873 Mar 10 20:56 generate-stackbrew-library.sh
-rwxr-xr-x 1 ubuntu ubuntu 621 Mar 10 20:56 update.sh
ubuntu@ip-172-31-32-26:~/wordpress-master$ 
```

Open the file in any text editor to view its content. If you are—like me—using the terminal, you can use `vi apache/Dockerfile` to open it in the `vi` file editor. The current Dockerfile for the official WordPress image looks like this:

```
FROM php:5.6-apache
RUN a2enmod rewrite
# install the PHP extensions we need
RUN apt-get update && apt-get install -y libpng12-dev libjpeg-dev && rm
-rf /var/lib/apt/lists/* \
&& docker-php-ext-configure gd --with-png-dir=/usr --with-jpeg-dir=/
usr \
&& docker-php-ext-install gd
RUN docker-php-ext-install mysqli
VOLUME /var/www/html
ENV WORDPRESS_VERSION 4.1.1
ENV WORDPRESS_UPSTREAM_VERSION 4.1.1
ENV WORDPRESS_SHA1 15d38fe6c73121a20e63ccd8070153b89b2de6a9
# upstream tarballs include ./wordpress/ so this gives us /usr/src/
wordpress
RUN curl -o wordpress.tar.gz -SL https://wordpress.org/wordpress-
${WORDPRESS_UPSTREAM_VERSION}.tar.gz \
&& echo "$WORDPRESS_SHA1 *wordpress.tar.gz" | shasum -c - \
&& tar -xzf wordpress.tar.gz -C /usr/src/ \
&& rm wordpress.tar.gz
COPY docker-entrypoint.sh /entrypoint.sh
# grr, ENTRYPOINT resets CMD now
ENTRYPOINT ["/entrypoint.sh"]
CMD ["apache2-foreground"]
```

This image uses the `php:5.6-apache` image as a base and downloads and extracts WordPress 4.1 to `/usr/src/wordpress`. Then it adds an `ENTRYPOINT` and starts Apache2 in the foreground.

Our objective

To make this WordPress image useable for more than demo purposes, we need to modify the Dockerfile in three ways. Our objectives are as follows:

- Preparing Apache for caching (through the WP Super Cache plugin)
- Raising the upload limit in both PHP and Apache2
- Installing two plugins: WP Super Cache and WP Mail SMTP

Preparing for caching

There are two small steps to be performed to obtain website caching through WP Super Cache – we need to enable the `mod_headers` and `mod_expires` modules in Apache2.

On line 5 in the Dockerfile, you can see `RUN a2enmod rewrite`. The `a2enmod` command enables modules in Apache2, and modules are disabled by the `a2dismod` command. Enabling our desired modules is as easy as appending them to that line:

```
RUN a2enmod rewrite expires headers
```

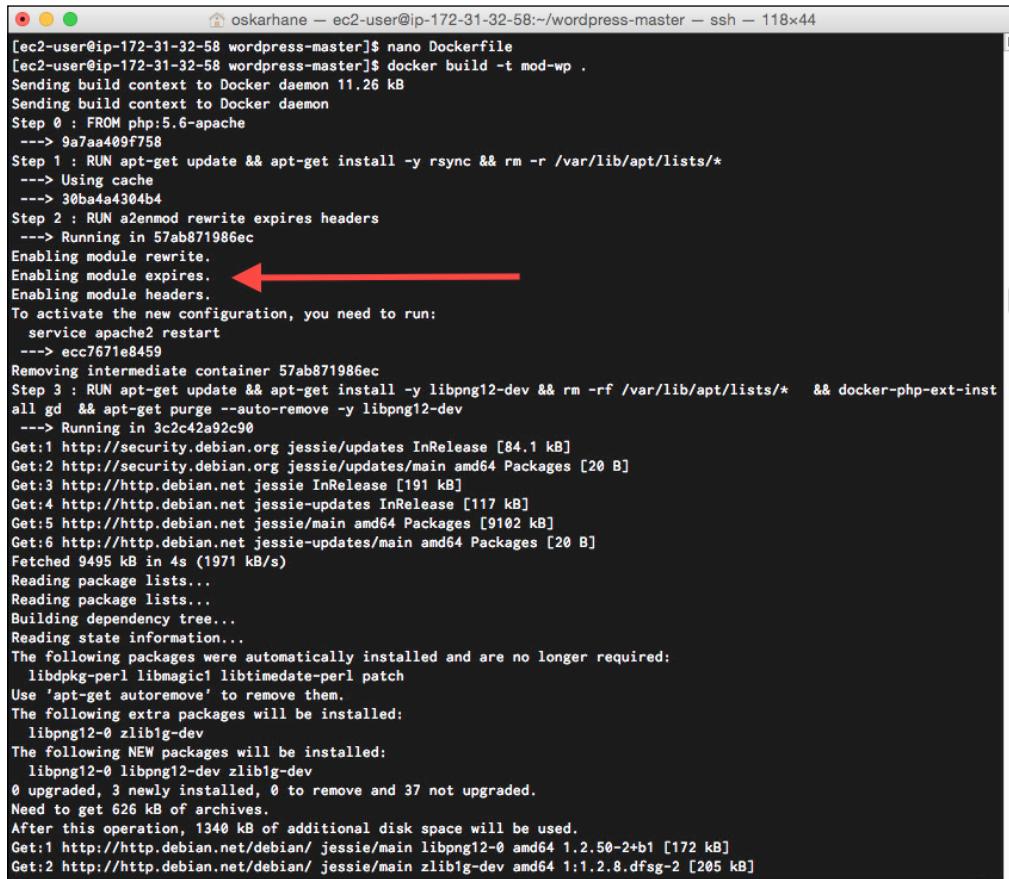
We make those edits, build a new image, and see what happens. It takes a long time to build these images, since PHP is built from source. What we are looking for are lines that state that our modules are enabled. They will show up for just a few seconds in the build process.

You initiate a build from a Dockerfile by executing this:

```
docker build -t mod-wp .
```

The `-t mod-wp` command sets the name of our new image to `mod-wp`.

The screenshot is shown below:



```
[ec2-user@ip-172-31-32-58 wordpress-master]$ nano Dockerfile
[ec2-user@ip-172-31-32-58 wordpress-master]$ docker build -t mod-wp .
Sending build context to Docker daemon 11.26 kB
Sending build context to Docker daemon
Step 0 : FROM php:5.6-apache
--> 9a7aaa09f758
Step 1 : RUN apt-get update && apt-get install -y rsync && rm -r /var/lib/apt/lists/*
--> Using cache
--> 30ba4a4304b4
Step 2 : RUN a2enmod rewrite expires headers
--> Running in 57ab871986ec
Enabling module rewrite.
Enabling module expires. ←
Enabling module headers.
To activate the new configuration, you need to run:
service apache2 restart
--> ecc7671e8459
Removing intermediate container 57ab871986ec
Step 3 : RUN apt-get update && apt-get install -y libpng12-dev && rm -rf /var/lib/apt/lists/* && docker-php-ext-inst
all gd && apt-get purge --auto-remove -y libpng12-dev
--> Running in 3c2c42a92c90
Get:1 http://security.debian.org jessie/updates InRelease [84.1 kB]
Get:2 http://security.debian.org jessie/main amd64 Packages [20 B]
Get:3 http://http.debian.net jessie InRelease [191 kB]
Get:4 http://http.debian.net jessie-updates InRelease [117 kB]
Get:5 http://http.debian.net jessie/main amd64 Packages [9102 kB]
Get:6 http://http.debian.net jessie-updates/main amd64 Packages [20 B]
Fetched 9495 kB in 4s (1971 kB/s)
Reading package lists...
Reading package lists...
Building dependency tree...
Reading state information...
The following packages were automatically installed and are no longer required:
libdpkg-perl libmagic1 libtimedate-perl patch
Use 'apt-get autoremove' to remove them.
The following extra packages will be installed:
libpng12-0 zlib1g-dev
The following NEW packages will be installed:
libpng12-0 libpng12-dev zlib1g-dev
0 upgraded, 3 newly installed, 0 to remove and 37 not upgraded.
Need to get 626 kB of archives.
After this operation, 1340 kB of additional disk space will be used.
Get:1 http://http.debian.net/debian/ jessie/main libpng12-0 amd64 1:2.5.0-2+b1 [172 kB]
Get:2 http://http.debian.net/debian/ jessie/main zlib1g-dev amd64 1:1.2.8.dfsg-2 [205 kB]
```

The build should run through the whole process without any errors, and then the preparation for the cache plugin is done.

Raising the upload limit

The default upload size is limited to 2 MB by PHP. This limit is too low, especially since blogging from mobile phones is popular and the size of a mobile phone photo or video is often bigger than this. I would like to have the option to upload videos directly on my blogs, and they can be up to 32 MB.

For this limit to be raised, we need to change the limit for two parameters in the PHP configuration file: `upload_max_filesize` and `post_max_size`.

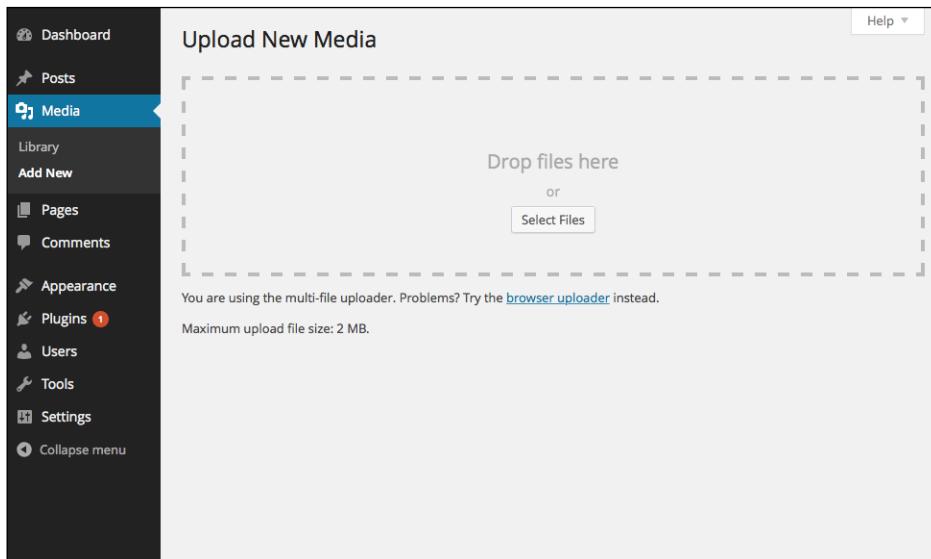
Looking at the `php:5.6-Apache` image, which is the base image of the WordPress image, Dockerfile we see that it runs Debian and PHP configuration files are supposed to be in the `/usr/local/etc/php/conf.d/` directory. This means that if we add a file to that directory, it should get read in and parsed.



The Dockerfile for PHP 5.6 can be found at <https://github.com/docker-library/php/blob/master/5.6/Dockerfile>.



To verify that the upload limit is as low as said before, I started and installed an unmodified WordPress container. Then I clicked on the **Add new media** button.



It says that the upload limit is 2 MB.

Let's add a configuration file named `upload-limit.ini` to the configuration directory, and add the two parameters to the file.

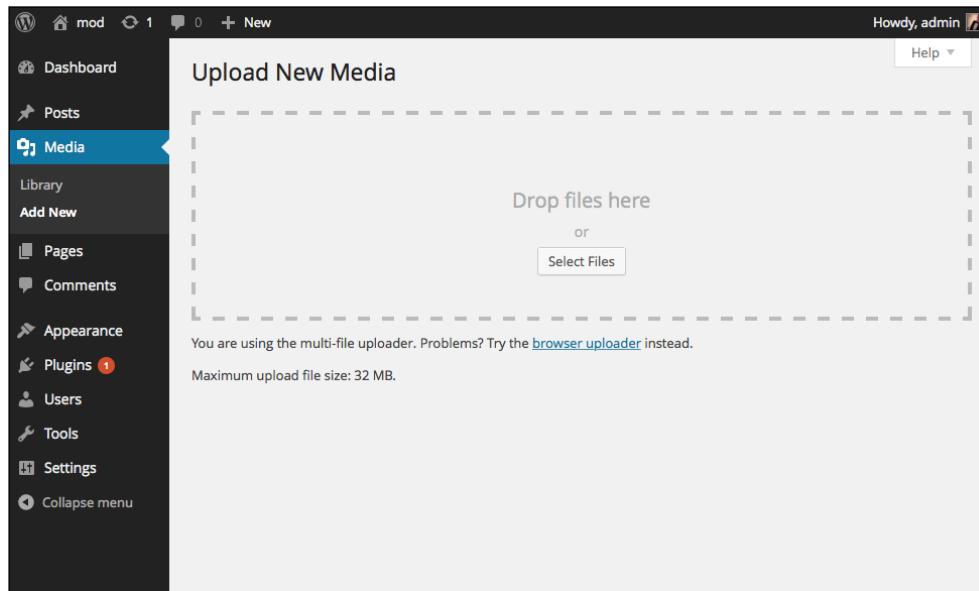
These commands, all of which should be on a single line, are added to our Dockerfile right above the line we modified when preparing Apache for caching:

```
RUN touch /usr/local/etc/php/conf.d/upload-limit.ini \
    && echo "upload_max_filesize = 32M" >> /usr/local/etc/php/conf.d/
upload-limit.ini \
    && echo "post_max_size = 32M" >> /usr/local/etc/php/conf.d/
upload-limit.ini
#Paste above this line.
RUN a2enmod rewrite expires headers
```

Once again, build the image to ensure that no errors are produced. If you get an error saying that the image name already exists, you can delete the old image with the `docker rmi` command or change the name to `mod-wp:latest`, which will update the image's tag to `latest`.

When the build finishes, we run a new container from the new image to check out what the WordPress administration interface says. We can run a container from our new image, like this:

```
docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=mysecretpassword -d
mysql
docker run --name some-wordpress --link some-mysql:mysql -d -p 80 mod-
wp:latest
```



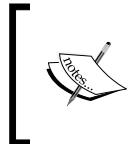
We can now see that we can upload bigger files. Just to verify, if you upload a file bigger than 2 MB, it will prove that the limit has been raised.

Plugin installation

Here, we are going to download and install two plugins that we want in all our future WordPress sites. All the tasks for these plugins will be done in the entry point file, since we have to edit a few files in the WordPress installation.

The first plugin is WP Super Cache. We prepared Apache2 for this earlier, and now it's time to use that. With this plugin, our site will run faster and demand fewer resources from our host.

The second plugin is WP Mail SMTP, with the help of which WordPress can send outgoing e-mails. This container does not (and should not) include a mail server. With this plugin, we can make WordPress send e-mails via an external SMTP (Gmail, your ISPs, or anything else).



Even though I have hosted and managed my own mail server for a few years now, it is a hassle with keeping it up to date and managing spam filters and redundancy. We're better off leaving that to the specialists.



All plugins will be downloaded with CURL and unpacked with unzip. CURL is already installed but unzip is not, so we have to add it to our Dockerfile, close to the top where the `apt-get install` command is running:

```
RUN apt-get update && apt-get install -y unzip rsync && rm -r /var/lib/apt/lists/*
```

If we don't do this, we will get error messages during the build process.

Since there are two plugins we have to download, extract, and activate, we will create a function in the `docker-entrypoint.sh` file.

This function will go to Wordpress' plugin site and look for the download URL for the latest version of the plugin. It will download and then extract it to the plugin folder in our Wordpress installation:

```
dl_and_move_plugin() {  
    name="$1"  
    curl -O $(curl -i -s "https://wordpress.org/plugins/$name/" | egrep  
-o "https://downloads.wordpress.org/plugin/[^\']+")  
    unzip -o "$name".*.zip -d $(pwd)/wp-content/plugins  
}
```

Now that we have the function there, we can add these lines near the end of the file, just above the line that says `chown -R www-data:www-data ...`:

```
dl_and_move_plugin "wp-super-cache"  
dl_and_move_plugin "wp-mail-smtp"
```

Place the function and the function calls close to the bottom—in the `docker-entrypoint.sh` file, just above the `exec` command.

We will build the image again and start a container so that we can verify that everything is working as we want:

```
docker build -t mod-wp:latest
```

This will take a while, and when it's ready, you can fire up a MySQL container and a mod-wp container:

```
docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=mysecretpassword -d  
mysql  
docker run --name some-wordpress --link some-mysql:mysql -d -p 80 mod-  
wp:latest
```

If you get an error that tells you that you already have a container with that name, either remove the old container with `docker rm some-wordpress` or use another name for the new container.

Creating Our First PaaS Image

Get the port by invoking `docker ps`, and look for the port binding to port 80 on the WordPress container. Then load the URL into your browser. This time, install WordPress, log in, and go to the plugins page, as shown in the following screenshot:

The screenshot shows the WordPress admin interface with the Plugins page selected. The sidebar on the left includes links for Dashboard, Posts, Media, Pages, Comments, Appearance, Plugins (with 1 new item), and Settings. The main content area displays a table of installed plugins:

Plugin	Description
Akismet	Used by millions, Akismet is quite possibly the best way in the world to protect your blog from comment and trackback spam. It keeps your site protected from spam even while you sleep. To get started: 1) Click the "Activate" link to the left of this description, 2) Sign up for an Akismet API key, and 3) Go to your Akismet configuration page, and save your API key. Version 3.0.2 By Automattic View details
Hello Dolly	This is not just a plugin, it symbolizes the hope and enthusiasm of an entire generation summed up in two words sung most famously by Louis Armstrong: Hello, Dolly. When activated you will randomly see a lyric from Hello, Dolly in the upper right of your admin screen on every page. Version 1.6 By Matt Mullenweg View details
Two Factor Auth	Secure your WordPress login with two factor auth. Users will be prompted with a page to enter a One Time Password when they login. Version 4.4 By Oskar Hane View details
WP-Mail-SMTP	Reconfigures the <code>wp_mail()</code> function to use SMTP instead of <code>mail()</code> and creates an options page to manage the settings. Version 0.9.5 By Callum Macdonald View details
WP Super Cache	Very fast caching plugin for WordPress. Version 1.4.2 By Automattic View details

This looks just like we want it to! Great!

Let's go ahead and activate and set up these plugins just to verify that they work. Start with the WP Mail SMTP plugin. I will use my Gmail account as the sender, but you can choose which SMTP you want. Here is a screenshot showing the settings for Gmail:

The screenshot shows the 'Advanced Email Options' settings page in the WordPress admin. The left sidebar is dark-themed, showing the 'Settings' menu item selected under 'Email'. The main area has a light background.

From Email: oskar.hane@gmail.com (with a note: You can specify the email address that emails should be sent from. If you leave this blank, the default email will be used.)

From Name: WP (with a note: You can specify the name that emails should be sent from. If you leave this blank, the emails will be sent from WordPress.)

Mailer: Send all WordPress emails via SMTP (radio button selected)

Return Path: Set the return-path to match the From Email (checkbox checked)

SMTP Options: These options only apply if you have chosen to send mail by SMTP above.

- SMTP Host:** smtp.gmail.com
- SMTP Port:** 587
- Encryption:** Use TLS encryption. This is not the same as STARTTLS. For most servers SSL is the recommended option. (radio button selected)
- Authentication:** Yes: Use SMTP authentication. (radio button selected) (Note: If this is set to no, the values below are ignored.)
- Username:** oskar.hane@gmail.com
- Password:** hunter10

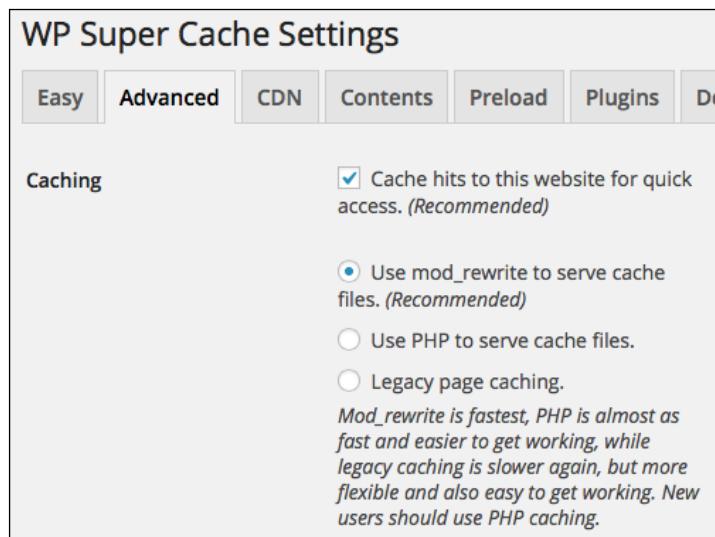
Save Changes button at the bottom.

From the bottom of this page, you can send a test e-mail. I strongly recommend doing this because Gmail sometimes blocks new SMTP clients. If you get an error message saying **Please log in via your web browser and then try again**, you've triggered that. In that case, you'll soon get an e-mail from Google explaining suspicious activity and asking you to go through a few steps to make it work. This is annoying but it's a good thing.

Now let's move on to the WP Super Cache plugin. Go ahead and activate the plugin from the plugin page. Before we can enable it, we have to go to **Settings | Permalinks**, check the **Post name** button, and save.

Then go to **Settings | WP Super Cache**.

Click on **Caching On** and then on **Update Status**. Now click on the **Advanced** tab and enable **mod_rewrite caching**, as shown:



Scroll down to the **Miscellaneous** section and check the boxes that are shown in the following screenshot. If you want to know exactly what all of these checkboxes do, you can refer to the plugins' documents.

Miscellaneous	<p><input checked="" type="checkbox"/> Compress pages so they're served more quickly to visitors. <i>(Recommended)</i></p> <p><i>Compression is disabled by default because some hosts have problems with compressed files. Switching it on and off clears the cache.</i></p> <p><input type="checkbox"/> 304 Not Modified browser caching. Indicate when a page has not been modified since last requested. <i>(Recommended)</i></p> <p>Warning! 304 browser caching is only supported when not using mod_rewrite caching.</p> <p><input checked="" type="checkbox"/> Don't cache pages for known users. <i>(Recommended)</i></p> <p><input type="checkbox"/> Don't cache pages with GET parameters. (?x=y at the end of a url)</p> <p><input type="checkbox"/> Make known users anonymous so they're served supercached static files.</p> <p><input checked="" type="checkbox"/> Cache rebuild. Serve a supercache file to anonymous users while a new file is being generated. <i>(Recommended)</i></p> <p><input type="checkbox"/> Proudly tell the world your server is Stephen Fry proof! (places a message in your blog's footer)</p>
----------------------	---

When you've saved this, you'll get a notice at the top saying that you need to update the rewrite rules, as shown:

WP Super Cache Settings

Rewrite rules must be updated

The rewrite rules required by this plugin have changed or are missing. Scroll down the Advanced Settings page and click the [Update Mod_Rewrite Rules](#) button.

Scroll down the page and click on the **Update Mod_Rewrite Rules** button to update the rewrite rules, as shown:



```
</IfModule>
<IfModule mod_deflate.c>
    SetEnvIfNoCase Request_URI \.gz$ no-gzip
</IfModule>
<IfModule mod_headers.c>
    Header set Vary "Accept-Encoding, Cookie"
    Header set Cache-Control 'max-age=3, must-revalidate'
</IfModule>
<IfModule mod_expires.c>
    ExpiresActive On
    ExpiresByType text/html A3
</IfModule>
# END supercache
```

Update Mod_Rewrite Rules

The cache plugins' status should now be green, and all of the setup should be done. Since we are logged in to this web browser, we will not be served cached pages. This is important to know, and the advantage is that you won't have to disable the whole cache plugin just to see the uncached version of your site. Open another web browser (not just another window or tab in your current browser, unless you are using incognito or private mode) and go to your WordPress instance. Click on the **Hello World** title on the post. Go back to the start page. Click on the title again. It feels pretty fast, right?

To verify that it works, you can open the development tools in your browser. Make sure that you don't have caching disabled in your browser when the development tools are open. Click on the **Network** tab, then click on the post's title again, and then inspect that call, as shown in the following screenshot:

Remote Address: 54.148.253.187:49156
 Request URL: http://ec2-54-148-253-187.us-west-2.compute.amazonaws.com:49156/hello-world/
 Request Method: GET
 Status Code: 200 OK

Request Headers

- Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
- Accept-Encoding: gzip, deflate, sdch
- Accept-Language: sv-SE,sv;q=0.8,en-US;q=0.6,en;q=0.4
- Connection: keep-alive
- Host: ec2-54-148-253-187.us-west-2.compute.amazonaws.com:49156
- If-Modified-Since: Mon, 24 Nov 2014 19:56:10 GMT
- Referer: http://ec2-54-148-253-187.us-west-2.compute.amazonaws.com:49156/
- User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.65 Safari/537.36

Response Headers

- Cache-Control: max-age=3, must-revalidate
- Connection: Keep-Alive
- Content-Encoding: gzip
- Content-Length: 3565
- Content-Type: text/html; charset=UTF-8
- Date: Mon, 24 Nov 2014 19:57:26 GMT
- Keep-Alive: timeout=5, max=95
- Last-Modified: Mon, 24 Nov 2014 19:56:10 GMT
- Server: Apache/2.4.10 (Debian) PHP/5.6.3
- Vary: Accept-Encoding, Cookie
- WP-Super-Cache: Served supercache file from PHP
- X-Powered-By: PHP/5.6.3

This is just what we wanted to see. Great!

Making our changes persist

Now that we have made our changes, we want to create our own Dockerfile to build on top of the official WordPress image.

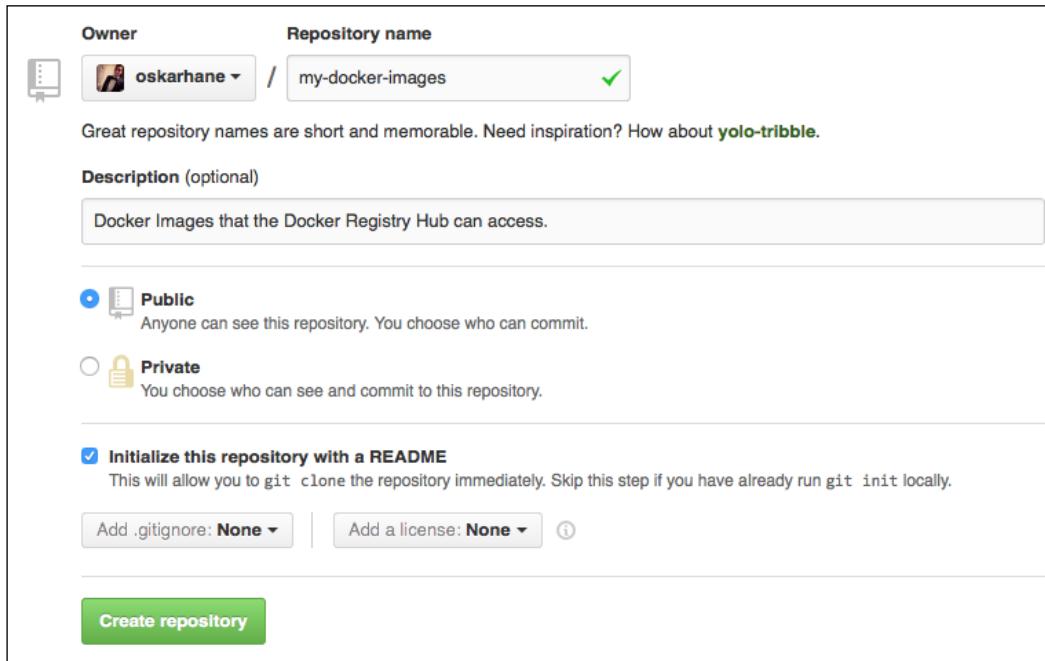
This is what the Dockerfile should look like:

```
FROM wordpress:latest
RUN apt-get update && apt-get install -y unzip && rm -r /var/lib/apt/lists/*
RUN touch /usr/local/etc/php/conf.d/upload-limit.ini \
    && echo "upload_max_filesize = 32M" >> /usr/local/etc/php/conf.d/upload-limit.ini \
    && echo "post_max_size = 32M" >> /usr/local/etc/php/conf.d/upload-limit.ini
```

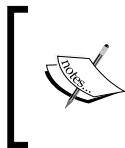
```
RUN a2enmod expires headers
VOLUME /var/www/html
COPY docker-entrypoint.sh /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]
CMD ["apache2", "-DFOREGROUND"]
```

Hosting image sources on GitHub

The Docker Registry Hub has very good support for automatic fetching of image updates from both Bitbucket and GitHub. You can pick whatever you want, but for this book, I will use GitHub. I have accounts on both services and they are both excellent.



At GitHub, create a new empty repository called `my-docker-images` and add an appropriate license if you like.



This book will not go into how to add your SSH keys to GitHub and so on. There are excellent guides for this online. GitHub has a great guide at <https://help.github.com/articles/generating-ssh-keys/>.



Let's create a branch and copy our files for the modified Docker image to it.

Clone the repository locally so that you can add files to it. Make sure you are not inside your `wordpress-master` directory, but on the same level as it is:

```
git clone git@github.com:yourusername/my-docker-images.git
```

The output of this command is as follows:

```
oskarhane — ec2-user@ip-172-31-32-58:~/my-docker-images — ssh — 100x18
[ec2-user@ip-172-31-32-58 ~]$ git clone git@github.com:oskarhane/my-docker-images.git
Cloning into 'my-docker-images'...
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (3/3), done.
Checking connectivity... done.
[ec2-user@ip-172-31-32-58 ~]$ cd my-docker-images && ls -la
total 16
drwxrwxr-x 3 ec2-user ec2-user 4096 Nov 17 20:59 .
drwxr----- 5 ec2-user ec2-user 4096 Nov 17 20:59 ..
drwxrwxr-x 8 ec2-user ec2-user 4096 Nov 17 20:59 .git
-rw-rw-r-- 1 ec2-user ec2-user 90 Nov 17 20:59 README.md
[ec2-user@ip-172-31-32-58 my-docker-images]$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
[ec2-user@ip-172-31-32-58 my-docker-images]$
```

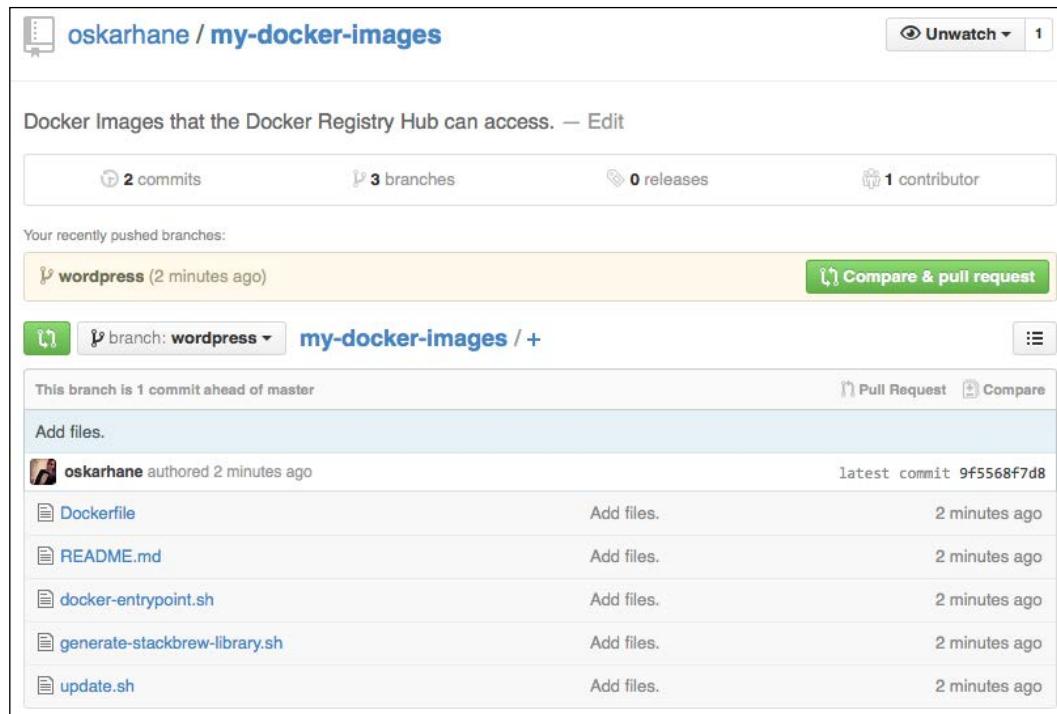
We'll execute these commands one by one:

```
cd my-docker-images
git checkout -b wordpress
git add .
git commit -m "Adding new files."
git push origin wordpress
```

Go to your GitHub page and try to find the WordPress branch.

Creating Our First PaaS Image

For every new Docker image we want to create and publish on the Docker Registry Hub, we need to create a new branch in this GitHub repository. If you have a lot of Docker images and the images have a lot of versions, you might want to consider a different structure, but for this book, this approach will be great!



Docker Images that the Docker Registry Hub can access. — Edit

2 commits 3 branches 0 releases 1 contributor

Your recently pushed branches:

wordpress (2 minutes ago) Compare & pull request

This branch is 1 commit ahead of master Pull Request Compare

Add files.

oskarhane authored 2 minutes ago	latest commit 9f5568f7d8
Dockerfile	Add files.
README.md	Add files.
docker-entrypoint.sh	Add files.
generate-stackbrew-library.sh	Add files.
update.sh	Add files.

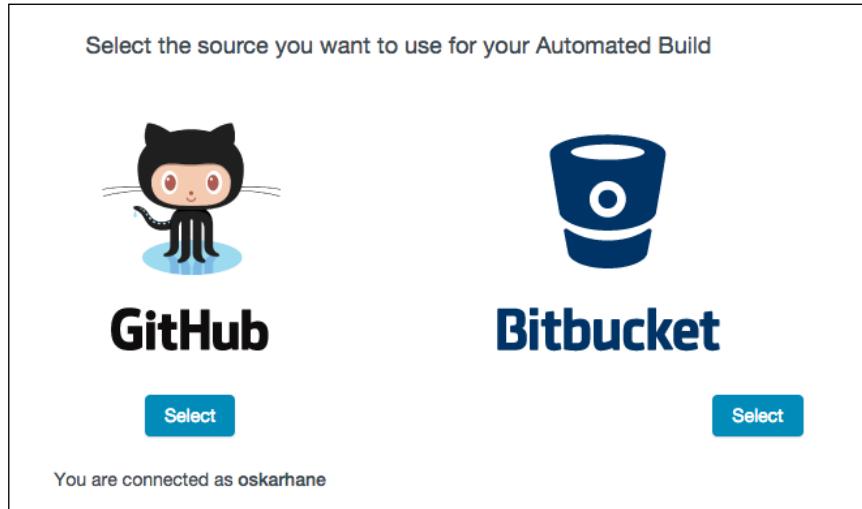
All files are in place, and you can click on them to verify that the contents are what we would expect.

Publishing an image on the Docker Registry Hub

If you're not a member of the Docker Registry Hub (<https://hub.docker.com>), now is the time to register so that you can publish your images on the public Docker repository, which can be accessed from anywhere.

Automated builds

When you add a repository, you should choose the **Automated Build** option so that you can fetch code from GitHub (or Bitbucket), as shown in the following screenshot:



We'll connect with our GitHub account and select the repository we just created and pushed to `my-docker-images`.

We will start to add our WordPress image, so let's set the repository name to `wordpress` on the next screen. It's important that you enter this name correctly, since it cannot be changed later.

At this time, we will just use one tag for our image—the **latest** tag. Ensure that the source: **Type** is set to **Branch** and that you've entered `wordpress` as its name.

Creating Our First PaaS Image

Choose to add this as a public repository and check the **active** checkbox. This means that if you push any updates to this on GitHub, the Registry Hub will automatically pull it and publish its changes, as shown in the following screenshot:

Namespace (optional) and Repository Name

oskarhane	/	wordpress	<input checked="" type="checkbox"/>
-----------	---	-----------	-------------------------------------

New unique Repo name; 3 - 30 characters. Only lowercase letters, digits and _ - . characters are allowed

Tags

Type	Name	Dockerfile Location	Docker Tag Name
Branch	wordpress	/	latest

Public
任何人都可以拉取，并且可以在 Docker Index 上搜索到。
 Private
只有你才能拉取，且不在 Docker Index 上列出。

Active:

When active we will build when new pushes occur

Create Repository

The Registry Hub will now pull your branch and try to build your Docker image to verify that it works. You can head over to the **Build Details** tab to see the progress. Since it's the official WordPress image base, it should go pretty fast if they cache the images on their build servers. If not, it could take a few minutes, since PHP is compiled from source.

This is shown in the following screenshot:

The screenshot shows a card on a Docker Registry Hub interface. At the top, it says "No description set" with social sharing icons (0 stars, 0 comments, 0 saves). Below this is a navigation bar with tabs: "Information", "Dockerfile" (which is active), "Build Details", and "Tags". The "Build Details" tab contains the following information:

Type	Name	Dockerfile Location	Tag Name
Branch	wordpress	/	latest

Below this is a section titled "Builds History" with a table:

build Id	Status	Created Date	Last Updated
bdebwe26bizth7xbhr9vsyc	Building	2014-11-24 22:59:33	2014-11-24 23:00:03

Wow! We've just published an image on the Docker Registry Hub, which means that anyone can fetch and run containers on top of it. The status will go from **Building** to **Finished** when the image is published.

The next step would be to actually pull it ourselves to verify that it works as expected:

```
docker pull oskarhane/wordpress
docker images
docker run --name mysql -e MYSQL_ROOT_PASSWORD=mysecretpassword -d
mysql
docker run --name my-wordpress --link mysql:mysql -d -p 80 oskarhane/
wordpress
docker ps
```

Open your web browser and head over to your new container. You should be presented with the WordPress setup page.

Summary

In this chapter, you learned quite a lot. The most part was about modifying the Dockerfile and `ENTRYPOINT` files in order to get the Docker image that we wanted. Bash knowledge and programming skills are very convenient, but since all of this is mostly about installation, moving files, and editing settings files, very basic knowledge can be enough.

GitHub is an excellent place to host your Docker repositories, and it's very easy to set up a new repository to get started. The Docker Registry Hub takes your GitHub repository and lets you pick a branch. This branch will be the source for a public Docker image that anyone can pull and use.

One question arises though; what about our data? It's trapped inside these MySQL and WordPress containers. The next chapter will show you how to handle your data.

4

Giving Containers Data and Parameters

The WordPress data inside the WordPress container and the database's data inside the MySQL container may not be what we want. It's considered good practice to keep the data outside the service containers because you may want to separate the data from the service container. In this chapter, we'll cover the following topics:

- Data volumes
- Creating a data volume image
- Host on GitHub
- Publishing on Docker Registry Hub
- Running on Docker Registry Hub
- Passing parameters to containers
- Creating a parameterized image

Data volumes

There are two ways in which we can mount external volumes on our containers. A data volume lets you share data between containers, and the data inside the data volume is untouched if you update, stop, or even delete your service container.

A data volume is mounted with the `-v` option in the `docker run` statement:

```
docker run -v /host/dir:container/dir
```

You can add as many data volumes as you want to a container, simply by adding multiple `-v` directives.

A very good thing about data volumes is that the containers that get data volumes passed into them don't know about it, and don't need to know about it either. No changes are needed for the container; it works just as if it were writing to the local filesystem. You can override existing directories inside containers, which is a common thing to do. One usage of this is to have the web root (usually at /var/www inside the container) in a directory at the Docker host.

Mounting a host directory as a data volume

You can mount a directory (or file) from your host on your container:

```
docker run -d --name some-wordpress -v /home/web/wp-one:/var/www wordpress
```

This will mount the host's local directory, /home/web/wp-one, as /var/www on the container. If you want to give the container only the read permission, you can change the directive to -v /home/web/wp-one:/var/www:ro where the :ro is the read-only flag.

It's not very common to use a host directory as a data volume in production, since data in a directory isn't very portable. But it's very convenient when testing how your service container behaves when the source code changes.

Any change you make in the host directory is direct in the container's mounted data volume.

Mounting a data volume container

A more common way of handling data is to use a container whose only task is to hold data. The services running in the container should be as few as possible, thus keeping it as stable as possible.

Data volume containers can have exposed volumes via the Dockerfile's VOLUME keyword, and these volumes will be mounted on the service container while using the data volume container with the --volumes-from directive.

A very simple Dockerfile with a VOLUME directive can look like this:

```
FROM ubuntu:latest
VOLUME ["/var/www"]
```

A container using the preceding Dockerfile will mount /var/www. To mount the volumes from a data container onto a service container, we create the data container and then mount it, as follows:

```
docker run -d --name data-container our-data-container
docker run -d --name some-wordpress --volumes-from data-container
wordpress
```

Backing up and restoring data volumes

Since the data in a data volume is shared between containers, it's easy to access the data by mounting it onto a temporary container. Here's how you can create a .zip file (from your host) from the data inside a data volume container that has VOLUME ["/var/www"] in its Dockerfile:

```
docker run --volumes-from data-container -v $(pwd):/
host ubuntu zip -r /host/data-containers-www /var/www
```

This creates a .zip file named data-containers-www.zip, containing what was in the www data container from var directory. This .zip file places that content in your current host directory.

Creating a data volume images

Since our data volume container will just hold our data, we should keep it as small as possible to start with so that it doesn't take lots of unnecessary space on the server. The data inside the container can, of course, grow to be as big as the space on the server's disk. We don't need anything fancy at all; we just need a working file storage system.

For this book, we'll keep all our data (MySQL database files and WordPress files) in the same container. You can, of course, separate them into two data volume containers named something like dbdata and webdata.

Data volume image

Our data volume image does not need anything other than a working filesystem that we can read and write to. That's why our base image of choice will be BusyBox. This is how BusyBox describes itself:

"BusyBox combines tiny versions of many common UNIX utilities into a single small executable. It provides replacements for most of the utilities you usually find in GNU fileutils, shellutils, etc. The utilities in BusyBox generally have fewer options than their full-featured GNU cousins; however, the options that are included provide the expected functionality and behave very much like their GNU counterparts. BusyBox provides a fairly complete environment for any small or embedded system."

That sounds great! We'll go ahead and add this to our Dockerfile:

```
FROM busybox:latest
```

Exposing mount points

There is a VOLUME instruction for the Dockerfile, where you can define which directories to expose to other containers when this data volume container is added using --volumes-from attribute. In our data volume containers, we first need to add a directory for MySQL data. Let's take a look inside the MySQL image we will be using to see which directory is used for the data storage, and expose that directory to our data volume container so that we can own it:

```
RUN mkdir -p /var/lib/mysql  
VOLUME ["/var/lib/mysql"]
```

We also want our WordPress installation in this container, including all .php files and graphic images. Once again, we go to the image we will be using and find out which directory will be used. In this case, it's /var/www/html. When you add this to the Dockerfile, don't add new lines; just append the lines with the MySQL data directory:

```
RUN mkdir -p /var/lib/mysql && mkdir -p /var/www/html  
VOLUME ["/var/lib/mysql", "/var/www/html"]
```

The Dockerfile

The following is a simple Dockerfile for the data image:

```
FROM busybox:latest
MAINTAINER Oskar Hane <oh@oskarhane.com>
RUN mkdir -p /var/lib/mysql && mkdir -p /var/www/html
VOLUME ["/var/lib/mysql", "/var/www/html"]
```

And that's it! When publishing images to the Docker Registry Hub, it's good to include a MAINTAINER instruction in the Dockerfiles so that you can be contacted if someone wants, for some reason.

Hosting on GitHub

When we use our knowledge on how to host Docker image sources on GitHub and how to publish images on the Docker Registry Hub, it'll be no problem creating our data volume image.

Let's create a branch and a Dockerfile and add the content for our data volume image:

```
git checkout -b data
vi Dockerfile
git add Dockerfile
```

On line number 2 in the preceding code, you can use the text editor of your choice. I just happen to find vi suits my needs. The content you should add to the Dockerfile is this:

```
FROM busybox:latest
MAINTAINER Oskar Hane <oh@oskarhane.com>
RUN mkdir /var/lib/mysql && mkdir /var/www/html
VOLUME ["/var/lib/mysql", "/var/www/html"]
```

Replace the maintainer information with your name and e-mail.

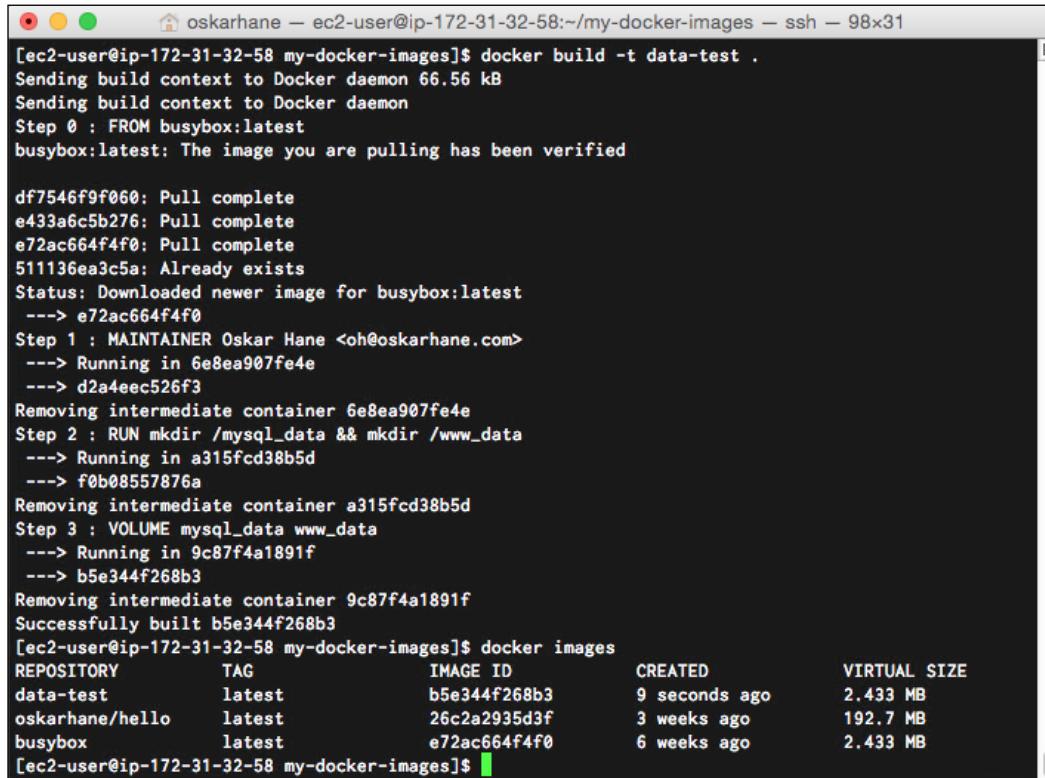
You can—and should—always ensure that it works before committing and pushing to GitHub. To do so, you need to build a Docker image from your Dockerfile:

```
docker build -t data-test .
```

Giving Containers Data and Parameters

Make sure you notice the dot at the end of the line, which means that Docker should look for a Dockerfile in the current directory. Docker will try to build an image from the instructions in our Dockerfile. It should be pretty fast, since it's a small base image and there's nothing but a couple of VOLUME instructions on top of it.

The screenshot is as follows:



```
[ec2-user@ip-172-31-32-58 my-docker-images]$ docker build -t data-test .
Sending build context to Docker daemon 66.56 kB
Sending build context to Docker daemon
Step 0 : FROM busybox:latest
busybox:latest: The image you are pulling has been verified

df7546f9f060: Pull complete
e433a6c5b276: Pull complete
e72ac664f4f0: Pull complete
511136ea3c5a: Already exists
Status: Downloaded newer image for busybox:latest
--> e72ac664f4f0
Step 1 : MAINTAINER Oskar Hane <oh@oskarhane.com>
--> Running in 6e8ea907fe4e
--> d2a4eec526f3
Removing intermediate container 6e8ea907fe4e
Step 2 : RUN mkdir /mysql_data && mkdir /www_data
--> Running in a315fc38b5d
--> f0b08557876a
Removing intermediate container a315fc38b5d
Step 3 : VOLUME mysql_data www_data
--> Running in 9c87f4a1891f
--> b5e344f268b3
Removing intermediate container 9c87f4a1891f
Successfully built b5e344f268b3
[ec2-user@ip-172-31-32-58 my-docker-images]$ docker images
REPOSITORY      TAG          IMAGE ID      CREATED       VIRTUAL SIZE
data-test        latest       b5e344f268b3   9 seconds ago  2.433 MB
oskarhane/hello  latest       26c2a2935d3f   3 weeks ago   192.7 MB
busybox          latest       e72ac664f4f0    6 weeks ago   2.433 MB
[ec2-user@ip-172-31-32-58 my-docker-images]$
```

When everything works as we want, it's time to commit the changes and push it to our GitHub repository:

```
git commit -m "Dockerfile for data volume added."
git push origin data
```

When you have pushed it to the repository, head over to GitHub to verify that your new branch is present there.

The following screenshot shows the GitHub repository:

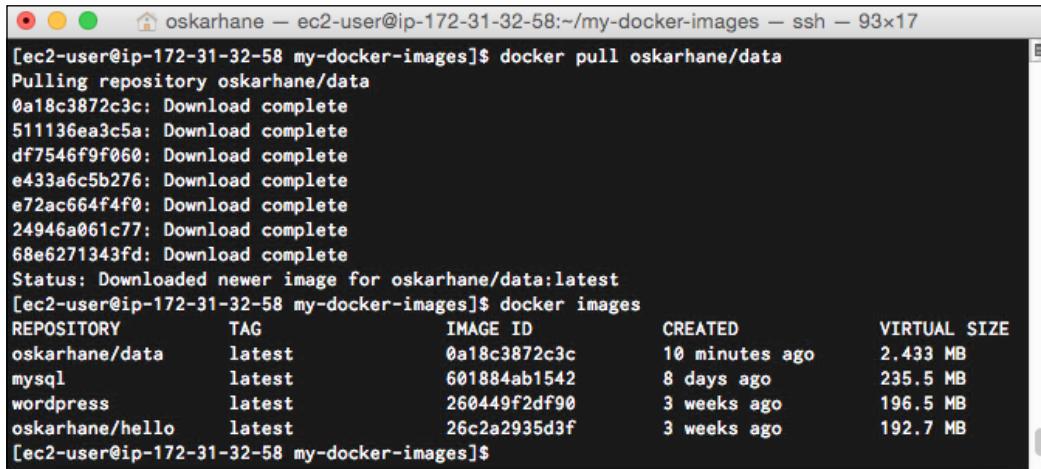
Publishing on the Docker Registry Hub

Now that we have our new branch on GitHub, we can go to the Docker Hub Registry and create a new automated build, named `data`. It will have our GitHub `data` branch as source.

Type	Name	Dockerfile Location	Docker Tag Name
Branch	data	/	latest

Wait for the build to finish, and then try to pull the image with your Docker daemon to verify that it's there and it's working.

The screenshot will be as follows:



A screenshot of a terminal window titled "oskarhane" with the command line "ec2-user@ip-172-31-32-58 my-docker-images". The window shows the output of the "docker pull oskarhane/data" command, which pulls a series of layers (0a18c3872c3c, 511136ea3c5a, df7546f9f060, e433a6c5b276, e72ac664f4f0, 24946a061c77, 68e6271343fd) and ends with a status message: "Status: Downloaded newer image for oskarhane/data:latest". Below this, the "docker images" command is run, displaying a table of images:

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
oskarhane/data	latest	0a18c3872c3c	10 minutes ago	2.433 MB
mysql	latest	601884ab1542	8 days ago	235.5 MB
wordpress	latest	260449f2df90	3 weeks ago	196.5 MB
oskarhane/hello	latest	26c2a2935d3f	3 weeks ago	192.7 MB

Amazing! Check out the size of the image; it's just less than 2.5 MB. This is perfect since we just want to store data in it. A container on top of this image can, of course, be as big as your hard drive allows. This is just to show how big the image is. The image is read-only, remember?

Running a data volume container

Data volume containers are special; they can be stopped and still fulfill their purpose. Personally, I like to see all containers in use when executing `docker ps` command, since I like to delete stopped containers once in a while.

This is totally up to you. If you're okay with keeping the container stopped, you can start it using this command:

```
docker run -d oskarhane/data true
```

The `true` argument is just there to enter a valid command, and the `-d` argument places the container in detached mode, running in the background.

If you want to keep the container running, you need to place a service in the foreground, like this:

```
docker run -d oskarhane/data tail -f /dev/null
```

The output of the preceding command is as follows:

```
[ec2-user@ip-172-31-32-58 case]$ docker run -d oskarhane/data true
140545daed681ab784dc31d0352be4fecf96d72cda19c7aa83e475a26a5b94
[ec2-user@ip-172-31-32-58 case]$
[ec2-user@ip-172-31-32-58 case]$
[ec2-user@ip-172-31-32-58 case]$
[ec2-user@ip-172-31-32-58 case] docker run -d oskarhane/data tail -f /dev/null
5c7e8317f131cb44720b0d0b817bd456493681c8c4a8cdb19ab56091dba7cd
[ec2-user@ip-172-31-32-58 case]$
[ec2-user@ip-172-31-32-58 case]$
[ec2-user@ip-172-31-32-58 case]$
[ec2-user@ip-172-31-32-58 case] docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
5c7e8317f13d        oskarhane/data:latest   "tail -f /dev/null"   11 seconds ago    Up 10 seconds
[ec2-user@ip-172-31-32-58 case]$
[ec2-user@ip-172-31-32-58 case]$
[ec2-user@ip-172-31-32-58 case] docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
5c7e8317f13d        oskarhane/data:latest   "tail -f /dev/null"   16 seconds ago    Up 15 seconds
140545daed68        oskarhane/data:latest   "true"            29 seconds ago    Exited (0) 28 seconds ago
[ec2-user@ip-172-31-32-58 case]$
```

The `tail -f /dev/null` command is a command that never ends, so the container will be running until we stop it. Resource-wise, the `tail` command is pretty harmless.

Passing parameters to containers

We have seen how to give containers parameters or environment variables when starting the official MySQL container:

```
docker run --name mysql-one -e MYSQL_ROOT_PASSWORD=pw -d mysql
```

The `-e MYSQL_ROOT_PASSWORD=pw` command is an example showing how you can do it. It means that the `MYSQL_ROOT_PASSWORD` environment variable inside the container has `pw` as the value.

This is a very convenient way to have configurable containers where you can have a setup script as `ENTRYPOINT` or a foreground script configuring passwords; hosts; test, staging, or production environments; and other settings that the container needs.

Creating a parameterized image

Just to get the hang of this feature, which is very good, let's create a small Docker image that converts a string to uppercase or lowercase, depending on the state of an environment variable.

The Docker image will be based on the latest Debian distribution and will have only an `ENTRYPOINT` command. This is the `Dockerfile`:

```
FROM debian:latest
ADD ./case.sh /root/case.sh
RUN chmod +x /root/case.sh
ENTRYPOINT /root/case.sh
```

This takes the case.sh file from our current directory, adds it to the container, makes it executable, and assigns it as ENTRYPOINT.

The case.sh file may look something like this:

```
#!/bin/bash

if [ -z "$STR" ]; then
    echo "No STR string specified."
    exit 0
fi

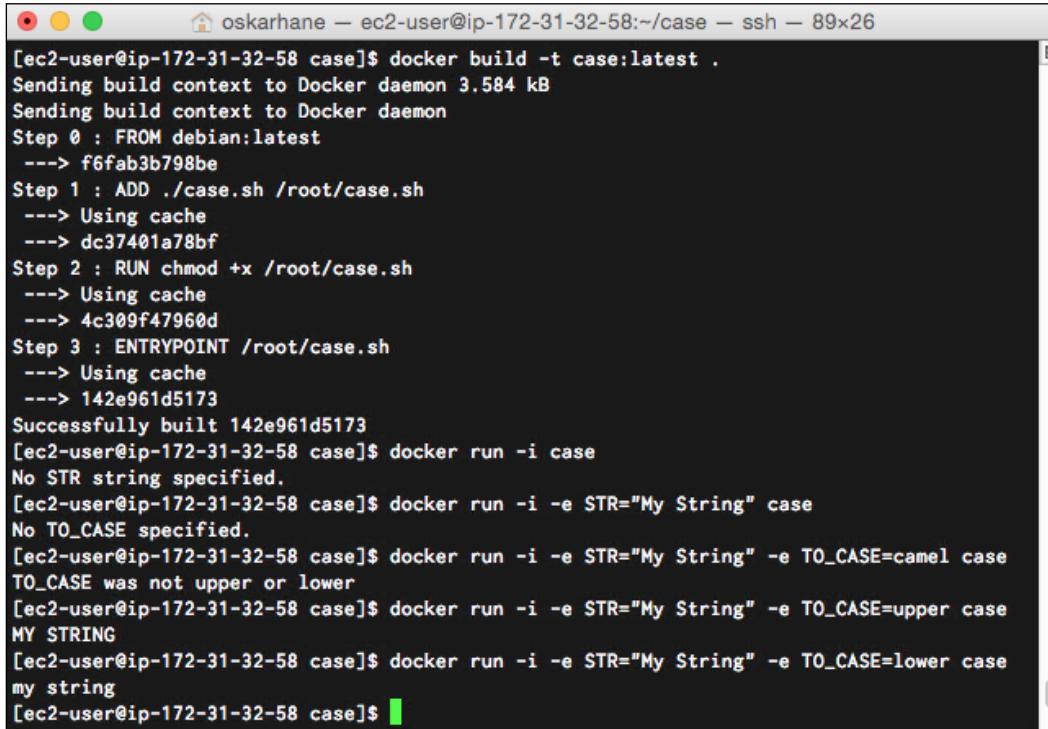
if [ -z "$TO_CASE" ]; then
    echo "No TO_CASE specified."
    exit 0
fi

if [ "$TO_CASE" = "upper" ]; then
    echo "${STR^^*}"
    exit 0
fi
if [ "$TO_CASE" = "lower" ]; then
    echo "${STR,,*}"
    exit 0
fi
echo "TO_CASE was not upper or lower"
```

This file checks whether the \$STR and \$TO_CASE environment variables are set. If the check on whether \$TO_CASE is upper or lower is done and if that fails, an error message saying that we only handle upper and lower is displayed.

If \$TO_STR was set to upper or lower, the content of the environment variable \$STR is transformed to uppercase or lowercase respectively, and then printed to stdout.

Let's try this!



A screenshot of a terminal window titled "oskarhane — ec2-user@ip-172-31-32-58:~/case — ssh — 89x26". The terminal shows the following sequence of commands:

```
[ec2-user@ip-172-31-32-58 case]$ docker build -t case:latest .
Sending build context to Docker daemon 3.584 kB
Sending build context to Docker daemon
Step 0 : FROM debian:latest
--> f6fab3b798be
Step 1 : ADD ./case.sh /root/case.sh
--> Using cache
--> dc37401a78bf
Step 2 : RUN chmod +x /root/case.sh
--> Using cache
--> 4c309f47960d
Step 3 : ENTRYPOINT /root/case.sh
--> Using cache
--> 142e961d5173
Successfully built 142e961d5173
[ec2-user@ip-172-31-32-58 case]$ docker run -i case
No STR string specified.
[ec2-user@ip-172-31-32-58 case]$ docker run -i -e STR="My String" case
No TO_CASE specified.
[ec2-user@ip-172-31-32-58 case]$ docker run -i -e STR="My String" -e TO_CASE=camel case
TO_CASE was not upper or lower
[ec2-user@ip-172-31-32-58 case]$ docker run -i -e STR="My String" -e TO_CASE=upper case
MY STRING
[ec2-user@ip-172-31-32-58 case]$ docker run -i -e STR="My String" -e TO_CASE=lower case
my string
[ec2-user@ip-172-31-32-58 case]$
```

Here are some commands we can try:

```
docker run -i case
docker run -i -e STR="My String" case
docker run -i -e STR="My String" -e TO_CASE=camel case
docker run -i -e STR="My String" -e TO_CASE=upper case
docker run -i -e STR="My String" -e TO_CASE=lower case
```

This seems to be working as expected, at least for this purpose. Now we have created a container that takes parameters and acts upon them.

Summary

In this chapter, you learned that you can keep your data out of your service containers using data volumes. Data volumes can be any one of directories, files from the host's filesystem, or data volume containers.

We explored how we can pass parameters to containers and how to read them from inside `ENTRYPOINT`. Parameters are a great way to configure containers, making it easier to create more generalized Docker images.

We created a data volume container and published it to the Docker Registry Hub, preparing us for the next chapter, where we will connect our three containers to create one loosely coupled unit.

5

Connecting Containers

It's time to connect all our three containers to form a single unit of modularized parts. I'll introduce you to two services, **Docker Compose** and **Crane**, which can be used to automate this. We'll go through the following topics in this chapter:

- Manually connecting containers together
- Exploring the contents of a data volume container
- Connecting containers to a configuration file using Docker Compose
- Connecting containers to a configuration file using Crane

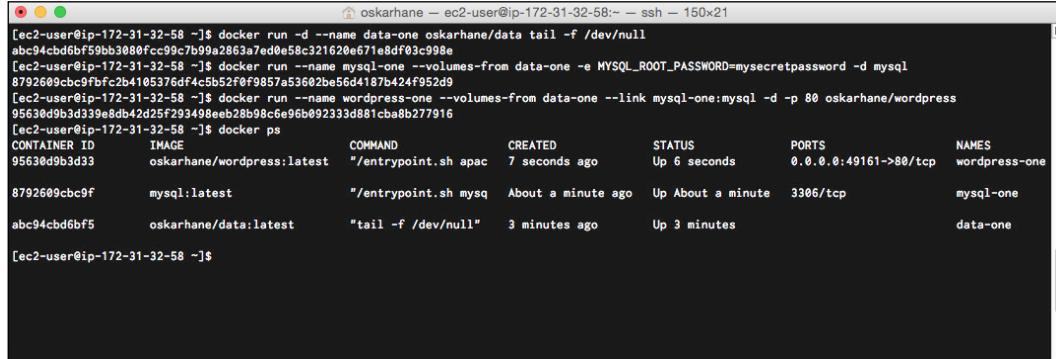
Manually connecting containers

Let's take a look at how to connect our service containers to our data volume container. First, we have to run our data volume container, then run our MySQL container, and lastly run our WordPress container, as shown in the following command:

```
docker run -d --name data-one oskarhane/data tail -f /dev/null
docker run --name mysql-one --volumes-from data-one -e MYSQL_ROOT_
PASSWORD=myscretpassword -d mysql
docker run --name wordpress-one --volumes-from data-one --link mysql-
one:mysql -d -p 80 oskarhane/wordpress
```

Here, we have fired up and named the data volume container `data-one`. The next line fires up the MySQL container, named `mysql-one`, and gives it the data volume container. The last line fires up our WordPress container, named `wordpress-one`, links `mysql-one` as the MySQL link, and gives it the data volume container.

The following output is displayed:



```
oskarhane — ec2-user@ip-172-31-32-58:~ ssh - 160x21
[ec2-user@ip-172-31-32-58 ~]$ docker run -d --name data-one oskarhane/data tail -f /dev/null
abc94cbdbfb59bb3080fc99c7b99a2863a7ed0e58c321620e671e8df03c998e
[ec2-user@ip-172-31-32-58 ~]$ docker run --name mysql-one --volumes-from data-one -e MYSQL_ROOT_PASSWORD=mysecretpassword -d mysql
8792609cbc9fbfc2b4105376d4c5b52f0f9857a53602be56d4187b424f952d9
[ec2-user@ip-172-31-32-58 ~]$ docker run --name wordpress-one --volumes-from data-one --link mysql-one:mysql -d -p 80 oskarhane/wordpress
95630d9b3d339e8db42d25f293498eeb28b98c6e96b092333d881cba8b277916
[ec2-user@ip-172-31-32-58 ~]$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
95630d9b3d33        oskarhane/wordpress:latest   "/entrypoint.sh apac"   7 seconds ago     Up 6 seconds          0.0.0.0:49161->80/tcp   wordpress-one
8792609cbc9f        mysql:latest           "/entrypoint.sh mysq"  About a minute ago Up About a minute    3306/tcp            mysql-one
abc94cbdbfb5        oskarhane/data:latest      "tail -f /dev/null"  3 minutes ago     Up 3 minutes          data-one
[ec2-user@ip-172-31-32-58 ~]$
```

Open your web browser and head over to the container's URL and port in order to verify that all the services are running and the containers are tied together as they should be. You should see the, now familiar, WordPress installation page.

As you may have figured out by now, you can fire up another WordPress container using the same MySQL link and the same data volume container. What do you think will happen?

The new WordPress container will be another instance of the same WordPress site, with the same files and the same database.

When you link containers, Docker will set some environment variables in the target container in order to enable you to get information about the linked source container. In our case, these environment variables will be set when we link the MySQL container, as shown in the following command:

```
MYSQL_NAME=/wordpress-one/mysql-one
MYSQL_PORT=tcp://ip:3306
MYSQL_3306_TCP=tcp://ip:3306
MYSQL_3306_TCP_PROTO=tcp
MYSQL_3306_TCP_PORT=3306
MYSQL_3306_TCP_ADDR=ip
```

Exploring the contents of a data volume container

Is the data being written to the data volume container? Or, is the data stored inside the MySQL and WordPress containers when connected? How can you tell?

One way to determine this is to enter a container via a shell so that you can navigate around its filesystem. Since version 1.3, Docker has the ability to start a new instance of a container's shell. Running the old `docker attach` command just gets you in the current shell instance, which in our case has `tail -f /dev/null` running. If we exit this `tail` command, the container will exit and shut down. Therefore, we need a new shell instance in a running container so that we can invoke any commands we want inside the container without the risk of the container exiting. The following command can be used to do this:

```
docker exec -i -t data-one /bin/sh
```

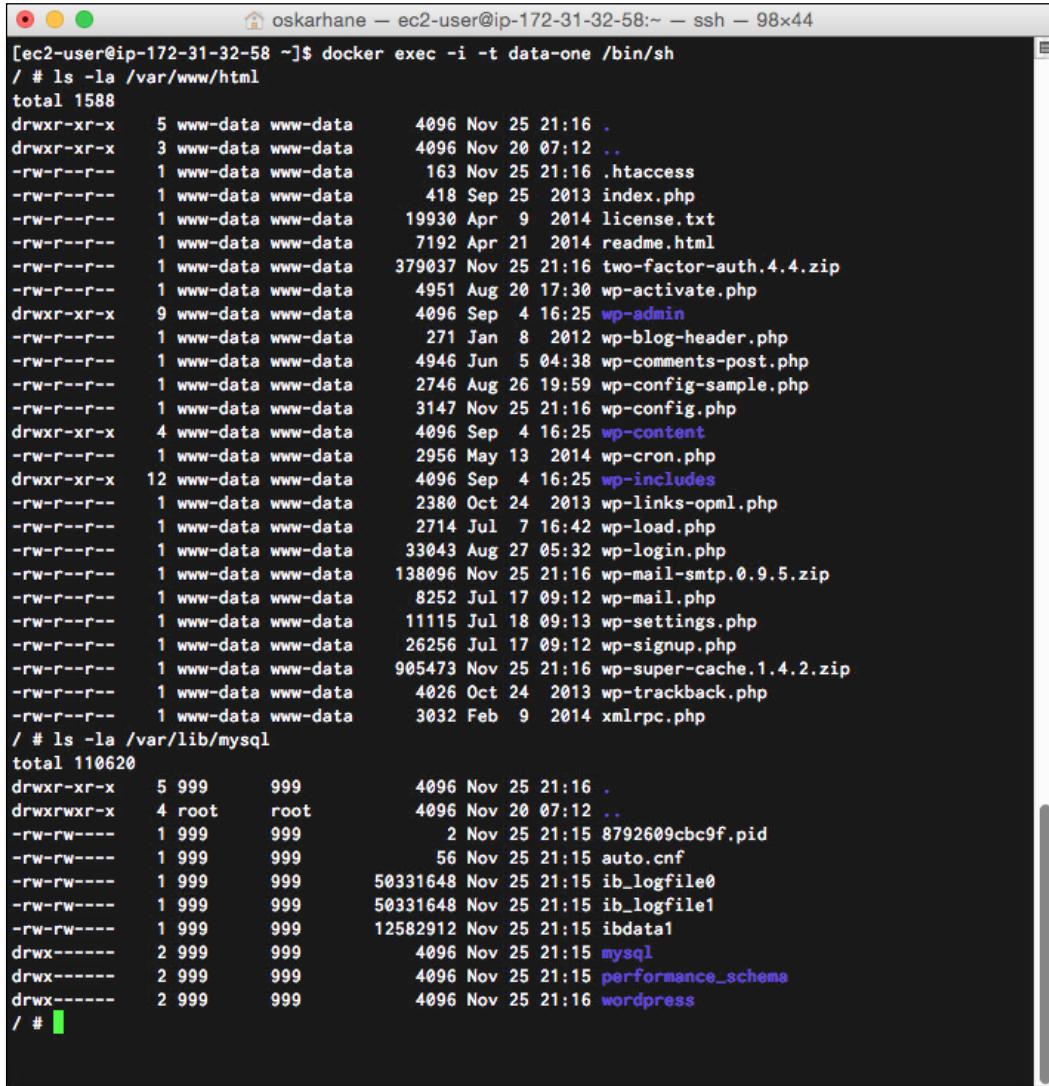
The `-i` and `-t` flags mean that we want to keep the session interactive and allocate a pseudo-TTY. `data-one` is the name of the container, but you can use the container ID if you like. I would choose `/bin/bash` over `/bin/sh`, but the container runs BusyBox and `/bin/bash` isn't available there. For the kinds of tasks that we are about to perform, it doesn't matter which shell we use.

What we want to do is to take a look in the directories we exposed as VOLUMES in this data volume container. The directories are `/var/www/html` and `/var/lib/mysql`.

Let's explore in the following command:

```
ls -la /var/www/html  
ls -la /var/lib/mysql
```

The following output is displayed:



A screenshot of a terminal window titled "oskarhane — ec2-user@ip-172-31-32-58:~ — ssh — 98x44". The terminal displays two commands: "ls -la /var/www/html" and "ls -la /var/lib/mysql". The first command shows a large directory listing for the "/var/www/html" directory, which contains numerous files and sub-directories related to a WordPress installation, such as "index.php", "license.txt", "readme.html", and various PHP files. The second command shows a smaller directory listing for the "/var/lib/mysql" directory, containing MySQL-related files like "ib_logfile0", "ib_logfile1", and "ibdata1". Both lists show files owned by "www-data" and "root" with various timestamps and permissions.

```
[ec2-user@ip-172-31-32-58 ~]$ docker exec -i -t data-one /bin/sh
/ # ls -la /var/www/html
total 1588
drwxr-xr-x    5 www-data www-data    4096 Nov 25 21:16 .
drwxr-xr-x    3 www-data www-data    4096 Nov 20 07:12 ..
-rw-r--r--    1 www-data www-data    163 Nov 25 21:16 .htaccess
-rw-r--r--    1 www-data www-data   418 Sep 25 2013 index.php
-rw-r--r--    1 www-data www-data 19930 Apr  9 2014 license.txt
-rw-r--r--    1 www-data www-data  7192 Apr 21 2014 readme.html
-rw-r--r--    1 www-data www-data 379037 Nov 25 21:16 two-factor-auth.4.4.zip
-rw-r--r--    1 www-data www-data  4951 Aug 20 17:30 wp-activate.php
drwxr-xr-x    9 www-data www-data  4096 Sep  4 16:25 wp-admin
-rw-r--r--    1 www-data www-data   271 Jan  8 2012 wp-blog-header.php
-rw-r--r--    1 www-data www-data  4946 Jun  5 04:38 wp-comments-post.php
-rw-r--r--    1 www-data www-data  2746 Aug 26 19:59 wp-config-sample.php
-rw-r--r--    1 www-data www-data  3147 Nov 25 21:16 wp-config.php
drwxr-xr-x    4 www-data www-data  4096 Sep  4 16:25 wp-content
-rw-r--r--    1 www-data www-data  2956 May 13 2014 wp-cron.php
drwxr-xr-x   12 www-data www-data  4096 Sep  4 16:25 wp-includes
-rw-r--r--    1 www-data www-data  2380 Oct 24 2013 wp-links-opml.php
-rw-r--r--    1 www-data www-data  2714 Jul  7 16:42 wp-load.php
-rw-r--r--    1 www-data www-data 33043 Aug 27 05:32 wp-login.php
-rw-r--r--    1 www-data www-data 138096 Nov 25 21:16 wp-mail-smtp.0.9.5.zip
-rw-r--r--    1 www-data www-data   8252 Jul 17 09:12 wp-mail.php
-rw-r--r--    1 www-data www-data  11115 Jul 18 09:13 wp-settings.php
-rw-r--r--    1 www-data www-data  26256 Jul 17 09:12 wp-signup.php
-rw-r--r--    1 www-data www-data 905473 Nov 25 21:16 wp-super-cache.1.4.2.zip
-rw-r--r--    1 www-data www-data  4026 Oct 24 2013 wp-trackback.php
-rw-r--r--    1 www-data www-data  3032 Feb  9 2014 xmlrpc.php
/ # ls -la /var/lib/mysql
total 110620
drwxr-xr-x    5 999     999    4096 Nov 25 21:16 .
drwxrwxr-x    4 root     root    4096 Nov 20 07:12 ..
-rw-rw----    1 999     999     2 Nov 25 21:15 8792609cbc9f.pid
-rw-rw----    1 999     999    56 Nov 25 21:15 auto.cnf
-rw-rw----    1 999     999  50331648 Nov 25 21:15 ib_logfile0
-rw-rw----    1 999     999  50331648 Nov 25 21:15 ib_logfile1
-rw-rw----    1 999     999 12582912 Nov 25 21:15 ibdata1
drwx-----   2 999     999    4096 Nov 25 21:15 mysql
drwx-----   2 999     999    4096 Nov 25 21:15 performance_schema
drwx-----   2 999     999    4096 Nov 25 21:16 wordpress
/ #
```

We see files on both those directories, which indicates that the two other containers are writing to this one. It separates the services with the data. If you want further proof, launch `vi` in the shell, edit a file, and reload the site in your browser.

This worked out really smooth and easy, didn't it? The containers interact with each other and all we have to do is to link them together with just one command.

Connecting containers using Docker Compose

Docker Compose was previously called Fig, but Docker acquired Fig and the name was changed. This is how Docker describes Docker Compose:

"Compose is a tool for defining and running complex applications with Docker. With Compose, you define a multi-container application in a single file, then spin your application up in a single command which does everything that needs to be done to get it running."

Docker Compose basically gives us a way to define settings in a configuration file, so we don't have to remember all the names for all the containers when linking them together, the ports to expose, the data volume container to use, and so on.

Installing Docker Compose

Docker Compose has regular releases on GitHub, and at the time of writing this book, the latest release is 1.0.1.

We will install Docker Compose with the Python package manager, pip. Our EC2 instance does not come with pip installed, so we have to start with the installation, as shown here:

```
sudo su
wget https://bootstrap.pypa.io/get-pip.py && python ./get-pip.py
```

The following output is displayed:

```
[root@ip-172-31-32-58 ec2-user]# wget https://bootstrap.pypa.io/get-pip.py && python ./get-pip.py
converted 'https://bootstrap.pypa.io/get-pip.py' (ANSI_X3.4-1968) -> 'https://bootstrap.pypa.io/get-pip.py' (UTF-8)
--2014-12-02 21:09:34-- https://bootstrap.pypa.io/get-pip.py
Resolving bootstrap.pypa.io (bootstrap.pypa.io)... 199.27.79.175
Connecting to bootstrap.pypa.io (bootstrap.pypa.io)|199.27.79.175|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1340903 (1.3M) [text/x-python]
Saving to: 'get-pip.py.2'

get-pip.py.2          100%[=====] 1.28M 3.49MB/s in 0.4s

2014-12-02 21:09:34 (3.49 MB/s) - 'get-pip.py.2' saved [1340903/1340903]

Requirement already up-to-date: pip in /usr/lib/python2.6/site-packages
Cleaning up...
[root@ip-172-31-32-58 ec2-user]# pip -V
pip 1.5.6 from /usr/lib/python2.6/site-packages (python 2.6)
[root@ip-172-31-32-58 ec2-user]#
```

After pip is installed, you can go ahead and install Docker Compose:

```
sudo pip install -U docker-compose
```

Now, you'll see Docker Compose installed along with all of its dependencies. Invoke `docker-compose --version` to verify that it works as expected.

Basic Docker Compose commands

The following are the basic Docker Compose commands that you should be familiar with:

- `build`: This is used to build or rebuild services
- `kill`: This forces the service containers to stop
- `logs`: This views the output from the services
- `port`: This is used to print the public port for a port binding
- `ps`: This is used to list containers
- `pull`: This is used to pull service images
- `rm`: This is used to remove stopped service containers
- `run`: This is used to run a one-off command on a service
- `scale`: This sets the number of containers to be run for a service
- `start`: This is used to start existing containers for a service
- `stop`: This stops running containers without removing them
- `up`: This builds, recreates, starts, and attaches to containers for a service; linked containers will be started, unless they are already running

As you can see, the commands are very similar to the Docker client commands and most of them do the exact same thing by forwarding the commands to the Docker daemon. We will go through some of them a little more in detail.

Service

When the word `service` is used with Docker Compose, it refers to a named container in a `docker-compose.yml` configuration file.

Using the run command

We are used to starting containers with the `run` command for the Docker client. With docker-compose, the `run` command is very different. When you `run` a command with docker-compose, it's a one-off command on a service. This means that if we name a container configuration `Ubuntu` and invoke `docker-compose run ubuntu /bin/bash echo hello`, the container will start and execute `/bin/bash echo hello` and then shut down. The difference with this and running the command directly with Docker is that all the linked containers and `VOLUME` containers will be started and connected when you use docker-compose.

Using the scale command

The `scale` command is very interesting. When we invoke `docker-compose scale web=3`, we actually start three containers of the service that we named `web`.

Setting up our PaaS with Docker Compose

Every Docker Compose instance lives in its own directory and has a configuration file named `docker-compose.yml` inside it:

```
mkdir docker-compose-wp && cd $_
touch docker-compose.yml
```

This is how the contents of our `docker-compose.yml` file will look:

```
wp:
  image: oskarhane/wordpress
  links:
    - mysql:mysql
  ports:
    - "80"
  volumes_from:
    - paasdata
mysql:
  image: mysql
  volumes_from:
    - paasdata
  environment:
    - MYSQL_ROOT_PASSWORD=myrootpass
paasdata:
  image: oskarhane/data
  command: tail -f /dev/null
```

You can see that we have defined three services here, namely `wp`, `mysql`, and `paasdata`.

Lets try these services and the following output is displayed:



```
oskarhane — ec2-user@ip-172-31-32-58:~/docker-compose-wp — ssh — 92x12
[ec2-user@ip-172-31-32-58 docker-compose-wp]$ docker-compose up -d
Recreating dockercomposewp_paasdata_1...
Recreating dockercomposewp_mysql_1...
Recreating dockercomposewp_wp_1...
[ec2-user@ip-172-31-32-58 docker-compose-wp]$ docker-compose ps
          Name        Command     State      Ports
-----  -----
dockercomposewp_mysql_1   /entrypoint.sh mysqld --da ...   Up      3306/tcp
dockercomposewp_paasdata_1 tail -f /dev/null           Up
dockercomposewp_wp_1      /entrypoint.sh apache2 -DF ...   Up      0.0.0.0:49155->80/tcp
[ec2-user@ip-172-31-32-58 docker-compose-wp]$
[ec2-user@ip-172-31-32-58 docker-compose-wp]$
```

Invoke `docker-compose up -d` to run `docker-compose` and the containers in daemon mode.

That's how easy it is. Open your web browser and head to your Docker host and the port stated in the table (in my case, port **49155**); you should see the very familiar WordPress installation page.

Connecting containers using Crane

Crane is much like Docker Compose, but it has more configuration possibilities. This is how its creator describes Crane:

"Crane is a tool to orchestrate Docker containers. It works by reading in some configuration (JSON or YAML) which describes how to obtain images and how to run containers. This simplifies setting up a development environment a lot as you don't have to bring up every container manually, remembering all the arguments you need to pass. By storing the configuration next to the data and the app(s) in a repository, you can easily share the whole environment."

This paragraph can be about Docker Compose as well, as you can see.

Installing Crane

Crane is easy to install but not easy to keep updated. The same command is used to install as well as update, so we have to invoke this once in a while in order to have the latest version.

Invoke the following command on a single line to install Crane:

```
bash -c "`curl -sL https://raw.githubusercontent.com/michaelsauter/crane/master/download.sh`" && sudo mv crane /usr/local/bin/crane
```

Crane is now installed in /usr/local/bin.

Usage

I won't go through all the commands here since they're similar to Docker Compose's commands, but I'll comment on a few here:

- `lift`: This command, like Docker Compose's `up` command, builds and runs containers from your configuration file
- `graph`: This prints your containers' relations from the configuration file
- `logs`: This maps to the Dockers Compose's command, but here you can get the logs for a whole group
- `status`: This also maps to the Dockers Compose's command but lets you get the logs for a group

Configuration

This is where Crane really leaves Docker Compose behind. You have many more configuration options for Crane apps. The configuration file must be named `crane.json` or `crane.yaml`. For every container, this is what you can configure:

- `image` (string, required): This is the name of the image to build/pull
- `dockerfile` (string, optional): This gives the relative path to the Dockerfile
- `run` (object, optional): These parameters are mapped to Docker's `run` and `create` commands:
 - `add-host` (array): This adds custom host-to-IP mappings
 - `cpuset` (integer)
 - `cpu-shares` (integer)
 - `detach` (boolean) `sudo docker attach <container name>` will work as normal

- device (array): This adds host devices
- dns (array)
- entrypoint (string)
- env (array)
- expose (array): This denotes the ports to be exposed to linked containers
- hostname (string)
- interactive (boolean)
- link (array): This links containers
- memory (string)
- privileged (boolean)
- publish (array): This maps network ports to the container
- publish-all (boolean)
- restart (string) Restart policy
- rm (boolean)
- tty (boolean)
- volume (array): In contrast to plain Docker, the host path can be relative
- volumes-from (array): This is used to mount volumes from other containers
- workdir (string)
- cmd (array/string): This command is used to append to `docker run` (overwriting CMD)
- `rm` (object, optional) : These parameters are mapped to Docker's `rm` command:
 - volumes (boolean)
- `start` (object, optional): These parameters are mapped to Docker's `start` command:
 - attach (boolean)
 - interactive (boolean)

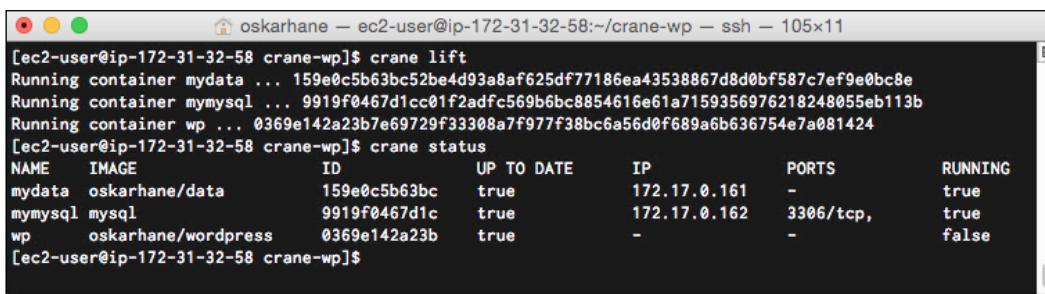
Set up the same configuration that you did in Docker Compose; it will look something like the following code. As you might understand, you can write this in the JSON format as well, but for the comparison to Docker Compose's version to be as easy as possible, I'll keep it in the yaml format:

```
containers:
  wp:
    image: oskarhane/wordpress
    run:
      volumes-from: ["mydata"]
      link:
        - mymysql:mysql
      publish: ["80"]
      detach: true
  mymysql:
    image: mysql
    run:
      volumes-from: ["mydata"]
      detach: true
      env: ["MYSQL_ROOT_PASSWORD=rootpass"]
  mydata:
    image: oskarhane/data
    run:
      detach: true
      cmd: "tail -f /dev/null"
```

Here, we specify three containers, where the data container is added as a data volume container to the others and the MySQL container is linked to the WordPress container.

Save this file as `crane.yaml` and type `crane lift` to run your app.

The following output is displayed:



The screenshot shows a terminal window with the following content:

```
oskarhane - ec2-user@ip-172-31-32-58:~/crane-wp - ssh - 105x11
[ec2-user@ip-172-31-32-58 crane-wp]$ crane lift
Running container mydata ...
Running container mymysql ...
Running container wp ...
[ec2-user@ip-172-31-32-58 crane-wp]$ crane status
NAME     IMAGE          ID           UP TO DATE    IP          PORTS        RUNNING
mydata   oskarhane/data 159e0c5b63bc  true        172.17.0.161  -
mymysql  mysql          9919f0467d1c  true        172.17.0.162  3306/tcp,  true
wp       oskarhane/wordpress 0369e142a23b  true        -           -           false
```

To see the containers' current statuses, we can type `crane status`. Take a look at the last column in our `wp` container. It says it's not running. Type `crane logs wp` and see what it says in following command:

```
wp * WordPress not found in /var/www/html - copying now...
wp * Complete! WordPress has been successfully copied to /var/www/html
wp |
wp | Warning: mysqli::mysqli(): (HY000/2002): Connection refused in - on
line 5
wp * MySQL Connection Error: (2002) Connection refused
```

It seems that our WordPress container starts faster than our MySQL container, so the WordPress container can't find it when it starts.

This can happen in Docker Compose as well because there's no check if `--link:ed` containers are up, at least not at the time when this is being written.

This cannot be solved in Docker Compose; we have to rely on pure luck that the MySQL container will get ready before the WordPress container tries to use the linked MySQL container.

With Crane, you can group containers inside the configuration file in different groups and then `run` commands on that group instead of the whole configuration.

This is very easy; we just add these lines at the end of our `crane.yaml` file:

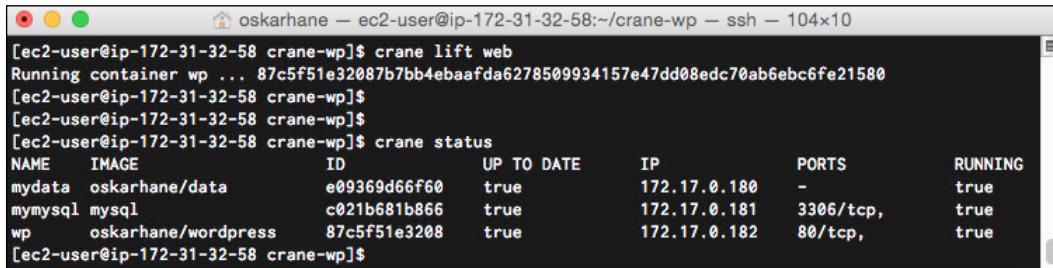
```
groups:
  default: ['mydata', 'mymysql', 'wp']
  data_db: ['mydata', 'mymysql']
  web: ['wp']
```

Here, we have separated the WordPress container from the other two containers we have so that we can `run` commands on them separately.

Let's start our `data_db` group first by invoking the `crane lift data_db --recreate` command. I added the flag `--recreate` and to make sure that we're creating new containers and not reusing the old ones. Run `crane status data_db` to make sure they're running.

Now that we know that the MySQL container is running, we can start the WordPress container by invoking the `crane lift web --recreate` command.

The following output is displayed:



```
oskarhane — ec2-user@ip-172-31-32-58:~/crane-wp — ssh — 104x10
[ec2-user@ip-172-31-32-58 crane-wp]$ crane lift web
Running container wp ... 87c5f51e32087b7bb4ebaafda6278509934157e47dd08edc70ab6ebc6fe21580
[ec2-user@ip-172-31-32-58 crane-wp]$
[ec2-user@ip-172-31-32-58 crane-wp]$
[ec2-user@ip-172-31-32-58 crane-wp]$ crane status
NAME      IMAGE          ID           UP TO DATE    IP          PORTS        RUNNING
mydata    oskarhane/data e09369d66f60  true        172.17.0.180 -            true
mymysql   mysql          c021b681b866  true        172.17.0.181 3306/tcp,   true
wp        oskarhane/wordpress 87c5f51e3208  true        172.17.0.182 80/tcp,    true
[ec2-user@ip-172-31-32-58 crane-wp]$
```

Summary

Now, we can connect containers in different ways to keep different services separate on different containers. We learned how to do this manually, which can be quite hard when you have lots of dependencies between containers.

We had a brief look at two orchestration tools: Docker Compose and Crane. Crane is an independent and more advanced tool for the administrators who want more control over containers. The ability to group containers in Crane makes it more reliable when there can be timing issues in dependencies.

In the next chapter, we will run two instances of our app using Crane to see what problems and possibilities crop up when we want to make both our blogs publicly accessible on the regular HTTP port (80).

6

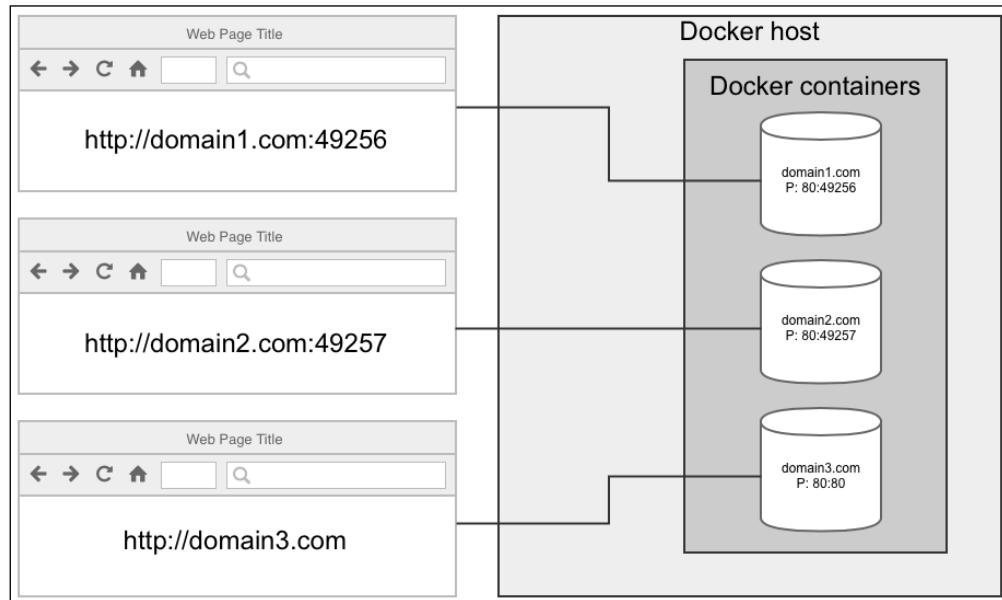
Reverse Proxy Requests

One big problem in having many containers with public ports on the same server is that they can't all listen to the standard ports for their kinds of services. If we have a MySQL backend service and have 10 MySQL containers running, only one of them can listen to the MySQL standard port 3306. For those who expose a web server, the standard port 80 can only be used by one of their WordPress containers. In this chapter, we'll cover the following topics:

- Explaining the problem
- Coming up with a solution to the problem
- Implementing the solution with Nginx and HAProxy
- Automating the process of mapping domains

Explaining the problem

The problem in having many containers with the same services on the same host is that there are standard ports used by user applications. Using a web browser and entering the IP to a Docker host running a WordPress container will ask for resources on port 80 by default. You can't expect your users to remember a nonstandard port in order to enter your website.



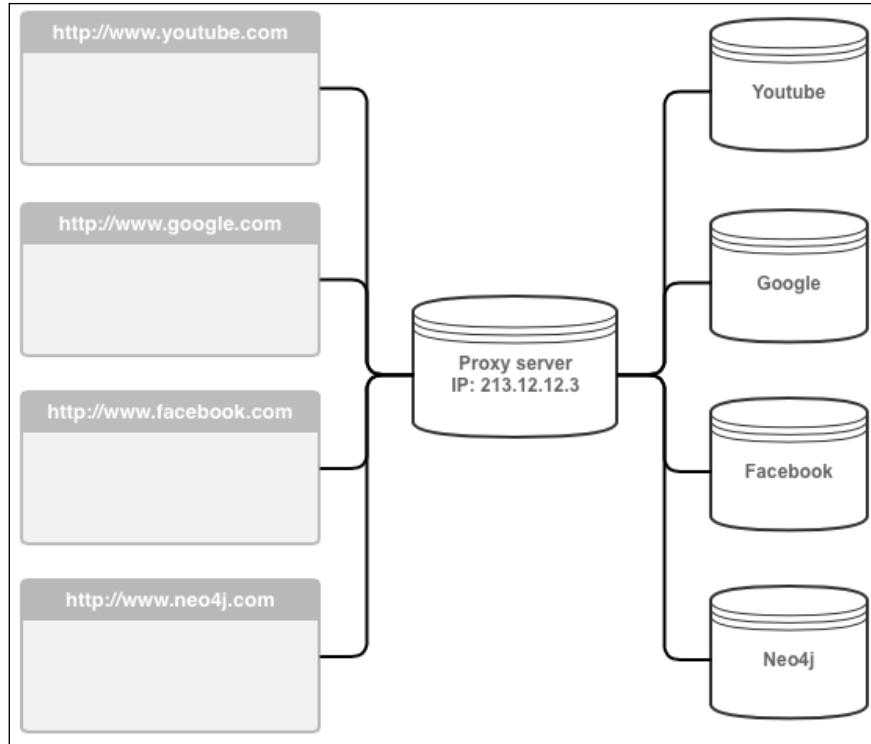
The only way to reach each of the three containers is to manually enter the containers' exposed port number.

Finding a solution

Before we head to the solution, let me explain what a regular proxy server is, in case you're not familiar with it.

A proxy server is a server that connects to services on your behalf and forwards all the results to you. After you've set up to route all your traffic through the proxy server, you – as a user – won't notice it's there. Everything will work as usual.

However, service owners only see that a certain machine (the proxy server) is connected to them. If another user uses the same proxy server and the same service as you do, the service owner can't tell the difference and will perceive you as one single user.



Different users connecting through a proxy server appear as one user.

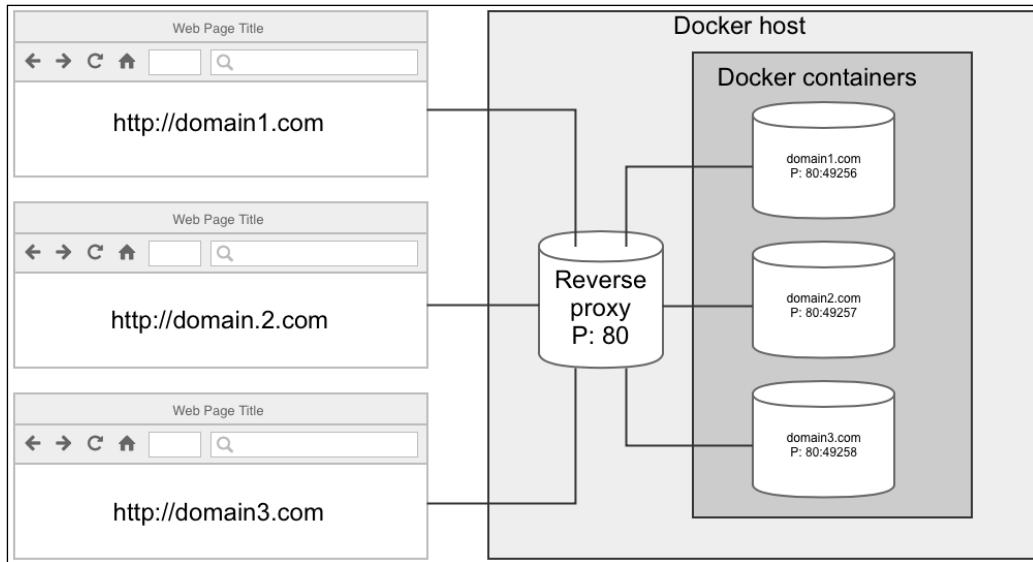
As you can see in the preceding diagram, the service owners just see that someone with an IP of **213.12.12.3** has connected to them.

So, what if we use this on the Docker host? What if we put something in front of all the containers? Depending on which domain name is being requested, this thing will forward the request to the right container and port and then just forward the request's response to the requesting user.

There are things especially made to solve this kind of problem. They're called **reverse proxies** (reverse because the proxy is at the other end, making the user only see one IP and forwarding the request).

Reverse Proxy Requests

If we install and configure a reverse proxy on our Docker host server, then this is how the diagram will look:



A reverse proxy lets all Docker containers appear as one.

The reverse proxy listens to port 80 – the standard web port – and when a request for `domain1.com` comes in, the proxy looks at its configuration to see whether there is a specified forwarding endpoint for this domain. If there is, the reverse proxy forwards the request to the right Docker container, waits for its response, and forwards the container's response to the requesting users when it comes.

This is the solution we're after. The only question now is which reverse proxy we are going to use. There are quite a bunch of them out there; some reverse proxies have more specific purposes, such as load balancing, and some are services that do a lot of other stuff and have this feature as well, such as a web server.

Implementing the solution

You will always have preferences when selecting a tool to solve a problem. Sometimes, you select a tool because you're comfortable using it and it's good enough; sometimes, you select it because it has great performance or because you just want to try something new.

That's why we will go through this problem and solve it with two different tools. The end result will be the same, but the tools have a slightly different setup.

Before we start implementing the solutions, we use Crane to start an instance of our three-container application and verify that it's working by connecting it to the site. Have Docker decide the public port for you, so it's 491xx. Remember this port since we will use it when implementing the solutions.

We need to point out the domain names we want to use to our Docker host's IP address. We can do this either by setting the domain names A-record to our server's IP address or by adding a line in our local /etc/hosts file, which directs requests to the domain names to our server's IP address.

I'll go with the latter and enter this in my Mac's /etc/hosts file:

```
54.148.253.187 domain1.com  
54.148.253.187 domain2.com  
54.148.253.187 domain3.com
```



Make sure you replace the above IP address with your server's IP address.



Implementation with HAProxy

HAProxy (<http://www.haproxy.org>) is a load balancer, which has the role of forwarding traffic to different services behind it.

This is how HAProxy describe themselves:

"HAProxy is a free, very fast and reliable solution offering high availability, load balancing, and proxying for TCP and HTTP-based applications. It is particularly suited for very high traffic web sites and powers quite a number of the world's most visited ones. Over the years it has become the de-facto standard open source load balancer, is now shipped with most mainstream Linux distributions, and is often deployed by default in cloud platforms."

This sounds like something that fits our needs.

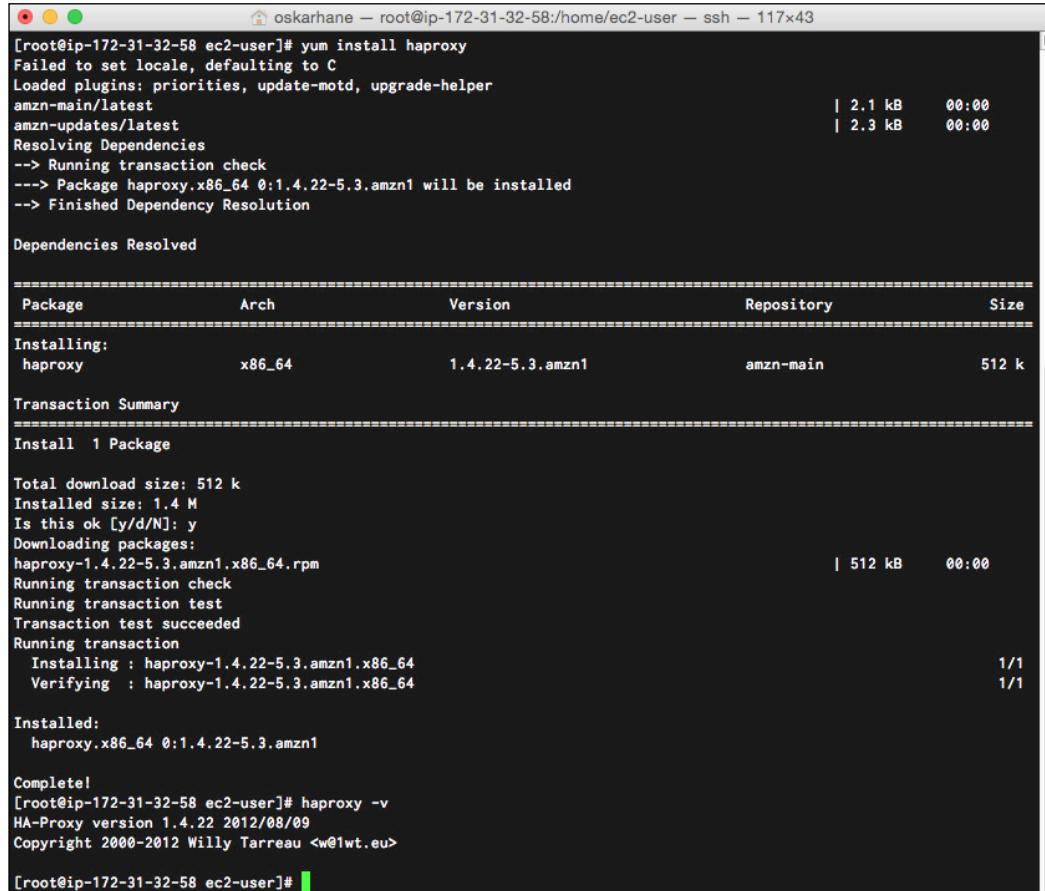
Installing HAProxy

As noted in the quote, many systems are installed already and shipped with it. If you can't find it, it should be available in your package manager if you use Ubuntu or Debian (`apt-get install haproxy`) or in some other distro with a package manager.

On my Amazon EC2 instance that runs Amazon Linux, HAProxy can be installed using `yum install haproxy`.

Reverse Proxy Requests

The following output will be obtained as follows:



A screenshot of a terminal window titled "oskarhane - root@ip-172-31-32-58:/home/ec2-user - ssh - 117x43". The terminal shows the command "yum install haproxy" being run. The output indicates that the package "haproxy.x86_64 0:1.4.22-5.3.amzn1" is being installed from the "amzn-main" repository. The transaction summary shows one package being installed, totaling 512 k. The process includes downloading the package, running a transaction check, and performing a transaction test. Finally, it installs the package and verifies its integrity. The HAProxy version is confirmed as 1.4.22. The terminal ends with a prompt "[root@ip-172-31-32-58 ec2-user]#".

It's not the most recent version, but that's OK for the things we are about to do.

Configuring HAProxy

We'll write an HAProxy configuration in the file `/etc/haproxy/docker.cfg` so that we don't have to remove everything in the default configuration file, as it may be good for reference in the future.

HAProxy divides its configuration into four parts: global, defaults, frontend, and backend. Don't confuse frontend and backend with frontend and backend development. Here, frontend means the server part that's facing the Internet, and backend is the server part that's behind HAProxy, which in our case are the Docker containers.

Open the configuration file and start by typing in the generic stuff, as shown here:

```
global
    daemon
    maxconn 4096
    pidfile /var/run/haproxy.pid
defaults
    mode http
    timeout connect 5000ms
    timeout client 50000ms
    timeout server 50000ms
```

Now, we enter the port to listen on and the backend configurations to use for which domain:

```
frontend http-in
    bind *:80
    acl is_site1 hdr_end(host) -i domain1.com
    use_backend site1 if is_site1
```

We define that regular incoming HTTP traffic on port 80 should be captured. The `acl` here means access control list and is a flexible solution to take decisions based on content extracted from the requests. The `hdr_end(host) -i domain1.com` function call means that the end of the header host is case-insensitive, matched against the string `domain1.com`. The result (Boolean) of this match is saved in the `is_site1` variable.

Note that this means that all the subdomains for `domain1.com` will be matched with this setup. If you just want to match `www.domain1.com`, you can use `hdr(host) -i www.domain1.com` instead.

Now that we have the match result in the `is_site1` variable, we can send the request to a backend configuration, named `site1`.

We append this to our configuration file:

```
backend site1
    balance roundrobin
    option httpclose
    option forwardfor
    server s1 127.0.0.1:49187 maxconn 450
```

We define our backend name as `site1`, set a few options, and add the server and the port to our WordPress container.



Make sure you enter your WordPress container's exposed port instead of 49187 in the preceding code.



It's time to try this configuration. Save the configuration file and test it in a shell with this command:

```
haproxy -f /etc/haproxy/docker.cfg -c
```

The output should say Configuration file is valid.

Make sure you don't have something already listening to port 80 on your machine. You can use something such as netstat -a to verify that 80 or HTTP isn't listed. If they are, find the app that's listening and shut it down.

Start HAProxy with this command:

```
haproxy -f /etc/haproxy/docker.cfg -D
```

The -D option means that we want to run it as a daemon in the background. You shouldn't see any output when you invoke this command.

Let's check whether HAProxy is running by invoking ps aux | grep haproxy. You should see it listed there. Finally, let's verify that it is listening to port 80 by invoking netstat -a | grep http. Now, you should have something in that list.

The output obtained is displayed here:

```
oskarhane ~ root@ip-172-31-32-58:/home/ec2-user/crane-wp ~ ssh ~ 103x9

[root@ip-172-31-32-58 crane-wp]# haproxy -f /etc/haproxy/docker.cfg -D
[root@ip-172-31-32-58 crane-wp]# ps aux | grep haproxy
root    21322  0.0  0.0 16852  956 ?        Ss   20:57  0:00 haproxy -f /etc/haproxy/docker.cfg -D
root    21324  0.0  0.0 110256  640 pts/0   S+  20:57  0:00 grep haproxy
[root@ip-172-31-32-58 crane-wp]# netstat -a | grep http
tcp      0      0 *:http          *:*                  LISTEN
[root@ip-172-31-32-58 crane-wp]#
```

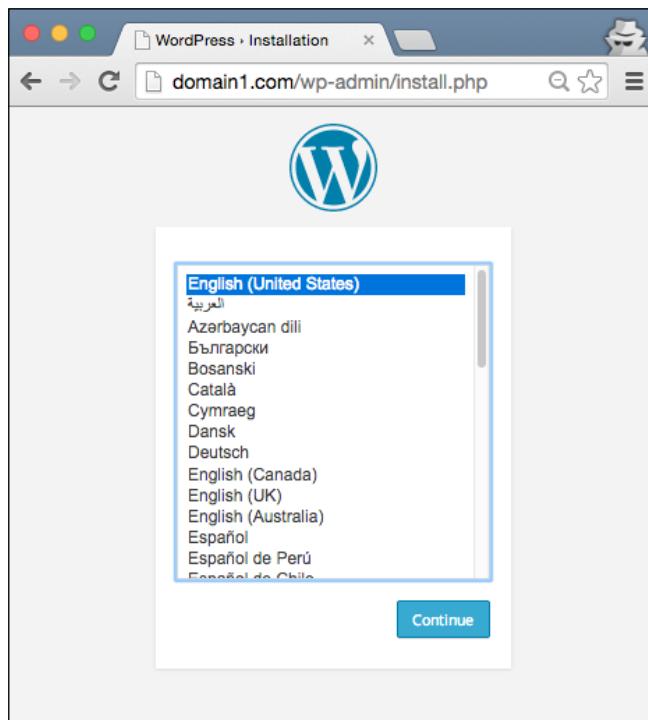
It all looks good!

Just to recap what we have done here: we set up a service that listens for incoming requests on port 80 on our server. When a request on this port comes in, a check on the request header's host is performed to see whether it matches domain1.com. If we have a match, the request is forwarded to the IP address 127.0.0.1 and to the port 49187. The response from this IP and port are sent back to the requester.

Now to the moment of truth. Open your web browser and enter the URL `domain1.com`.

Make sure that you have entries for `domain1.com` in your host's file, pointing it to your server.

After you perform the preceding instructions, you will see the following website screen:



You can see that in the location bar, no port is specified. Wonderful!

Adding more domains to HAProxy

We did not go through all this just to serve a single web application on port 80, which can be done without a reverse proxy. Start another WordPress application with Crane by copying the old configuration to a new directory and change the service's names, as shown here:

```
cd..  
cp -r crane-wp crane-wp2  
cd crane-wp2
```

```
sed -i "s/wp/wp2/g" crane.yaml
sed -i "s/mydata/mydata2/g" crane.yaml
sed -i "s/mymysql/mymysql2/g" crane.yaml
crane lift data_db
crane lift wp2

#check out port for new container named wp2
docker ps
```

Open the HAProxy configuration file again and add two lines in the frontend:

```
acl is_site2 hdr_end(host) -i domain2.com
use_backend site2 if is_site2
```

After that, add a new backend configuration named site2:

```
backend site2
    balance roundrobin
    option httpclose
    option forwardfor
    server s2 127.0.0.1:49188 maxconn 450
```

Make sure that you replace the port with the one you got. Restart HAProxy and do the checks we did the last time we started it.

To restart HAProxy, run `/etc/init.d/haproxy restart`.

 HAProxy can reload a new configuration without dropping active sessions with this command:
`haproxy -f /etc/haproxy/docker.cfg -p /var/run/haproxy.pid -sf $(cat /var/run/haproxy.pid)`

Open your browser and go to `domain1.com` in order to make sure that the old one is working. If it does, go to `domain2.com`. You should see another WordPress installation site. Just to be sure that it's not the same, go ahead and install one of them. Or, go to `domain3.com` and see what happens when a domain points to the server without having it match in HAProxy.

Implementation with Nginx

Now, we are going to do the same thing as we did with HAProxy, but we will use the excellent web server Nginx (<http://nginx.org/en/>) as our reverse proxy instead. Nginx is a full featured and really fast web server that leaves a small footprint in the memory.

This is how Nginx is described:

"nginx [engine x] is an HTTP and reverse proxy server, as well as a mail proxy server, written by Igor Sysoev. For a long time, it has been running on many heavily loaded Russian sites including Yandex, Mail.Ru, VK, and Rambler. According to Netcraft, nginx served or proxied 20.41% busiest sites in November 2014. Here are some of the success stories: Netflix, Wordpress.com, FastMail.FM."

This also sounds like what we need, just like it did with HAProxy.

Installing Nginx

Nginx is available in all Linux package managers, such as aptitude/apt, yum, and others, so an install can be simply done with `apt-get install nginx` or `yum install nginx`. Since it's open source, you can, of course, install it from the source as well.

Configuring Nginx

We are going to add the configuration to a file named `/etc/nginx/conf.d/wp1.conf`.

Create and open this file in your favorite text editor:

```
server {  
    listen 80;  
    server_name domain1.com;  
    charset UTF-8;  
  
    if ($host !~ ^(domain1.com)$ ) {  
        return 444;  
    }  
}
```

This block, as you can see, makes the server listen to port 80 and to match the domain `domain1.com` for this configuration to apply. It's always good to specify the server charset so that the website text does not get the wrong encoding during the forwarding process; so, we add that line as well. To just listen to `domain1.com` and nothing else (Nginx uses the first configuration found as a default configuration if there's no match in the server name part), we return the HTTP status code 444 (no response) on the other requests that get in there.

What are we going to do with the requests on port 80 for `domain1.com` then?

Add this inside the server's scope (curly brackets):

```
location / {  
    proxy_pass http://wp1;  
    proxy_set_header X-Real-IP $remote_addr;  
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;  
    proxy_set_header X-NginX-Proxy true;  
    proxy_set_header Host $host;  
    proxy_set_header X-Forwarded-Proto $scheme;  
    proxy_redirect off;  
}
```

The location block will match all the requests since it matches /. We will get back to the proxy_pass part in a while. Other than this, you'll see that we set a lot of headers, most of them telling our Docker container the requesters' real IP address and so on.

Back to the proxy_pass part. This is the part that actually forwards the request, to something named wp1. This is called an upstream, which we have to define.

Add this outside the server's scope:

```
upstream wp1 {  
    server 127.0.0.1:49187;  
}
```

The complete configuration file named /etc/nginx/conf.d/wp1.conf should look like this now:

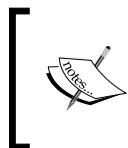
```
upstream wp1 {  
    server 127.0.0.1:49187;  
}  
  
server {  
    listen 80;  
    server_name domain1.com;  
    charset UTF-8;  
  
    if ($host !~ ^domain1.com$ ) {  
        return 444;  
    }  
  
    location / {  
        proxy_pass http://wp1;  
        proxy_set_header X-Real-IP $remote_addr;
```

```

proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_set_header X-NginX-Proxy true;
proxy_set_header Host $host;
proxy_set_header X-Forwarded-Proto $scheme;
proxy_redirect off;
}
}

```

Save the file and on most Linux systems, you can test it for syntax errors using the command `sudo /etc/init.d/nginx configtest` or `sudo service nginx configtest`.



Make sure that you have shut down HAProxy before you start Nginx, or you will get an error saying that Nginx can't bind to port 80. You can do this with the following command:

`/etc/init.d/haproxy stop`

If the test was successful, we can now restart (or start) the Nginx server. Again, use `sudo /etc/init.d/nginx restart` or `sudo service nginx restart` on most systems.

Head over to your web browser and enter the URL `domain1.com` to take a look at our WordPress installation site. To make sure nothing but `domain1.com` works, try to go to `domain2.com` and expect no response.

Adding more domains to Nginx

To add another domain to match in Nginx, you can create a new file in the `/etc/nginx/conf.d/` directory and reload the Nginx configuration, as shown in the following code:

```

cp /etc/nginx/conf.d/wp1.conf /etc/nginx/conf.d/wp2.conf
sed -i "s/wp1/wp2/g" /etc/nginx/conf.d/wp2.conf
sed -i "s/domain1/domain2/g" /etc/nginx/conf.d/wp2.conf
sed -i "s/49187/49188/g" /etc/nginx/conf.d/wp2.conf

#test config
/etc/init.d/nginx configtest

#reload config
/etc/init.d/nginx reload

```

Copy the configuration file, replace a few names, run `configtest`, and reload Nginx.

Try `domain1.com` in your browser to make sure it still works. You should still see the WordPress installation page (unless you installed WordPress, of course); head over to `domain2.com` after that to see whether our new configuration is used.

If you want to take a site down, just change the file's extension from `.conf` to something else and reload Nginx.

Automating the process of mapping domains

The limitations in this setup are that it's manual and hands-on every time a new domain is added. On my website (`http://oskarhane.com`), I've written some blog posts about how this process could be automated and those posts are my most-read posts of all time.

I was very glad when I found **nginx-proxy** by Jason Wilder. `nginx-proxy` solves this problem in a more clever way than me by monitoring Docker events via the Docker Remote API.



You can read more about `nginx-proxy` on its GitHub page (<https://github.com/jwilder/nginx-proxy>).



`nginx-proxy` comes as a container and we can run it by executing the following command:

```
docker run -d -p 80:80 -v /var/run/docker.sock:/tmp/docker.sock jwilder/nginx-proxy
```

We are giving the container our Docker socket, so it can listen for the events we are interested in, which are container starts and stops. We also bind the Docker hosts' port 80 to this new container, making it the entrance container for all incoming web requests. Make sure you stop Nginx on the Docker host before starting the `nginx-proxy` container. You can do this with the following command:

```
/etc/init.d/nginx stop
```

When a container starts, `nginx-proxy` creates an Nginx reverse proxy config file and reloads Nginx—just like we did, but fully automated with `nginx-proxy`.

To tell `nginx-proxy` which domain we want mapped to which container, we must run our containers with an environment variable named `VIRTUAL_HOST`.

In our `crane.yaml` file , we add an environment variable in the `wp` run section:

```
containers:  
  wp:  
    image: oskarhane/wordpress  
    run:  
      volumes-from: ["mydata"]  
      link:  
        - mymysql:mysql  
      publish: ["80"]  
      detach: true  
      env: ["VIRTUAL_HOST=domain1.com"]
```

Now, we just have to lift this with crane again to have this container mapped to the domain `domain1.com` on port 80:

```
crane lift web --recreate
```

Summary

In this chapter, we saw how you can solve the problem of having multiple containers that want to serve data on the same public port. We learned what a proxy server and reverse proxy server is and how a reverse proxy is used in load balancing.

We installed and configured two different reverse proxies: HAProxy and Nginx. In my workflow, the Nginx setup fits better, just copying a file, replacing a few words, and then reloading Nginx to have it working. HAProxy might work better in your setup; the choice is yours and one cannot be said to be better than the other.

`nginx-proxy` automates the process of creating a reverse proxy for containers that are started and is an OK solution for a PaaS, except for one thing: easy and straightforward deployment. That's what the next chapter is all about.

7

Deployment on Our PaaS

In the previous chapters, we went from setting up our PaaS in a very hands-on manner to a "hacked-together-automated" way by combining tools such as Crane and nginx-proxy. One part is still missing – how to deploy your code.

In this chapter we will go through the following topics:

- The problem with our current setup
- The tools/services available
- Dokku – mini-Heroku
- Setting up a WordPress app with Dokku

The problem with our current setup

Our current setup consists of three containers: a WordPress container, a MySQL container and a data volume container, tied together with Crane.

The main problem with our current setup using a VOLUME container as file storage is that we need a way into the volume to edit files. As of now, the only way to get into it is by mounting it on another container.

Another problem is that we don't version control our source code. We have just downloaded WordPress and some plugins and left it there. What if we update WordPress or make some other changes? We surely want to have that under version control.

If we want to keep the application architecture as it is, there are two options:

- Create a new container that mounts our data volume container, install it, and get access to it with SSH
- Install and open access to SSH in our WordPress container

With SSH installed, we can access the containers shell from a remote machine, and so, we can install Git to version control to our files. In this way, we can connect and push new code into the data volume container when we need to.

When connecting with SSH, you can go straight into the container without needing to connect to the Docker hosts shell.

If you are okay with connecting to the Docker host, and from there, if you open a new shell to get into your data volume container, a third option would be to SSH into your Docker hosts and then access the container with `docker exec -it container_name /bin/sh`.

While this certainly works, there are easier ways to do it.

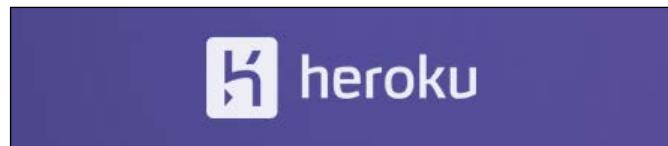
The tools/services available

When we look at hosted PaaS providers available today, two of them come to mind – OpenShift and Heroku. Many developers love Heroku because of its ease of use. Their philosophy gives a hint why:

"Developer Productivity:

Developer productivity is our battle cry, at the core of everything we do. Why require three steps when one will do? Why require any action at all when zero steps will do?"

Developers usually want to spend time on their code, not managing servers, deployment, and so on.



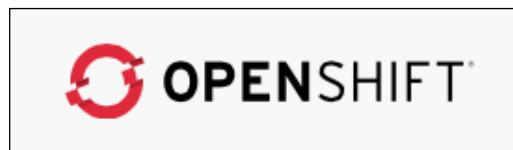
On Heroku, you get a remote Git repository into which you can push code. Your app's language and dependencies are identified by special files, depending on the language you use. Environment variables are used for configuration, and you instruct Heroku what to execute by specifying commands in a special file, called **Procfile**, that you include in your source code.

Whenever you push code into your remote Heroku Git repository, the app rebuilds and you have it online right away. If you have special build requirements, Heroku lets you create your own buildpacks where you can specify exactly what's to be done.

Basically, if you want to set up a WordPress blog on Heroku, you need to go through these steps:

1. Locally download the latest version of WordPress.
2. Create a Procfile and define what to execute (a buildpack that runs PHP and Apache2 in this case).
3. Create a `composer.json` file that specifies that PHP is a dependency.
4. Make some changes to the WordPress config files.
5. Create the Heroku app, add add-ons (such as a database), and define environment variables on Heroku.
6. Push your local code into Heroku.

When you make a change to the code, you just Git push to Heroku to deploy the new code. You cannot edit code directly on Heroku's servers, nor can you install themes or plugins (you have to do that locally and push the new code).



If you chose a provider such as OpenShift instead, you will have a bit more control over your PaaS. You can connect to it with SSH and also store static files downloaded by apps.

It is something like this we are looking for; it's just that we want to host our own platform and have Docker containers used in the background.

Dokku – Docker-powered mini-Heroku

Dokku can be found at <https://github.com/progrum/dokku>. It is a project that is described by its authors as follows:

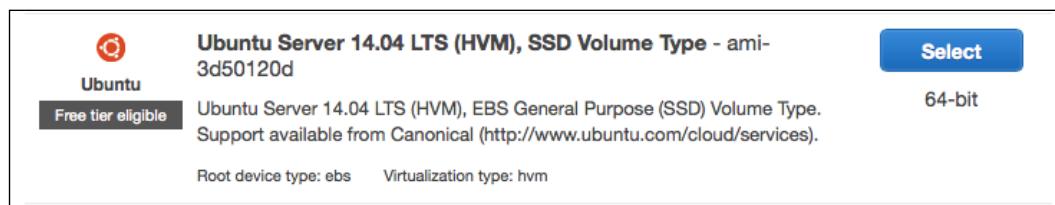
"Docker powered mini-Heroku in around 100 lines of Bash."

Feature wise, Dokku carries out deployment in the same way as Heroku does. Let's install Dokku and see what it can do for our PaaS.

Installation

Dokku requires Ubuntu 14.04 to run, and we start by creating a new EC2 instance running that.

Here is a screenshot of what we see:



When we have created an instance and have it up and running, we can start by installing Docker itself:

```
sudo apt-get install docker.io
```

When that is done, we go ahead and install Dokku.

The recommended bootstrap bash installation didn't work for me, so I cloned the repo instead:

```
cd /tmp
git clone https://github.com/progrum/dokku.git
cd dokku
sudo make install
dokku version
```



You can read about the installation process on the official installation page at <http://progrum.viewdocs.io/dokku/installation>.

The installation part will take a while, but it should succeed.

According to the document through the preceding link, we should edit the `/home/dokku/VHOST` file to hold the content of a domain name we plan to use. We skip this for now because it includes setting some DNS records. When we leave that file empty, we will be reaching our PaaS in the form of `http://ip:port`. We will come back to this step at a later point.

The only step left now is to create an `ssh` key pair on our local machine and add the public part in the server user Dokku's `authorized_keys` file so that we can connect with Git in a very secure way without using a password.

On your local machine, use these commands:

```
cd ~/.ssh  
ssh-keygen -t rsa  
#I named my key pair id_rsa  
cat id_rsa.pub  
#copy the output so you have it in your clipboard
```

On the server, use the following:

```
#As your ubuntu user  
#Replace <publickey> with the key you just copied  
#<remoteuser> can be replaced with anything, like "remoteuser".  
echo "<publickey>" | sudo sshcommand acl-add dokku <remoteuser>
```

If you name your `ssh-key` something other than `id_rsa`, you will have to edit your local `.ssh/config` file to get it to work.

Now the Dokku configuration is done and we should be able to start using it.

Creating a sample Dokku app

It is time for us to set up a demo app just so that you can learn the process. In this case, let's take Heroku's `Node.js` sample app.

We start off by cloning Heroku's `node-js-sample` GitHub repository to get the app's content. The following tasks are all supposed to be done on your local machine, and when I enter `server.com`, you should enter the URL or the IP address of your server. If you use a domain, make sure that you've set up DNS records for it or entered a record in your local `/etc/hosts` file:

```
#Clone the repo  
git clone git@github.com:heroku/node-js-sample.git  
cd node-js-sample
```

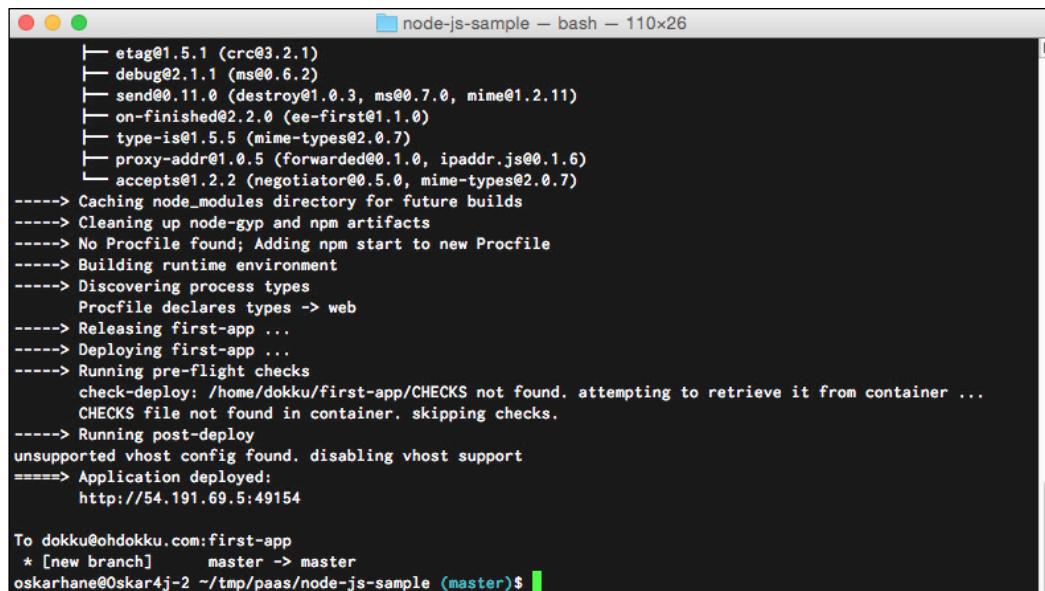
```
#Add a Dokku git remote  
git remote add dokku dokku@server.com:first-app  
  
#Push to Dokku  
git push dokku master
```

When we push to a non-existing branch or app name in Dokku, Dokku will create a new app and deploy it. When the push is done, you should see something like this at the bottom of the output:

```
=====> Application deployed:  
      http://54.191.69.5:49154
```

Of course, the IP address and port will not be the same for you.

The output is displayed, as follows:



The screenshot shows a terminal window titled "node-js-sample — bash — 110x26". The window displays the deployment logs for a Node.js application. The logs include:

- Dependency resolution: etag@1.5.1 (crc@3.2.1), debug@2.1.1 (ms@0.6.2), send@0.11.0 (destroy@1.0.3, ms@0.7.0, mime@1.2.11), on-finished@2.2.0 (ee-first@1.1.0), type-is@1.5.5 (mime-types@2.0.7), proxy-addr@1.0.5 (forwarded@0.1.0, ipaddr.js@0.1.6), accepts@1.2.2 (negotiator@0.5.0, mime-types@2.0.7).
- Build steps: Caching node_modules directory for future builds, Cleaning up node-gyp and npm artifacts, No Procfile found; Adding npm start to new Procfile, Building runtime environment, Discovering process types (Procfile declares types -> web), Releasing first-app ..., Deploying first-app ..., Running pre-flight checks (check-deploy: /home/dokku/first-app/CHECKS not found, attempting to retrieve it from container ...), CHECKS file not found in container, skipping checks.
- Post-deployment: Running post-deploy (unsupported vhost config found, disabling vhost support), Application deployed: http://54.191.69.5:49154.

At the bottom of the terminal, the prompt shows the deployment was successful to the 'dokku@ohdokku.com:first-app' server, and the current branch is 'master'.

Enter the ip:port in your web browser to find yourself a page saying **Hello World**. We've just deployed our first app on Dokku!

To modify and redeploy the site, we can create a file named `index.html` inside `public/` folder in our `node-js-sample` project. This node app will always look for files in the public folder. If the requested file isn't found, the app falls back to just printing **Hello World**. So, if we create a file and request it, the node server will serve it to us.

Paste this as the content of the `index.html` file:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello</title>
  </head>
  <body>
    <h1>First edit!</h1>
  </body>
</html>
```

It's a simple HTML page.

Let's go ahead and create the file and push it, as shown in the following code:

```
nano public/index.html
#paste the HTML
#save the file

#commit your changes
git add public/index.html
git commit -m "Added first HTML page."

#push to dokku
git push dokku master
```



Note from the output that the port will change every time you deploy, since a new container is created and your old container is shut down.

]

Later, when we add a domain name to deploy on, the URL will, of course, be the same. The Nginx config file is updated upon deployment. Point your browser to the new ip:port, and you should see a huge headline saying **First edit!**.

Whenever you make edits, just push them. Dokku will take care of the rest.

How Dokku works

As I described the basic step of Heroku earlier, you might recognize the steps when deploying on Dokku, and that is also Dokku's goal. They want people like us to feel comfortable with the deployment process.

Dokku can be seen as the glue between the following tools: Docker, Buildstep, ssh-command, pluginhook, ssh, git, and nginx. The source code is just about 100 lines long, and it ships with a few plugins that together contain about 500 lines of code. This is the power of Dokku – anyone can write plugins to extend the functionality of Dokku.

We have not yet installed any plugins, and a clean installation like ours can do only basic stuff such as deploy, see an app's logs, delete an app, and run a command in the app's container. There are quite a lot of plugins; they are all listed at <http://programm.viewdocs.io/dokku/plugins>.

The receive process

If we take a look at the main Dokku file (named `dokku` in the projects root), we notice that whenever a `receive` action is triggered (which happens when we push to the master branch), we see this code:

```
case "$1" in
  receive)
    APP="$2"; IMAGE="dokku/$APP"
    echo "----> Cleaning up ..."
    dokku cleanup
    echo "----> Building $APP ..."
    cat | dokku build $APP
    echo "----> Releasing $APP ..."
    dokku release $APP
    echo "----> Deploying $APP ..."
    dokku deploy $APP
    echo "===== Application deployed:"
    dokku urls $APP | sed "s/^/      /"
    echo
  ;;
;
```

Through this output we can recognize when we have pushed to the master.

If we follow the plugin chain when `deploy` is called, we end up with a plugin hook named `post-deploy` being called. A standard plugin, named `nginx-vhosts`, is triggered, and this in turn calls a function inside that plugin named `nginx:build-config`.

A code snippet from that preceding file looks like this:

```

case "$1" in
    nginx:build-config)
        APP="$2"; DOKKU_APP_LISTEN_PORT="$3"; DOKKU_APP_LISTEN_IP="${4}"
        VHOST_PATH="$DOKKU_ROOT/$APP/VHOST"
        WILDCARD_SSL="$DOKKU_ROOT/tls"
        SSL="$DOKKU_ROOT/$APP/tls"

        if [[ -z "$DOKKU_APP_LISTEN_PORT" ]] && [[ -f "$DOKKU_ROOT/$APP/
PORT" ]]; then
            DOKKU_APP_LISTEN_PORT=$(< "$DOKKU_ROOT/$APP/PORT")
        fi
        if [[ -z "$DOKKU_APP_LISTEN_IP" ]] && [[ -f "$DOKKU_ROOT/$APP/IP" ]]
        ]]; then
            DOKKU_APP_LISTEN_IP=$(< "$DOKKU_ROOT/$APP/IP")
        fi

        [[ -f "$DOKKU_ROOT/$APP/ENV" ]] && source $DOKKU_ROOT/$APP/ENV

        if [[ ! -n "$NO_VHOST" ]] && [[ -f "$DOKKU_ROOT/$APP/VHOST" ]];
        then
            ...
            NGINX_CONF="$PLUGIN_PATH/nginx-vhosts/templates/nginx.conf"
            SCHEME="http"
            ...
            APP_NGINX_TEMPLATE="$DOKKU_ROOT/$APP/nginx.conf.template"
            if [[ -f $APP_NGINX_TEMPLATE ]]; then
                echo "----> Overriding default nginx.conf with detected
nginx.conf.template"
                NGINX_CONF=$APP_NGINX_TEMPLATE
            fi

            xargs -i echo "----> Configuring {}..." < $VHOST_PATH
            # Include SSL_VHOSTS so we can redirect http to https on that
            hostname as well
            NOSSL_SERVER_NAME=$(echo $NONSSL_VHOSTS $SSL_VHOSTS| tr '\n' ' '
        )

        if [[ -n "$DOKKU_APP_LISTEN_PORT" ]] && [[ -n "$DOKKU_APP_
LISTEN_IP" ]]; then
            echo "----> Creating $SCHEME nginx.conf"
            echo "upstream $APP { server $DOKKU_APP_LISTEN_IP:$DOKKU_APP_
LISTEN_PORT; }" > $DOKKU_ROOT/$APP/nginx.conf

```

```
eval "cat <<< \$(< $NGINX_CONF) \>> $DOKKU_ROOT/$APP/nginx.conf"

echo "-----> Running nginx-pre-reload"
pluginhook nginx-pre-reload $APP $DOKKU_APP_LISTEN_PORT
$DOKKU_APP_LISTEN_IP

echo "      Reloading nginx"
restart_nginx
fi
else
if [[ -f "$DOKKU_ROOT/$APP/VHOST" ]]; then
echo "-----> VHOST support disabled, deleting $APP/VHOST"
rm "$DOKKU_ROOT/$APP/VHOST"
fi
if [[ -f "$DOKKU_ROOT/$APP/nginx.conf" ]]; then
echo "-----> VHOST support disabled, deleting nginx.conf"
rm "$DOKKU_ROOT/$APP/nginx.conf"

echo "-----> VHOST support disabled, reloading nginx after
nginx.conf deletion"
restart_nginx
fi
fi
;;
;
```

If we look through that code, we can see that it looks for a domain name in the `$DOKKU_ROOT/$APP/VHOST` file, and if that is found, sets some config variables and inserts them into a copy of the `templates/nginx.conf` file.

That file has these contents:

```
server {
listen      [::]:80;
listen      80;
server_name $NOSSL_SERVER_NAME;
location   / {
proxy_pass  http://$APP;
proxy_http_version 1.1;
proxy_set_header Upgrade \$http_upgrade;
proxy_set_header Connection "upgrade";
proxy_set_header Host \$http_host;
proxy_set_header X-Forwarded-Proto \$scheme;
proxy_set_header X-Forwarded-For \$remote_addr;
proxy_set_header X-Forwarded-Port \$server_port;
```

```
    proxy_set_header X-Request-Start \$msec;
}
include $DOKKU_ROOT/$APP/nginx.conf.d/*.conf;
}
```

Now that looks very much like the nginx config we created in the last chapter, right? The post-deploy part of Dokku is basically Jason Wilder's nginx-proxy. They accomplish the same result, but they get there in very different ways.

Dokku plugins

Add-ons in Heroku are called plugins in Dokku. Since we cannot specify `docker run` command parameters directly from Dokku, we need plugins to connect containers and add data volume containers.

Here's a list of a few usable Dokku plugins that we'll soon use.

Dokku domains plugin

Dokku domain plugin enables you to specify multiple domains in one app. By default, only one URL can be mapped to an app:

```
dokku domains:set myawesomeapp.com www.myawesomeapp.com
```

URL: <https://github.com/wmluke/dokku-domains-plugin>

Dokku-docker-options

With this plugin, you can pass any options to the Docker daemon when `docker run` command is executed. It can be used to mount volumes, link containers, and so on:

```
dokku docker-options:add myapp "-v /host/path:/container/path"
dokku docker-options:add myapp "-link container_name:alias"
```

URL: <https://github.com/dyson/dokku-docker-options>

Volume plugin for Dokku

Here's a plugin that enables you to mount volumes on your service containers. It also has commands to dump (export) and restore the data:

```
dokku volume:add foo /path/in/container
dokku volume:dump foo /path/in/container > foo.tar.gz
```

URL: <https://github.com/ohardy/dokku-volume>

Dokku-link

You can link containers with this plugin:

```
dokku link:create <app> NAME [ALIAS]  
dokku link:delete <app> NAME [ALIAS]
```

URL: <https://github.com/rlaneve/dokku-link>

MariaDB plugin for Dokku

This plugin enables you to create and use MariaDB containers. MariaDB can be used as a replacement for MySQL and is generally faster:

```
dokku mariadb:create <app>  
dokku mariadb:link <app> <db>  
dokku mariadb:dumpraw <app>
```

URL: <https://github.com/Kloadut/dokku-md-plugin>

MySQL plugin: <https://github.com/hughfletcher/dokku-mysql-plugin>

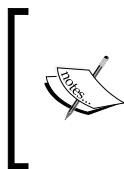
Setting up a WordPress app with Dokku

Now that we have played around with Dokku for a while, exploring how it works and what plugins are available, it's time to set up a WordPress site. After all, that's why we were exploring it in the first place.

This is what we are going to do:

1. Create a new local Git repository and download WordPress on it.
2. Install the MariaDB plugin, create a database, and link it to our app.
3. Configure WordPress to connect to our linked database.

On your local computer, download and unpack the latest version of WordPress and create a new Git repository. Create a `composer.json` file to tell Dokku that this is a PHP app we are creating.



You can read more about how to hint Dokku on what type of app you are creating at <https://devcenter.heroku.com/articles/buildpacks> (yes, Dokku uses Heroku buildpacks) and looks to detect functions. Dokku uses a library called Buildstep to make application builds using Docker and Buildpacks.

Let's go ahead and get started now.

I used a server on my domain, `ohdokku.com`, for this app:

```
#Download Wordpress
curl -O https://wordpress.org/latest.zip
unzip latest.zip
mv wordpress wp1
cd wp1

#Create a new Git repo
git init
git add .
git commit -m "Initial commit."

#Create a composer.json file to tell Dokku we are using php
echo '{}' > composer.json
git add .
git commit -am "Add composer.json for PHP app detection."

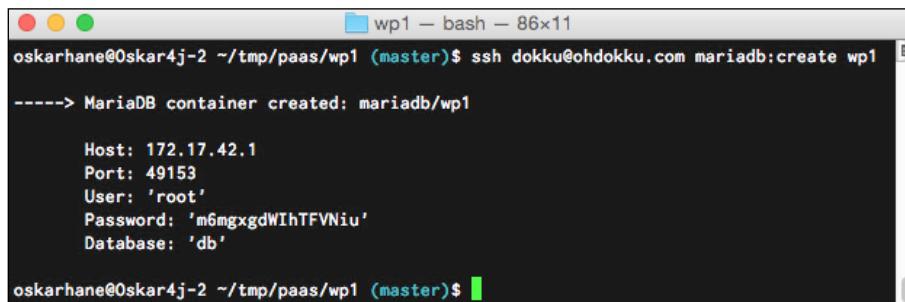
#Add a remote so we can push to Dokku
git remote add dokku dokku@ohdokku.com:wp1
```

On the server we have to install the MariaDB or MySQL plugin:

```
cd /var/lib/dokku/plugins
sudo git clone --recursive https://github.com/Kloadut/dokku-md-plugin
mariadb
cd mariadb/dockerfiles/
git checkout master
cd ../../
sudo dokku plugins-install
```

Back to the client side (you can do this on the server as well, but the whole point of this type of PaaS is being able to do all of this repetitive stuff on the client).

The result is as follows:



A screenshot of a terminal window titled "wp1 — bash — 86x11". The window shows the command `ssh dokku@ohdokku.com mariadb:create wp1` being run. The output indicates that a MariaDB container has been created with the following details:

```
oskarhane@Oskar4j-2 ~/tmp/paas/wp1 (master)$ ssh dokku@ohdokku.com mariadb:create wp1
-----> MariaDB container created: mariadb/wp1

  Host: 172.17.42.1
  Port: 49153
  User: 'root'
  Password: 'm6mgxgdWIhTFVNiu'
  Database: 'db'

oskarhane@Oskar4j-2 ~/tmp/paas/wp1 (master)$
```

As you can see, the output from the create command will show our database credentials.

Now that the database is set up, we can go ahead and push our app for the first time:

```
git push dokku master
```

You should notice that Dokku detects that you are pushing a PHP app. Since we haven't specified anything at all in our `composer.json` file, a default package of PHP and Apache2 will fire up.

Create a MariaDB database called `wp1_db`:

```
ssh dokku@ohdokku.com mariadb:create wp1_db
ssh dokku@ohdokku.com mariadb:link wp1 wp1_db
```

If we enter `ip:port` in a browser, a known page welcomes us – the WordPress installation page. When we click on the **Continue** button, we see that we can't continue before we create a `wp-config.php` file.

We have just created the link between the MariaDB container and the WP container, but we haven't made the link in code yet. WordPress has no idea how to connect to the database.

We start off by renaming the `wp-config-sample.php` file to `wp-config.php` and opening the file in an editor:

```
// ** MySQL settings - You can get this info from your web host ** //
/** The name of the database for Wordpress */
define('DB_NAME', getenv('DB_NAME'));

/** MySQL database username */
define('DB_USER', 'root');

/** MySQL database password */
define('DB_PASSWORD', getenv('DB_PASSWORD'));

/** MySQL hostname */
define('DB_HOST', getenv('DB_HOST') . ":" . getenv('DB_PORT'));
```

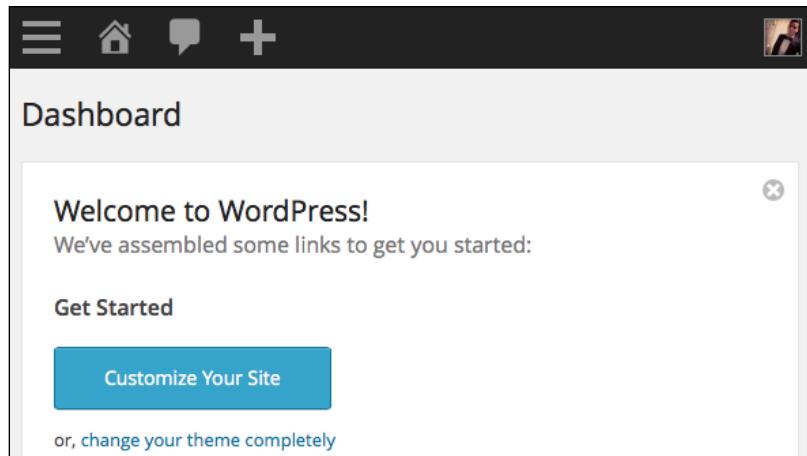
Edit the credentials as you just saw to make WordPress look for environment variables that our linked MariaDB gives us:

```
git add -A .
git commit -m "Add wp-config.php and add credentials."
git push dokku master
```

Wait until you get a new ip:port that our app is deployed to, and then enter the info in your web browser.

Now you should be able to install WordPress.

The output is as follows:



Starting multiple apps

To start multiple apps with Dokku, just repeat the simple process, as follows:

1. Create a local Git repository with WordPress in it, and create a remote Git repository.
2. Create and link a database with the MariaDB plugin.
Edit your `wp-config.php` file.
3. Push to deploy.

The name you set on your app when adding the remote from Git command:

```
git remote add dokku dokku@ohdokku.com:wp1
```

This command will create the URL to the WordPress site (`wp1.ohdokku.com`). You can set a complete custom domain as the name like: `git remote add dokku dokku@ohdokku.com:wp1.oskarhane.com` that will work if I point `wp1.oskarhane.com` to my server.

Adding a domain to Dokku

I waited with setting up domains to Dokku, since it involves logging in to a DNS provider and setting up DNS records to point the domain to our server. We set up DNS records to point our domain name to our server's IP address so that our server can be reached by entering our domain name in the web browser's location bar.

I usually use Amazon Route 53 to handle DNS for domains, since they're very stable and easy to use. It costs about a dollar a month for low-traffic sites. The setup is the same for any DNS provider. You have to add two records, one for `yourdomain.com` and one for `*.yourdomain.com`.

The records we are going to enter are A-records, which means that we point the domain names to a specific IPv4 address. The **Time To Live (TTL)** value is not important right now, but it means TTL and tells all other DNS servers that get requests for this domain how long they can cache the current value.

The output is as follows:

Create Record Set

Name:

Type:

Alias: Yes No

TTL (Seconds):

Value:

IPv4 address. Enter multiple addresses
on separate lines.
Example:
192.0.2.235
198.51.100.234

Routing Policy:

Route 53 responds to queries based only on the values in this record. [Learn More](#)

You should, of course, change the IP to the public IP your server has. When setting the A-record for the wildcard subdomains, just enter * in the input field at the top.

To see whether your DNS provider can resolve your domain name, execute `ping yourdomain.com` in a terminal. You'll see the resolved IP right there. If you've just bought the domain, you should be able to see the result right away, but if you've used the domain for a while, the old TTL value might delay the effect a bit.

If you want to wait for the setting of DNS records (which is common during development), you can set local records on your computer by editing the `/etc/hosts` file, as shown in the following command snippet:

```
sudo nano /etc/hosts

#Add this line to the file
54.191.69.5 ohdokku.com
#Save and exit
```

One thing to remember here is that you can't enter records for wildcard subdomains. If you plan to develop multiple apps on subdomains, you have to enter one record for each of them. Also, don't forget to remove these records when you're done; it can get quite confusing (and interesting) when you forget you have records for the domains you used.

On the Dokku server, create a file named `/home/dokku/VHOST` and enter `yourdomain.com` in it.

All apps being created from now on will be subdomains of this domain, unless you give the apps complete domain names.

More notes on Dokku

Just like Heroku, Dokku makes it easy for developers to deploy and push code. If you download a WordPress plugin straight from your Dokku app, it will be gone when you restart your Dokku app. It is advisable to keep a local copy that can easily be started or a dev, test, and staging server that you can download new plugins on and push to your Dokku app from to ensure they are persistent.



Images and videos should be uploaded to something such as Amazon via a plugin when using this kind of infrastructure.

You must also have your WordPress site send e-mails from an external e-mail provider, such as Mandrill. A plugin like WP Mail SMTP will solve that for you.

We still have a few manual steps (for example, downloading WordPress and editing `wp-config.php`) to do when deploying a WordPress app on Dokku, but the task of creating a custom Buildstep to remove the manual parts is beyond the scope of this book.

Another option is to have Composer handle the installation of WordPress with the `composer.json` file, but WordPress does not officially support this and it requires a few hacks, so I'll leave that up to you.



If you want to learn more about composer, you can go to the provided link <http://wpackagist.org>.



Summary

In this chapter, we went all the way to create our own PaaS by adding deployment to the process. What we looked into up to this chapter was all about organizing containers and direct incoming traffic so that visitors can reach the correct container.

With Dokku, we don't have to worry about that; all we have to care about is our code. As soon as we push our code, Dokku takes over and does the right things. Dokku makes it look really easy and that is why, I started from manually creating and linking containers and configuring reverse proxies – so that you would understand what Dokku does.

The next chapter takes us to the bleeding edge: what's being developed right now that can take private PaaS with Docker one step further?

8

What's Next?

So far, we have run our PaaS on a single host, which can be a problem if we need to scale out. There is a lot happening in this space, and I have selected a few projects that I will introduce in this chapter. These projects vary a lot in how mature they are, one is ready for use in production while the other is in a prototype state. In this chapter, we will cover the following topics:

- What is a Twelve-Factor app?
- Flynn
- Deis
- Rocket
- Orchestration tools

What is a Twelve-Factor app?

Many of today's apps are, in fact, web apps that you run in your web browser. Gmail, Slack, Wunderlist, Trello, and so on are all web apps or software-as-a-service.

It is these kind of apps that are suitable to be run on a PaaS.

The Twelve-Factor app is a methodology for building software-as-a-service apps that fulfill the following criteria:

- Use declarative formats to set up automation as well as to minimize the time and cost for new developers who join the project
- Have a clean contract with the underlying operating system, offering maximum portability between execution environments
- Suitable for deployment on modern cloud platforms, obviating the need for servers and systems administration

- Minimize divergence between development and production, enabling continuous deployment for maximum agility
- Scale up without significant changes to tooling, architecture, or development practices

The Twelve Factors are defined as follows:

- **Codebase** (One codebase tracked in revision control, many deploys): This puts your code in a version control system such as Git.
- **Dependencies** (Explicitly declare and isolate dependencies): This lists all the versions of all the libraries that your app depends on in a single place.
- **Config** (Store config in the environment): Since config will vary between environments such as the username or pass to a database, it should not be part of the code. You can set the config file in environment variables and have your app read them in at runtime.
- **Backing Services** (Treat backing services as attached resources): These have all the backing services, such as mail server, database, and cache system, among others. These will be referenced by a URL endpoint. This way your code doesn't have to care whether the backing service is running on the same machine or across the world.
- **Build, release, run** (Strictly separate build and run stages): The build stage creates bundles, assets, and binaries. This is the developer's job. When you've placed a package on a server, you are ready to enter the run stage by starting your application and making it available on the server. This stage should be as easy as possible so that anyone can do it.
- **Processes** (Execute the app as one or more stateless processes): As stated earlier in this book, you should separate your application data from your application service, that is, it makes the service stateless. All the states should be in the shared storages and databases.
- **Port binding** (Export services via port binding): An example is backing services; your service should be reachable via a URL endpoint.
- **Concurrency** (Scale out via the process model): This keeps every process as an independent service. This way you can scale just the parts of your app that really need to be scaled.
- **Disposability** (Maximize robustness with a fast startup and graceful shutdown): This is for app startup, which should be fast, and your app should be able to recover from a crash by itself.

- **Dev/prod parity** (Keep development, staging, and production as similar as possible): This keeps your development environment and setup as equal as possible to your production environment and setup. Docker really excels here.
- **Logs** (Treat logs as event streams): Place your app's error logs into a central place where you get notified when a new error has occurred.
- **Admin processes** (Run admin/management tasks as one-off processes): If you are doing administrative tasks, run them on a machine in the production environment with the latest code base. You should run queries directly against the database.

I encourage you to go to <http://12factor.net> in order to read more about each one of the Twelve Factors. It's a good read; you will get an understanding of why some design decisions were made on the following projects.

Flynn



The guy who created Dokku, Jeff Lindsay, has also co-created Flynn. Flynn is like a super-Dokku that, among other things, lets you run your PaaS on multiple hosts.

"Flynn is two things:

A distribution of components that out-of-the-box gives companies a reasonable starting point for an internal platform for running their applications and services.

The banner for a collection of independent projects that together make up a toolkit or loose framework for building distributed systems.

Flynn is both a whole and many parts, depending on what is most useful for you. The common goal is to democratize years of experience and best practices in building distributed systems. It is the software layer between operators and developers that makes both their lives easier."

I have tried using Flynn a few times, but I have always gone back to using Dokku again because I find Dokku easier to use, and my clients don't need the extra features such as multihost PaaS.

URL: <http://flynn.io>

Status: This is not suitable for use in the production environment because it's in a beta stage.

Deis



Deis is built on a lightweight Linux distribution that is built to run containers, called CoreOS, and on Docker to take advantage of the distributed services, such as etcd, available there.

"Deis is a lightweight application platform that deploys and scales Twelve-Factor apps as Docker containers across a cluster of CoreOS machines."

I found Deis to be a very promising project and would like to work with it more. I have barely touched it but what I have seen so far looks good.

Deis can deploy any language or framework running on Linux using Docker, and it also includes Heroku buildpacks for Ruby, Python, Node.js, Java, Clojure, Scala, Play, PHP, Perl, Dart, and Go.

The workflow is Heroku-like and you just need to deploy twelve-factor apps, that is, save the application state in a backing service.

Fun fact: Deis financially backs/supports Dokku.

URL: <http://deis.io>

State: Deis is ready for production from version 1.0.

Rocket



CoreOS has been one of the most popular ways to run a multihost Docker PaaS. They have done excellent work and have built some multihost PaaS tools, such as Deis, that use CoreOS tools and services to deliver their version of PaaS.

In December 2014, the CoreOS team decided to announce their own container runtime: Rocket. Rocket is a direct competitor to the original Docker. The reason why they are launching Rocket is because they believe Docker has lost its initial idea: running reusable standard containers. The CoreOS team believes that Docker is stepping away from the initial idea by adding more and more features and services around the Docker environment.

"Rocket is a new container runtime, designed for composability, security, and speed. Today we are releasing a prototype version on GitHub to begin gathering feedback from our community and explain why we are building Rocket."

According to the CoreOS team, they will continue to have CoreOS to be the perfect thing to run Docker. I guess we will see what happens in the future, but I hope they stand by their words.

URL: <https://github.com/coreos/rocket>

State: Rocket is in its very early state and not ready for production.

Orchestration tools

The tools I have introduced now are tools that will help you keep your mind on the code and give you an easy way to deploy your apps to production. If you are more interested in an orchestration tool—a tool that helps you manage container clusters—there are a few of them out there as well. The tools that currently come to mind are Google's Kubernetes, Apache Mesos/Marathon, CoreOS Fleet, and the soon to be released Swarm from Docker.

Summary

When you feel it's time to move your PaaS from a single host to scale across multiple hosts, these tools are what you should be looking for. I'm sure some worthy competitors will pop up in the future since this is a hot area right now.

Index

A

Amazon
 URL 8
Amazon EC2
 Docker, installing 8-11
 Docker, upgrading 12
 using 7
Apache
 preparing, for caching 32, 33

B

base image 15
Buildpacks
 reference link 104
Buildstep 104
BusyBox 54

C

command-line interface 17, 18
commands, Docker
 about 13, 14
 docker images 13
 docker ps 13
 docker ps -a 13
 docker run 13
 docker stop 13
composer
 reference link 110
container ID 17
containers
 about 2, 16, 17
 connecting, Crane used 70
 connecting, Docker Compose used 67
 connecting, manually 63, 64

parameters, passing 59
setup issue 93, 94

Crane

 about 63, 70
 configuring 71-74
 graph command 71
 installing 71
 lift command 71
 logs command 71
 status command 71
 usage 71
 used, for connecting containers 70

D

data volume container
 contents, exploring 65, 66
 executing 58
 mounting 52
data volume image
 BusyBox 54
 creating 53
 Dockerfile 55
 mount points, exposing 54
data volumes
 about 51
 backup 53
 data volume container, mounting 52
 host directory, mounting 52
 restoring 53
Deis
 about 114
 URL 114
Docker
 about 1
 URL 1

Docker Compose

- about 63, 67
- build command 68
- installing 67, 68
- kill command 68
- logs command 68
- PaaS, setting up 69, 70
- port command 68
- ps command 68
- pull command 68
- rm command 68
- run command 68
- run command, using 69
- scale command 68
- scale command, using 69
- service 68
- start command 68
- stop command 68
- up command 68
- used, for connecting containers 67

Dockerfile

- about 20
- creating, on WordPress image 43

Dockerfile, for PHP 5.6

- URL 34

Docker image

- about 15, 16
- base image 15
- hosting, on GitHub 55, 56
- parent images 15
- publishing, on Docker Registry Hub 57

docker images command 13**Docker on Amazon EC2**

- installing 8-11
- open ports 12
- upgrading 12
- user permissions 12

Docker on Mac OS X

- installing 3-5
- upgrading 6

Docker on Ubuntu Trusty 14.04 LTS

- installing 2
- upgrading 3
- user permissions 3

Docker on Windows

- installing 6, 7
 - upgrading 7
- docker ps -a command 13**
- docker ps command 13**
- Docker Registry Hub**
- about 19
 - Docker image, publishing 57
 - image, publishing 46
 - image, publishing with automated build option 47-49
 - published images, exploring 21-27
 - repositories, browsing 19, 20
 - URL 19

docker run command 13**docker stop command 13****Dokku**

- about 96
- deploying 100
- domains, adding 108, 109
- installing 96, 97
- multiple apps, starting 107
- plugins 103
- receive process 100-102
- sample app, creating 97-99
- URL 96
- WordPress app, deploying 109, 110
- WordPress app, setting up 104-106

Dokku-docker-options

- about 103
- URL 103

Dokku domains plugin

- about 103
- URL 103

Dokku-link plugin

- about 104
- URL 104

domains

- adding, to Dokku 108, 109
- mapping, nginx-proxy used 90, 91

F**Flynn**

- about 113, 114
- URL 114

G

GitHub

Docker image, hosting 55, 56
image sources, hosting 44-46
URL 45

H

HAProxy

about 81
configuring 82-85
installing 81, 82
multiple domains, adding 85, 86
URL 81

Heroku 94, 95

host directory

mounting, as data volume 52

I

image sources

hosting, on GitHub 44-46

installation, Docker

on Amazon EC2 8-11
on Mac OS X 3-5
on Ubuntu Trusty 14.04 LTS 2
on Windows 6, 7

installation

for Crane 71
for Docker Compose 67, 68
for Dokku 96, 97
for HAProxy 81, 82
for Nginx 87
for WP Mail SMTP 36-42
for WP Super Cache 36-42

M

Mac OS X

Docker, installing 3-5
Docker, upgrading 6

MariaDB plugin

about 104
URL 104

multiple containers, with same services

problem 78
solution, finding 78-80
solution, implementing 80, 81

MySQL docker repository

URL 25

N

Nginx

about 86, 87
configuring 87-89
installing 87
multiple domains, adding 89
URL 86

nginx-proxy

URL 90
used, for mapping domains 90, 91

O

OpenShift 94, 95

orchestration tools 116

OS X installer

URL 4

P

parameterized image

creating 59-61

parent images 15

Platform as a Service (PaaS)

about 1
setting up, with Docker Compose 69, 70

plugins, Dokku

about 103
Dokku-docker-options 103
Dokku domain plugin 103
Dokku-link plugin 104
MariaDB plugin 104
volume plugin 103

Procfile 95

proxy server 78

published images

exploring 21-27

R

receive process, Dokku 100-102
repositories
 browsing 19, 20
reverse proxies 79
Rocket
 about 115
 URL 115

S

solution, multiple containers with same services
 implementing, with HAProxy 81
 implementing, with Nginx 86, 87

T

tags 16
Time to live (TTL) 108
tools/services
 Heroku 94, 95
 OpenShift 94, 95
twelve factors
 about 111, 112
 admin processes 113
 backing services 112
 build 112
 codebase 112
 concurrency 112
 config 112
 dependencies 112
 dev/prod parity 113
 disposability 112
 logs 113
 port binding 112
 processes 112
 release 112
 run 112
 URL 113

U

Ubuntu Trusty 14.04 LTS
 Docker, installing 2
 Docker, upgrading 3

V

volume plugin
 about 103
 URL 103

W

Windows
 Docker, installing 6
 Docker, upgrading 7
 installer, URL 6
WordPress app
 configuring 31, 32
 deploying, on Dokku 109, 110
 domains, adding to Dokku 108, 109
 multiple apps, starting 107
 setting up, with Dokku 104-106
WordPress Docker image
 URL 21
WordPress image
 creating 29, 30
 Dockerfile, creating 43
 objective 32
WordPress image, objectives
 Apache, preparing for caching 32, 33
 upload limit, raising 34-36
 WP Mail SMTP, installing 36-42
 WP Super Cache, installing 36-42
WP Mail SMTP
 installing 36-42
WP Super Cache
 installing 36-42



**Thank you for buying
Build Your Own PaaS with Docker**

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

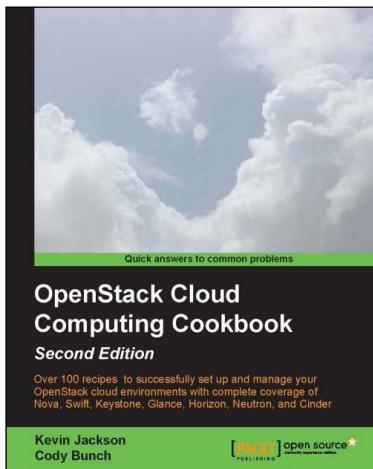
Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



OpenStack Cloud Computing Cookbook

Second Edition

ISBN: 978-1-78216-758-7 Paperback: 396 pages

Over 100 recipes to successfully set up and manage your OpenStack cloud environments with complete coverage of Nova, Swift, Keystone, Glance, Horizon, Neutron, and Cinder

1. Updated for OpenStack Grizzly.
2. Learn how to install, configure, and manage all of the OpenStack core projects including new topics like block storage and software defined networking.
3. Learn how to build your Private Cloud utilizing DevOps and Continuous Integration tools and techniques.



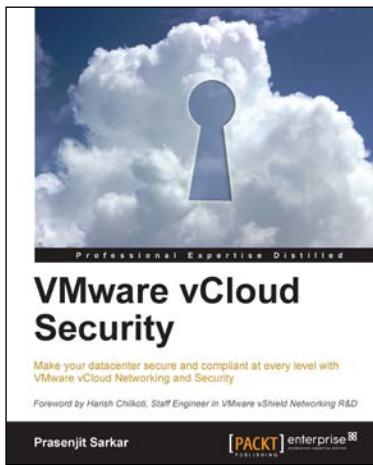
Cloud Development and Deployment with CloudBees

ISBN: 978-1-78328-163-3 Paperback: 114 pages

Develop and deploy your Java application onto the Cloud using CloudBees

1. Create, deploy, and develop applications using CloudBees.
2. Impress your colleagues and become a pro by using different tools to integrate CloudBees with SDK.
3. A step-by-step tutorial guide which will help you explore and maintain real-world applications with CloudBees.

Please check www.PacktPub.com for information on our titles



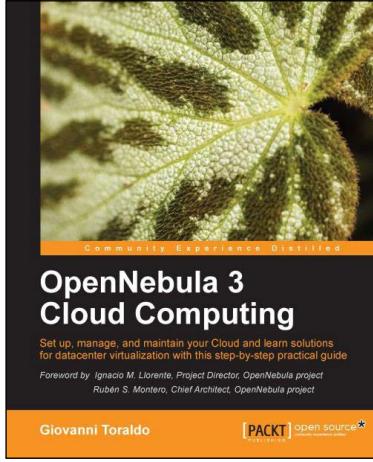
VMware vCloud Security

ISBN: 978-1-78217-096-9

Paperback: 106 pages

Make your datacenter secure and compliant at every level with VMware vCloud Networking and Security

1. Take away an in-depth knowledge of how to secure a private cloud running on vCloud Director.
2. Enable the reader with the knowledge, skills, and abilities to achieve competence at building and running a secured private cloud.
3. Focuses on giving you broader view of the security and compliance while still being manageable and flexible to scale.



OpenNebula 3 Cloud Computing

ISBN: 978-1-84951-746-1

Paperback: 314 pages

Set up, manage, and maintain your Cloud and learn solutions for datacenter virtualization with this step-by-step practical guide

1. Take advantage of open source distributed file-systems for storage scalability and high-availability.
2. Build-up, manage and maintain your Cloud without previous knowledge of virtualization and cloud computing.
3. Install and configure every supported hypervisor: KVM, Xen, VMware.

Please check www.PacktPub.com for information on our titles