

The Docker Book



by James Turnbull

The Docker Book

James Turnbull

October 8, 2014

Version: v1.2.2 (19946fc)

Website: [The Docker Book](#)



Some rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical or photocopying, recording, or otherwise, for commercial purposes without the prior permission of the publisher.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit [here](#).

© Copyright 2014 - James Turnbull <james@lovedthanlost.net>



Contents

	Page
List of Figures	ix
List of Listings	xxii
Foreword	1
Who is this book for?	1
Credits and Acknowledgments	1
Technical Reviewers	2
Scott Collier	2
John Ferlito	2
Paul Nasrat	3
Technical Illustrator	3
Proofreader	3
Author	4
Conventions in the book	4
Code and Examples	4
Colophon	5
Errata	5
Version	5
Chapter 1 Introduction	6
Introducing Docker	7
An easy and lightweight way to model reality	8
A logical segregation of duties	8
Fast, efficient development life cycle	8
Encourages service orientated architecture	9

Docker components	9
Docker client and server	9
Docker images	11
Registries	11
Containers	12
What can you use Docker for?	13
Docker with configuration management	13
Docker's technical components	15
What's in the book?	15
Docker resources	16
Chapter 2 Installing Docker	18
Requirements	19
Installing on Ubuntu	20
Checking for prerequisites	21
Installing Docker	23
Docker and UFW	24
Installing on Red Hat and family	25
Checking for prerequisites	26
Installing Docker	27
Starting the Docker daemon on the Red Hat family	28
Boot2Docker installation on OS X	29
Installing Boot2Docker on OS X	30
Setting up Boot2Docker on OS X	31
Testing Boot2Docker	32
Boot2Docker installation on Windows	32
Installing Boot2Docker on Windows	32
Setting up Boot2Docker on Windows	33
Testing Boot2Docker	34
Using Boot2Docker with this book	35
Docker installation script	37
Binary installation	38
The Docker daemon	38
Configuring the Docker daemon	39
Checking that the Docker daemon is running	40

Upgrading Docker	42
Docker user interfaces	42
Summary	42
Chapter 3 Getting Started with Docker	44
Ensuring Docker is ready	44
Running our first container	45
Working with our first container	48
Container naming	51
Starting a stopped container	51
Attaching to a container	52
Creating daemonized containers	53
Seeing what's happening inside our container	54
Inspecting the container's processes	55
Running a process inside a container	56
Stopping a daemonized container	57
Automatic container restarts	58
Finding out more about our container	59
Deleting a container	61
Summary	61
Chapter 4 Working with Docker images and repositories	62
What is a Docker image?	63
Listing Docker images	65
Pulling images	69
Searching for images	71
Building our own images	72
Creating a Docker Hub account	73
Using Docker commit to create images	75
Building images with a Dockerfile	77
Building the image from our Dockerfile	80
What happens if an instruction fails?	83
Dockerfiles and the build cache	84
Using the build cache for templating	85
Viewing our new image	86

Launching a container from our new image	87
Dockerfile instructions	90
Pushing images to the Docker Hub	104
Automated Builds	106
Deleting an image	112
Running your own Docker registry	115
Running a registry from a container	115
Testing the new registry	115
Alternative Indexes	117
Quay	117
Summary	117
Chapter 5 Testing with Docker	118
Using Docker to test a static website	119
An initial Dockerfile for the Sample website	119
Building our Sample website and Nginx image	123
Building containers from our Sample website and Nginx image . . .	125
Editing our website	128
Using Docker to build and test a web application	129
Building our Sinatra application	129
Creating our Sinatra container	130
Building a Redis image and container	133
Connecting to the Redis container	135
Our Redis connection	139
Linking Docker containers	141
Using our container link to communicate	146
Using Docker for continuous integration	149
Build a Jenkins and Docker server	150
Create a new Jenkins job	155
Running our Jenkins job	160
Next steps with our Jenkins job	162
Summary of our Jenkins setup	162
Multi-configuration Jenkins	163
Create a multi-configuration job	163
Testing our multi-configuration job	168

Summary of our multi-configuration Jenkins	170
Other alternatives	170
Drone	170
Shippable	171
Summary	171
Chapter 6 Building services with Docker	172
Building our first application	172
The Jekyll base image	173
Building the Jekyll base image	174
The Apache image	175
Building the Jekyll Apache image	177
Launching our Jekyll site	178
Updating our Jekyll site	181
Backing up our Jekyll volume	183
Extending our Jekyll website example	184
Building a Java application server with Docker	185
A WAR file fetcher	185
Fetching a WAR file	186
Our Tomcat 7 application server	188
Running our WAR file	190
Building on top of our Tomcat application server	191
A multi-container application stack	194
The Node.js image	195
The Redis base image	198
The Redis primary image	199
The Redis replica image	200
Creating our Redis back-end cluster	201
Creating our Node container	207
Capturing our application logs	208
Summary of our Node stack	212
Managing Docker containers without SSH	213
Summary	215
Chapter 7 Docker Orchestration and Service Discovery	216

Fig	217
Installing Fig	217
Getting our sample application	219
The <code>fig.yml</code> file	223
Running Fig	225
Using Fig	228
Fig in summary	231
Consul, Service Discovery and Docker	231
Building a Consul image	233
Testing a Consul container locally	237
Running a Consul cluster in Docker	238
Starting the Consul bootstrap node	242
Starting the remaining nodes	245
Running a distributed service with Consul in Docker	252
Orchestration alternatives and components	262
Fleet and etcd	263
Kubernetes	263
Apache Mesos	263
Helios	264
Centurion	264
Libswarm	264
Summary	264
Chapter 8 Using the Docker API	266
The Docker APIs	266
First steps with the Remote API	267
Testing the Docker Remote API	269
Managing images with the API	270
Managing containers with the API	273
Improving TProv	277
Authenticating the Docker Remote API	282
Create a Certificate Authority	283
Create a server certificate signing request and key	284
Configuring the Docker daemon	287
Creating a client certificate and key	289

Configuring our Docker client for authentication	291
Summary	293
Chapter 9 Getting help and extending Docker	294
Getting help	295
The Docker user and dev mailing lists	295
Docker on IRC	295
Docker on GitHub	296
Reporting issues for Docker	296
Setting up a build environment	297
Install Docker	297
Install source and build tools	297
Check out the source	297
Contributing to the documentation	298
Build the environment	299
Running the tests	300
Use Docker inside our development environment	301
Submitting a pull request	302
Merge approval and maintainers	304
Summary	305
Index	306

List of Figures

1.1 Docker architecture	10
2.1 Installing Boot2Docker on OS X	30
2.2 Running Boot2Docker on OS X	31
2.3 Installing Boot2Docker on Windows	33
2.4 Running Boot2Docker on Windows	34
3.1 Listing Docker containers	50
4.1 The Docker filesystem layers	64
4.2 Docker Hub	66
4.3 Creating a Docker Hub account.	74
4.4 Your image on the Docker Hub.	106
4.5 The Add Repository button.	107
4.6 Account linking options.	108
4.7 Linking your GitHub account	109
4.8 Selecting your repository.	110
4.9 Configuring your Automated Build.	111
4.10Creating your Automated Build.	112
4.11Deleting a repository.	114
5.1 Browsing the Sample website.	128
5.2 Browsing the edited Sample website.	129
5.3 Browsing the Jenkins server.	154
5.4 Creating a new Jenkins job.	155
5.5 Jenkins job details part 1.	156
5.6 Jenkins job details part 2.	159
5.7 Running the Jenkins job.	160

List of Figures

5.8 The Jenkins job details.	161
5.9 The Jenkins job console output.	161
5.10Creating a multi-configuration job.	163
5.11Configuring a multi-configuration job Part 1.	164
5.12Configuring a multi-configuration job Part 2.	165
5.13Our Jenkins multi-configuration job	167
5.14The centos sub-job.	168
5.15The centos sub-job details.	169
5.16The centos sub-job console output.	169
6.1 Our Jekyll website.	181
6.2 Our updated Jekyll website.	182
6.3 Our Tomcat sample application.	190
6.4 Our TProv web application.	192
6.5 Downloading a sample application.	193
6.6 Listing the Tomcat instances.	193
6.7 Our Node application.	207
7.1 Sample Fig application.	227
7.2 The Consul web interface.	238
7.3 The Consul service in the web interface.	250
7.4 The distributed_app service in the Consul web interface.	260
7.5 More distributed_app services in the Consul web interface.	261

Listings

1	Sample code block	4
2.1	Checking for the Linux kernel version on Ubuntu	21
2.2	Installing a 3.8 kernel on Ubuntu Precise	21
2.3	Updating the boot loader on Ubuntu Precise	22
2.4	Reboot the Ubuntu host	22
2.5	Checking for Device Mapper	22
2.6	Checking for Device Mapper in proc on Ubuntu	23
2.7	Loading the Device Mapper module	23
2.8	Adding the Docker APT repository	23
2.9	Testing for curl installation	23
2.10	Install curl if needed	24
2.11	Adding the Docker repository GPG key	24
2.12	Updating APT sources	24
2.13	Installing the Docker packages on Ubuntu	24
2.14	Checking Docker is installed on Ubuntu	24
2.15	Old UFW forwarding policy	25
2.16	New UFW forwarding policy	25
2.17	Reload the UFW firewall	25
2.18	Checking the Red Hat or Fedora kernel	26
2.19	Checking for Device Mapper	26
2.20	Checking for Device Mapper in proc on Red Hat	26
2.21	Installing the Device Mapper package	27
2.22	Loading the Device Mapper module	27
2.23	Installing EPEL on Red Hat Enterprise Linux 6 and CentOS 6	27
2.24	Installing the Docker package on Red Hat Enterprise Linux 6 and CentOS 6	27

2.25	Installing Docker on RHEL 7	28
2.26	Installing the Docker package on Fedora 19	28
2.27	Installing the Docker package on Fedora 20 and later	28
2.28	Starting the Docker daemon on Red Hat 6	28
2.29	Ensuring the Docker daemon starts at boot on Red Hat 6	29
2.30	Starting the Docker daemon on Red Hat 7	29
2.31	Ensuring the Docker daemon starts at boot on Red Hat 7	29
2.32	Checking Docker is installed on the Red Hat family	29
2.33	Downloading the Boot2Docker PKG file	30
2.34	Testing Boot2Docker on OS X	32
2.35	Downloading the Boot2Docker .EXE file	33
2.36	Testing Boot2Docker on Windows	35
2.37	Boot2Docker launch message	35
2.38	Getting the Boot2Docker IP address	36
2.39	Initial curl command	36
2.40	Updated curl command	36
2.41	Testing for curl	37
2.42	Installing curl on Ubuntu	37
2.43	Installing curl on Fedora	37
2.44	Installing Docker from the installation script	37
2.45	Downloading the Docker binary	38
2.46	Changing Docker daemon networking	39
2.47	Using the DOCKER_HOST environment variable	39
2.48	Binding the Docker daemon to a different socket	40
2.49	Binding the Docker daemon to multiple places	40
2.50	Turning on Docker daemon debug	40
2.51	Checking the status of the Docker daemon	41
2.52	Starting and stopping Docker with Upstart	41
2.53	Starting and stopping Docker on Red Hat and Fedora	41
2.54	The Docker daemon isn't running	41
2.55	Upgrade docker	42
3.1	Checking the docker binary works	45
3.2	Creating our first container	46
3.3	The docker run command	46
3.4	Our first container's shell	47

3.5	Checking the container's hostname	48
3.6	Checking the container's /etc/hosts	48
3.7	Checking the container's interfaces	49
3.8	Checking container's processes	49
3.9	Installing a package in our first container	49
3.10	Naming a container	51
3.11	Starting a stopped container	51
3.12	Starting a stopped container by ID	52
3.13	Attaching to a running container	52
3.14	Attaching to a running container via ID	52
3.15	Inside our re-attached container	53
3.16	Creating a long running container	53
3.17	Viewing our running daemon_dave container	53
3.18	Fetching the logs of our daemonized container	54
3.19	Tailing the logs of our daemonized container	54
3.20	Tailing the logs of our daemonized container	55
3.21	Inspecting the processes of the daemonized container	55
3.22	The docker top output	56
3.23	Running a background task inside a container	56
3.24	Running an interactive command inside a container	56
3.25	Stopping the running Docker container	57
3.26	Stopping the running Docker container by ID	57
3.27	Automatically restarting containers	58
3.28	On-failure restart count	58
3.29	Inspecting a container	59
3.30	Selectively inspecting a container	59
3.31	Inspecting the container's IP address	60
3.32	Inspecting multiple containers	60
3.33	Deleting a container	61
3.34	Deleting all containers	61
4.1	Revisiting creating a basic Docker container	62
4.2	Listing Docker images	65
4.3	Pulling the Ubuntu image	67
4.4	Listing all the ubuntu Docker images	67
4.5	Running a tagged Docker image	68

4.6	Docker run and the default latest tag	69
4.7	Pulling the fedora image	70
4.8	Viewing the fedora image	70
4.9	Pulling a tagged fedora image	71
4.10	Searching for images	71
4.11	Pulling down the jamtur01/puppetmaster image	72
4.12	Creating a Docker container from the Puppet master image	72
4.13	Logging into the Docker Hub	74
4.14	Creating a custom container to modify	75
4.15	Adding the Apache package	75
4.16	Committing the custom container	76
4.17	Reviewing our new image	76
4.18	Committing another custom container	76
4.19	Inspecting our committed image	77
4.20	Running a container from our committed image	77
4.21	Creating a sample repository	78
4.22	Our first Dockerfile	78
4.23	The RUN instruction in exec form	80
4.24	Running the Dockerfile	81
4.25	Tagging a build	82
4.26	Building from a Git repository	82
4.27	Uploading the build context to the daemon	82
4.28	Managing a failed instruction	83
4.29	Creating a container from the last successful step	84
4.30	Bypassing the Dockerfile build cache	84
4.31	A template Ubuntu Dockerfile	85
4.32	A template Fedora Dockerfile	85
4.33	Listing our new Docker image	86
4.34	Using the docker history command	86
4.35	Launching a container from our new image	87
4.36	Viewing the Docker port mapping	88
4.37	The docker port command	88
4.38	Exposing a specific port with -p	88
4.39	Binding to a different port	89
4.40	Binding to a specific interface	89

4.41 Binding to a random port on a specific interface	89
4.42 Exposing a port with docker run	90
4.43 Connecting to the container via curl	90
4.44 Specifying a specific command to run	91
4.45 Using the CMD instruction	91
4.46 Passing parameters to the CMD instruction	91
4.47 Overriding CMD instructions in the Dockerfile	92
4.48 Launching a container with a CMD instruction	92
4.49 Overriding a command locally	92
4.50 Specifying an ENTRYPOINT	93
4.51 Specifying an ENTRYPOINT parameter	93
4.52 Rebuilding static_web with a new ENTRYPOINT	94
4.53 Using docker run with ENTRYPOINT	94
4.54 Using ENTRYPOINT and CMD together	94
4.55 Using the WORKDIR instruction	95
4.56 Overriding the working directory	95
4.57 Setting an environment variable in Dockerfile	96
4.58 Prefixing a RUN instruction	96
4.59 Executing with an ENV prefix	96
4.60 Persistent environment variables in Docker containers	96
4.61 Runtime environment variables	97
4.62 Using the USER instruction	97
4.63 Specifying USER and GROUP variants	97
4.64 Using the VOLUME instruction	98
4.65 Using multiple VOLUME instructions	98
4.66 Using the ADD instruction	99
4.67 URL as the source of an ADD instruction	99
4.68 Archive as the source of an ADD instruction	99
4.69 Using the COPY instruction	100
4.70 Adding ONBUILD instructions	101
4.71 Showing ONBUILD instructions with docker inspect	102
4.72 A new ONBUILD image Dockerfile	102
4.73 Building the apache2 image	102
4.74 The webapp Dockerfile	103
4.75 Building our webapp image	103

4.76 Trying to push a root image	104
4.77 Pushing a Docker image	105
4.78 Deleting a Docker image	113
4.79 Deleting multiple Docker images	114
4.80 Deleting all images	114
4.81 Running a container-based registry	115
4.82 Listing the <code>jamtur01 static_web</code> Docker image	116
4.83 Tagging our image for our new registry	116
4.84 Pushing an image to our new registry	116
4.85 Building a container from our local registry	117
5.1 Creating a directory for our Sample website Dockerfile	119
5.2 Getting our Nginx configuration files	120
5.3 Our basic Dockerfile for the Sample website	120
5.4 The <code>global.conf</code>	121
5.5 The <code>nginx.conf</code> configuration file	122
5.6 Building our new Nginx image	123
5.7 Showing the history of the Nginx image	124
5.8 Downloading our Sample website	125
5.9 Building our first Nginx testing container	125
5.10 Controlling the write status of a volume	126
5.11 Viewing the Sample website container	127
5.12 Editing our Sample website	128
5.13 Old title	128
5.14 New title	128
5.15 Dockerfile for web application testing	130
5.16 Building our new Sinatra image	130
5.17 Download our Sinatra web application	131
5.18 Making <code>webapp/bin/webapp</code> executable	131
5.19 Launching our first Sinatra container	131
5.20 The <code>CMD</code> instruction in our Dockerfile	131
5.21 Checking the logs of our Sinatra container	132
5.22 Tailing the logs of our Sinatra container	132
5.23 Using <code>docker top</code> to list our Sinatra processes	132
5.24 Checking the Sinatra port mapping	132
5.25 Testing our Sinatra application	133

5.26 Dockerfile for Redis image	134
5.27 Building our Redis image	134
5.28 Launching a Redis container	134
5.29 Launching a Redis container	134
5.30 Installing the redis-tools package on Ubuntu	135
5.31 Testing our Redis connection	135
5.32 The docker0 interface	136
5.33 The veth interfaces	136
5.34 The eth0 interface in a container	137
5.35 Tracing a route out of our container	137
5.36 Docker iptables and NAT	138
5.37 Redis container's networking configuration	139
5.38 Finding the Redis container's IP address	139
5.39 Talking directly to the Redis container	140
5.40 Restarting our Redis container	140
5.41 Finding the restarted Redis container's IP address	140
5.42 Starting another Redis container	141
5.43 Linking our Redis container	142
5.44 Linking our Redis container	143
5.45 The webapp's /etc/hosts file	144
5.46 Pinging the db container	144
5.47 Showing linked environment variables	145
5.48 The linked via env variables Redis connection	146
5.49 The linked via hosts Redis connection	147
5.50 Starting the Redis-enabled Sinatra application	147
5.51 Testing our Redis-enabled Sinatra application	148
5.52 Confirming Redis contains data	148
5.53 Jenkins and Docker Dockerfile	151
5.54 Building our Docker-Jenkins image	153
5.55 Running our Docker-Jenkins image	153
5.56 Checking the Docker Jenkins container logs	154
5.57 Checking that Jenkins is up and running	154
5.58 The Docker shell script for Jenkins jobs	157
5.59 The Docker test job Dockerfile	158
5.60 Jenkins multi-configuration shell step	166

5.61 Our CentOS-based Dockerfile	167
6.1 Creating our Jekyll Dockerfile	173
6.2 Jekyll Dockerfile	174
6.3 Building our Jekyll image	175
6.4 Viewing our new Jekyll Base image	175
6.5 Creating our Apache Dockerfile	176
6.6 Jekyll Apache Dockerfile	176
6.7 Building our Jekyll Apache image	177
6.8 Viewing our new Jekyll Apache image	178
6.9 Getting a sample Jekyll blog	178
6.10 Creating a Jekyll container	179
6.11 Creating an Apache container	180
6.12 Resolving the Apache container's port	180
6.13 Editing our Jekyll blog	181
6.14 Restarting our <code>james_blog</code> container	181
6.15 Checking the <code>james_blog</code> container logs	182
6.16 Backing up the <code>/var/www/html</code> volume	183
6.17 Backup command	184
6.18 Creating our fetcher Dockerfile	185
6.19 Our war file fetcher	186
6.20 Building our fetcher image	186
6.21 Fetching a war file	187
6.22 Inspecting our Sample volume	187
6.23 Listing the volume directory	188
6.24 Creating our Tomcat 7 Dockerfile	188
6.25 Our Tomcat 7 Application server	189
6.26 Building our Tomcat 7 image	189
6.27 Creating our first Tomcat instance	190
6.28 Identifying the Tomcat application port	190
6.29 Installing Ruby	191
6.30 Installing the TProv application	191
6.31 Launching the TProv application	192
6.32 Creating our Node.js Dockerfile	195
6.33 Our Node.js image	196
6.34 Our Node.js <code>server.js</code> application	197

6.35 Building our Node.js image	198
6.36 Creating our Redis base Dockerfile	198
6.37 Our Redis base image	199
6.38 Building our Redis base image	199
6.39 Creating our Redis primary Dockerfile	200
6.40 Our Redis primary image	200
6.41 Building our Redis primary image	200
6.42 Creating our Redis replica Dockerfile	201
6.43 Our Redis replica image	201
6.44 Building our Redis replica image	201
6.45 Running the Redis primary container	202
6.46 Our Redis primary logs	202
6.47 Reading our Redis primary logs	203
6.48 Running our first Redis replica container	203
6.49 Reading our Redis replica logs	204
6.50 Running our second Redis replica container	205
6.51 Our Redis replica2 logs	206
6.52 Running our Node.js container	207
6.53 The nodeapp console log	207
6.54 Node application output	208
6.55 Creating our Logstash Dockerfile	208
6.56 Our Logstash image	209
6.57 Our Logstash configuration	210
6.58 Building our Logstash image	211
6.59 Launching a Logstash container	211
6.60 The logstash container's logs	211
6.61 A Node event in Logstash	212
6.62 Using docker kill to send signals	213
6.63 Installing nsenter	214
6.64 Finding the process ID of the container	214
6.65 Entering a container with nsenter	214
6.66 Running a command inside a container with nsenter	214
7.1 Installing Fig on Linux	218
7.2 Installing Fig on OS X	218
7.3 Installing Fig via Pip	218

7.4	Testing Fig is working	219
7.5	Creating the figapp directory	219
7.6	The app.py file	220
7.7	The requirements.txt file	220
7.8	The figapp Dockerfile	221
7.9	Building the figapp application	222
7.10	Creating the fig.yml file	223
7.11	The fig.yml file	224
7.12	The build instruction	224
7.13	The docker run equivalent command	225
7.14	Running fig up with our sample application	226
7.15	Fig service log output	227
7.16	Running Fig daemonized	227
7.17	Restarting Fig as daemonized	228
7.18	Running the fig ps command	229
7.19	Showing a Fig services logs	229
7.20	Stopping running services	229
7.21	Verifying our Fig services have been stopped	230
7.22	Removing Fig services	230
7.23	Showing no Fig services	230
7.24	Creating a Consul Dockerfile directory	233
7.25	The Consul Dockerfile	234
7.26	The consul.json configuration file	235
7.27	Building our Consul image	236
7.28	Running a local Consul node	237
7.29	Pulling down the Consul image	239
7.30	Assigning public IP on larry	239
7.31	Assigning public IP on curly and moe	240
7.32	Adding the cluster IP address	240
7.33	Getting the docker0 IP address	241
7.34	Original Docker defaults	241
7.35	New Docker defaults on larry	241
7.36	Restarting the Docker daemon on larry	242
7.37	Start the Consul bootstrap node	242
7.38	Consul agent command line arguments	243

7.39 Starting bootstrap Consul node	244
7.40 Cluster leader error	245
7.41 Starting the agent on curly	245
7.42 Launching the Consul agent on curly	245
7.43 Looking at the Curly agent logs	246
7.44 Curly joining Larry	246
7.45 Starting the agent on curly	247
7.46 Consul logs on moe	248
7.47 Consul leader election on larry	249
7.48 Testing the Consul DNS	251
7.49 Querying another Consul service via DNS	251
7.50 Creating a distributed_app Dockerfile directory	252
7.51 The distributed_app Dockerfile	253
7.52 The uWSGI configuration	254
7.53 The distributed_app config.ru file	254
7.54 The Consul plugin URL	255
7.55 Building our distributed_app image	255
7.56 Creating a distributed_client Dockerfile directory	255
7.57 The distributed_client Dockerfile	256
7.58 The distributed_client application	257
7.59 Building our distributed_client image	258
7.60 Starting distributed_app on larry	259
7.61 The distributed_app log output	259
7.62 Starting distributed_app on curly	260
7.63 Starting distributed_client on moe	261
7.64 The distributed_client logs on moe	262
8.1 Default systemd daemon start options	268
8.2 Network binding systemd daemon start options	268
8.3 Reloading and restarting the Docker daemon	268
8.4 Connecting to a remote Docker daemon	269
8.5 Revisiting the DOCKER_HOST environment variable	269
8.6 Using the info API endpoint	270
8.7 Getting a list of images via API	271
8.8 Getting a specific image	272
8.9 Searching for images with the API	273

8.10 Listing running containers	274
8.11 Listing all containers via the API	274
8.12 Creating a container via the API	275
8.13 Configuring container launch via the API	275
8.14 Starting a container via the API	276
8.15 API equivalent for docker run command	276
8.16 Listing all containers via the API	277
8.17 The legacy TProv container launch methods	278
8.18 The Docker Ruby client	279
8.19 Installing the Docker Ruby client API prerequisites	280
8.20 Testing our Docker API connection via irb	280
8.21 Our updated TProv container management methods	281
8.22 Checking for openssl	283
8.23 Create a CA directory	283
8.24 Generating a private key	283
8.25 Creating a CA certificate	284
8.26 Creating a server key	285
8.27 Creating our server CSR	286
8.28 Signing our CSR	287
8.29 Removing the passphrase from the server key	287
8.30 Securing the key and certificate on the Docker server	287
8.31 Enabling Docker TLS on systemd	288
8.32 Reloading and restarting the Docker daemon	288
8.33 Creating a client key	289
8.34 Creating a client CSR	290
8.35 Adding Client Authentication attributes	290
8.36 Signing our client CSR	291
8.37 Stripping out the client key pass phrase	291
8.38 Copying the key and certificate on the Docker client	292
8.39 Testing our TLS-authenticated connection	292
8.40 Testing our TLS-authenticated connection	293
9.1 Installing git on Ubuntu	297
9.2 Installing git on Red Hat et al	297
9.3 Check out the Docker source code	298
9.4 Building the Docker documentation	298

9.5 Building the Docker environment	299
9.6 Building the Docker binary	299
9.7 The Docker dev binary	299
9.8 Using the development daemon	300
9.9 Using the development binary	300
9.10 Running the Docker tests	300
9.11 Docker test output	301
9.12 Launching an interactive session	302
9.13 The Docker DCO	303
9.14 Git commit signing	304

Foreword

Who is this book for?

The Docker Book is for developers, sysadmins, and DevOps-minded folks who want to implement Docker™ and container-based virtualization.

There is an expectation that the reader has basic Linux/Unix skills and is familiar with the command line, editing files, installing packages, managing services, and basic networking.

NOTE This book focuses on Docker version 1.0.0 and later. It is not generally backwards-compatible with earlier releases. Indeed, it is recommended that for production purposes you use Docker version 1.0.0 or later.

Credits and Acknowledgments

- My partner and best friend, Ruth Brown, who continues to humor me despite my continuing to write books.
- The team at Docker Inc., for developing Docker and helping out during the writing of the book.
- The folks in the `#docker` channel and the Docker mailing list for helping out when I got stuck.

- Royce Gilbert for not only creating the amazing technical illustrations, but also the cover.
- Abhinav Ajgaonkar for his Node.js and Express example application.
- The technical review team for keeping me honest and pointing out all the stupid mistakes.

Images on pages 38, 45, 48, courtesy of Docker, Inc.

Docker™ is a registered trademark of Docker, Inc.

Technical Reviewers

Scott Collier

Scott Collier is a Senior Principal System Engineer for Red Hat's Systems Design and Engineering team. This team identifies and works on high-value solution stacks based on input from Sales, Marketing, and Engineering teams and develops reference architectures for consumption by internal and external customers. Scott is a Red Hat Certified Architect (RHCA) with more than 15 years of IT experience, currently focused on Docker, OpenShift, and other products in the Red Hat portfolio.

When he's not tinkering with distributed architectures, he can be found running, hiking, camping, and eating barbecue around the Austin, TX, area with his wife and three children. His notes on technology and other things can be found [here](#).

John Ferlito

John is a serial entrepreneur as well as an expert in highly available and scalable infrastructure. John is currently a founder and CTO of Bulletproof, who provide Mission Critical Cloud, and CTO of Vquence, a Video Metrics aggregator.

In his spare time, John is involved in the FOSS communities. He was a co-organizer of linux.conf.au 2007 and a committee member of SLUG in 2007,

and he has worked on various open-source projects, including Debian, Ubuntu, Puppet, and the Annodex suite. You can read more about John's work on his [blog](#). John has a Bachelor of Engineering (Computing) with Honors from the University of New South Wales.

Paul Nasrat

Paul Nasrat works as an SRE at Google and is a Docker contributor. He's worked on a variety of open source tools in the systems engineering space, including boot loaders, package management, and configuration management.

Paul has worked in a variety of Systems Administration and Software Development roles, including working as a Software Engineer at Red Hat and as an Infrastructure Specialist Consultant at ThoughtWorks. Paul has spoken at various conferences, from talking about Agile Infrastructure at Agile 2009 during the early days of the DevOps movement to smaller meetups and conferences.

Technical Illustrator

[Royce Gilbert](#) has over 30 years' experience in CAD design, computer support, network technologies, project management, and business systems analysis for major Fortune 500 companies, including Enron, Compaq, Koch Industries, and Amoco Corp. He is currently employed as a Systems/Business Analyst at Kansas State University in Manhattan, KS. In his spare time he does Freelance Art and Technical Illustration as sole proprietor of Royce Art. He and his wife of 38 years are living in and restoring a 127-year-old stone house nestled in the Flinthills of Kansas.

Proofreader

Q grew up in the New York area and has been a high school teacher, cupcake icer, scientist wrangler, forensic anthropologist, and catastrophic disaster response

planner. She now resides in San Francisco, making music, acting, putting together newsletter, and taking care of the fine folks at Stripe.

Author

James is an author and open source geek. His most recent book was [The LogStash Book](#) about the popular open source logging tool. James also authored two books about Puppet ([Pro Puppet](#) and the [earlier book](#) about Puppet). He is the author of three other books, including [Pro Linux System Administration](#), [Pro Nagios 2.0](#), and [Hardening Linux](#).

For a real job, James is VP of Engineering at Kickstarter. He was formerly at Docker as VP of Services and Support, Venmo as VP of Engineering and Puppet Labs as VP of Technical Operations. He likes food, wine, books, photography, and cats. He is not overly keen on long walks on the beach and holding hands.

Conventions in the book

This is an `inline` code statement.

This is a code block:

Listing 1: Sample code block

```
This is a code block
```

Long code strings are broken with `←`.

Code and Examples

You can find all the code and examples from the book on the [website](#), or you can check out the [Git repo](#).

Colophon

This book was written in Markdown with a large dollop of LaTeX. It was then converted to PDF and other formats using PanDoc (with some help from scripts written by the excellent folks who wrote [Backbone.js on Rails](#)).

Errata

Please email any errata you find [here](#).

Version

This is version v1.2.2 (19946fc) of The Docker Book.

Chapter 1

Introduction

Containers have a long and storied history in computing. Unlike hypervisor virtualization, where one or more independent machines run virtually on physical hardware via an intermediation layer, containers instead run user space on top of an operating system's kernel. As a result, container virtualization is often called operating system-level virtualization. Container technology allows multiple isolated user space instances to be run on a single host.

As a result of their status as guests of the operating system, containers are sometimes seen as less flexible: they can generally only run the same or a similar guest operating system as the underlying host. For example, you can run Red Hat Enterprise Linux on an Ubuntu server, but you can't run Microsoft Windows on top of an Ubuntu server.

Containers have also been seen as less secure than the full isolation of hypervisor virtualization. Countering this argument is that lightweight containers lack the larger attack surface of the full operating system needed by a virtual machine combined with the potential exposures of the hypervisor layer itself.

Despite these limitations, containers have been deployed in a variety of use cases. They are popular for hyperscale deployments of multi-tenant services, for lightweight sandboxing, and, despite concerns about their security, as process isolation environments. Indeed, one of the more common examples of a container is a chroot jail, which creates an isolated directory environment for running

processes. Attackers, if they breach the running process in the jail, then find themselves trapped in this environment and unable to further compromise a host.

More recent container technologies have included [OpenVZ](#), Solaris Zones, and Linux containers like [lxc](#). Using these more recent technologies, containers can now look like full-blown hosts in their own right rather than just execution environments. In Docker's case, having modern Linux kernel features, such as control groups and namespaces, means that containers can have strong isolation, their own network and storage stacks, as well as resource management capabilities to allow friendly co-existence of multiple containers on a host.

Containers are generally considered a lean technology because they require limited overhead. Unlike traditional virtualization or paravirtualization technologies, they do not require an emulation layer or a hypervisor layer to run and instead use the operating system's normal system call interface. This reduces the overhead required to run containers and can allow a greater density of containers to run on a host.

Despite their history containers haven't achieved large-scale adoption. A large part of this can be laid at the feet of their complexity: containers can be complex, hard to set up, and difficult to manage and automate. Docker aims to change that.

Introducing Docker

Docker is an open-source engine that automates the deployment of applications into containers. It was written by the team at [Docker, Inc](#) (formerly dotCloud Inc, an early player in the Platform-as-a-Service (PaaS) market), and released by them under the Apache 2.0 license.

NOTE Disclaimer and disclosure: I work at Docker.

So what is special about Docker? Docker adds an application deployment engine on top of a virtualized container execution environment. It is designed to provide

a lightweight and fast environment in which to run your code as well as an efficient workflow to get that code from your laptop to your test environment and then into production. Docker is incredibly simple. Indeed, you can get started with Docker on a minimal host running nothing but a compatible Linux kernel and a Docker binary. Docker's mission is to provide:

An easy and lightweight way to model reality

Docker is fast. You can Dockerize your application in minutes. Docker relies on a copy-on-write model so that making changes to your application is also incredibly fast: only what you want to change gets changed.

You can then create containers running your applications. **Most Docker containers take less than a second to launch.** Removing the overhead of the hypervisor also means containers are highly performant and you can pack more of them into your hosts and make the best possible use of your resources.

A logical segregation of duties

With Docker, Developers care about their applications running inside containers, and Operations cares about managing the containers. Docker is designed to enhance consistency by ensuring the environment in which your developers write code matches the environments into which your applications are deployed. This reduces the risk of "worked in dev, now an ops problem."

Fast, efficient development life cycle

Docker aims to reduce the cycle time between code being written and code being tested, deployed, and used. It aims to make your applications portable, easy to build, and easy to collaborate on.

Encourages service orientated architecture

Docker also encourages service orientated and [microservices](#) architectures. Docker recommends that each container run a single application or process. This promotes a distributed application model where an application or service is represented by a series of inter-connected containers. This makes it very easy to distribute, scale, debug and introspect your applications.

NOTE Although you don't need to build your applications this way if you don't wish. You can easily run a multi-processes application inside a single container.

Docker components

Let's look at the core components that compose Docker:

- The Docker client and server
- Docker Images
- Registries
- Docker Containers

Docker client and server

Docker is a client-server application. The Docker client talks to the Docker server or daemon, which, in turn, does all the work. Docker ships with a command line client binary, `docker`, as well as a [full RESTful API](#). You can run the Docker daemon and client on the same host or connect your local Docker client to a remote daemon running on another host. You can see Docker's architecture depicted here:

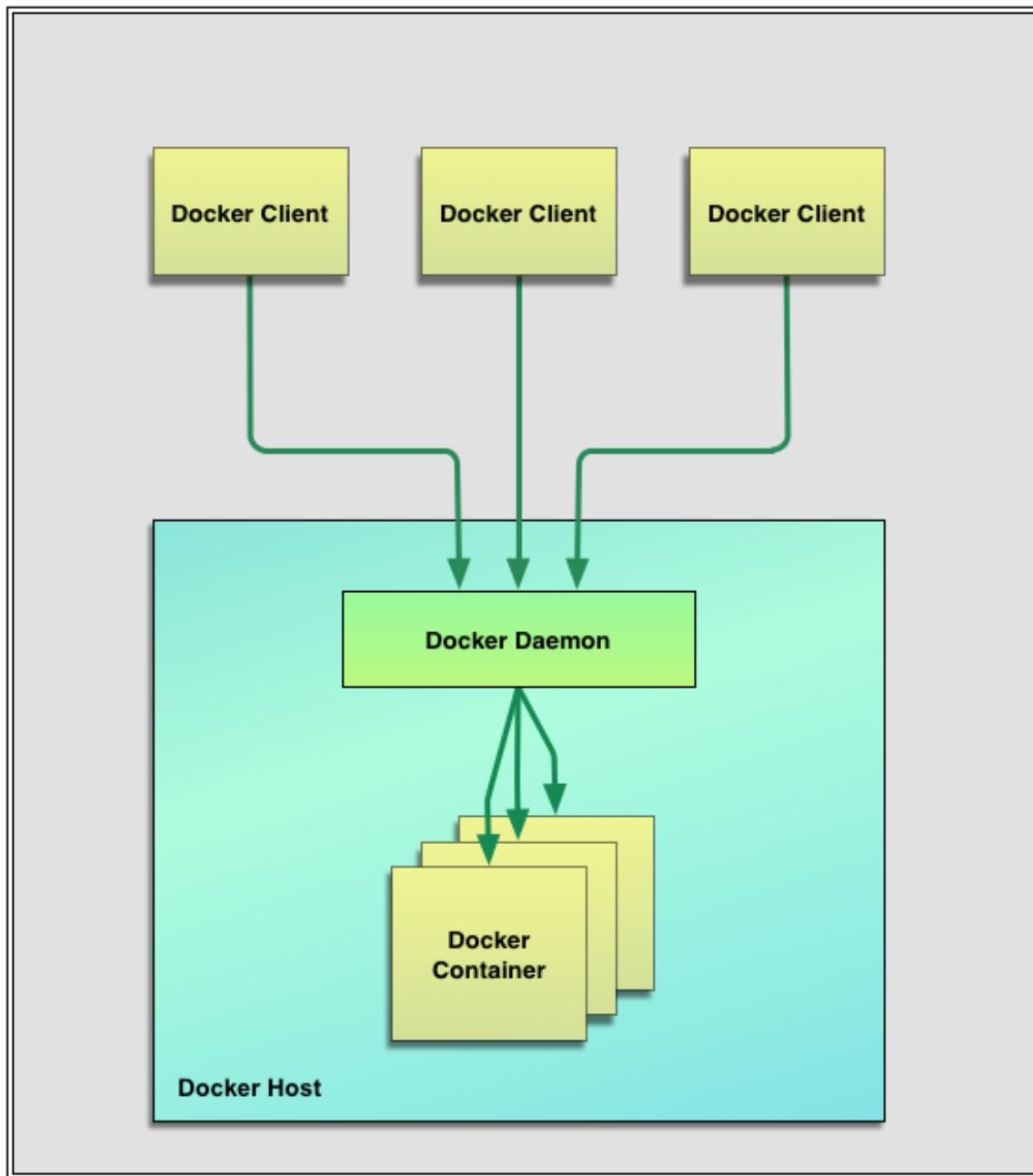


Figure 1.1: Docker architecture

Docker images

Images are the building blocks of the Docker world. You launch your containers from images. Images are the "build" part of Docker's life cycle. They are a layered format, using Union file systems, that are built step-by-step using a series of instructions. For example:

- Add a file.
- Run a command.
- Open a port.

You can consider images to be the "source code" for your containers. They are highly portable and can be shared, stored, and updated. In the book, we'll learn how to use existing images as well as build our own images.

Registries

Docker stores the images you build in registries. There are two types of registries: public and private. Docker, Inc., operates the public registry for images, called the [Docker Hub](#). You can [create an account](#) on the Docker Hub and use it to share and store your own images.

The Docker Hub also contains, at last count, over 10,000 images that other people have built and shared. Want a Docker image for [an Nginx web server](#), the [Asterisk open source PABX system](#), or a [MySQL database](#)? All of these are available, along with a whole lot more.

You can also store images that you want to keep private on the Docker Hub. These images might include source code or other proprietary information you want to keep secure or only share with other members of your team or organization.

You can also run your own private registry, and we'll show you how to do that in Chapter 4. This allows you to store images behind your firewall, which may be a requirement for some organizations.

Containers

Docker helps you build and deploy containers inside of which you can package your applications and services. As we've just learned, containers are launched from images and can contain one or more running processes. You can think about images as the building or packing aspect of Docker and the containers as the running or execution aspect of Docker.

A Docker container is:

- An image format.
- A set of standard operations.
- An execution environment.

Docker borrows the concept of the standard shipping container, used to transport goods globally, as a model for its containers. But instead of shipping goods, Docker containers ship software.

Each container contains a software image -- its 'cargo' -- and, like its physical counterpart, allows a set of operations to be performed. For example, it can be created, started, stopped, restarted, and destroyed.

Like a shipping container, Docker doesn't care about the contents of the container when performing these actions; for example, whether a container is a web server, a database, or an application server. Each container is loaded the same as any other container.

Docker also doesn't care where you ship your container: you can build on your laptop, upload to a registry, then download to a physical or virtual server, test, deploy to a cluster of a dozen Amazon EC2 hosts, and run. Like a normal shipping container, it is interchangeable, stackable, portable, and as generic as possible.

With Docker, we can quickly build an application server, a message bus, a utility appliance, a CI test bed for an application, or one of a thousand other possible applications, services, and tools. It can build local, self-contained test environments or replicate complex application stacks for production or development purposes. The possible use cases are endless.

What can you use Docker for?

So why should you care about Docker or containers in general? We've discussed briefly the isolation that containers provide; as a result, they make excellent sandboxes for a variety of testing purposes. Additionally, because of their 'standard' nature, they also make excellent building blocks for services. Some of the examples of Docker running out in the wild include:

- Helping make your local development and build workflow faster, more efficient, and more lightweight. Local developers can build, run, and share Docker containers. Containers can be built in development and promoted to testing environments and, in turn, to production.
- Running stand-alone services and applications consistently across multiple environments, a concept especially useful in service-oriented architectures and deployments that rely heavily on micro-services.
- Using Docker to create isolated instances to run tests like, for example, those launched by a Continuous Integration (CI) suite like Jenkins CI.
- Building and testing complex applications and architectures on a local host prior to deployment into a production environment.
- Building a multi-user Platform-as-a-Service (PAAS) infrastructure.
- Providing lightweight stand-alone sandbox environments for developing, testing, and teaching technologies, such as the Unix shell or a programming language.
- Software as a Service applications; for example, [Memcached as a service](#).
- Highly performant, hyperscale deployments of hosts.

You can see a list of some of the early projects built on and around the Docker ecosystem [here](#).

Docker with configuration management

Since Docker was announced, there have been a lot of discussions about where Docker fits with configuration management tools like Puppet and Chef. Docker

includes an image-building and image-management solution. One of the drivers for modern configuration management tools was the response to [the "golden image" model](#). With golden images, you end up with massive and unmanageable image sprawl: large numbers of (deployed) complex images in varying states of versioning. You create randomness and exacerbate entropy in your environment as your image use grows. Images also tend to be heavy and unwieldy. This often forces manual change or layers of deviation and unmanaged configuration on top of images, because the underlying images lack appropriate flexibility.

Compared to traditional image models, Docker is a lot more lightweight: images are layered, and you can quickly iterate on them. There is some legitimate argument to suggest that these attributes alleviate many of the management problems traditional images present. It is not immediately clear, though, that this alleviation represents the ability to totally replace or supplant configuration management tools. There is amazing power and control to be gained through the idempotence and introspection that configuration management tools can provide. Docker itself still needs to be installed, managed, and deployed on a host. That host also needs to be managed. In turn, Docker containers may need to be orchestrated, managed, and deployed, often in conjunction with external services and tools, which are all capabilities that configuration management tools are excellent in providing.

It is also apparent that Docker represents (or, perhaps more accurately, encourages) some different characteristics and architecture for hosts, applications, and services: they can be short-lived, immutable, disposable, and service-oriented. These behaviors do not lend themselves or resonate strongly with the need for configuration management tools. With these behaviors, you are rarely concerned with long-term management of state, entropy is less of a concern because containers rarely live long enough for it to be, and the recreation of state may often be cheaper than the remediation of state.

Not all infrastructure can be represented with these behaviors, however. Docker's ideal workloads will likely exist alongside more traditional infrastructure deployment for a little while. The long-lived host, perhaps also the host that needs to run on physical hardware, still has a role in many organizations. As a result of these diverse management needs, combined with the need to manage Docker itself, both Docker and configuration management tools are likely to be deployed

in the majority of organizations.

Docker's technical components

Docker can be run on any x64 host running a modern Linux kernel; we recommend kernel version 3.8 and later. It has low overhead and can be used on servers, desktops, or laptops. It includes:

- A native Linux container format that Docker calls `libcontainer`, as well as the popular container platform, [lxc](#). The `libcontainer` format is now the default format.
- [Linux kernel namespaces](#), which provide isolation for filesystems, processes, and networks.
- Filesystem isolation: each container is its own root filesystem.
- Process isolation: each container runs in its own process environment.
- Network isolation: separate virtual interfaces and IP addressing between containers.
- Resource isolation and grouping: resources like CPU and memory are allocated individually to each Docker container using the [cgroups](#), or control groups, kernel feature.
- [Copy-on-write](#): filesystems are created with copy-on-write, meaning they are layered and fast and require limited disk usage.
- Logging: `STDOUT`, `STDERR` and `STDIN` from the container are collected, logged, and available for analysis or trouble-shooting.
- Interactive shell: You can create a pseudo-tty and attach to `STDIN` to provide an interactive shell to your container.

What's in the book?

In this book, we walk you through installing, deploying, managing, and extending Docker. We do that by first introducing you to the basics of Docker and its com-

ponents. Then we start to use Docker to build containers and services to perform a variety of tasks.

We take you through the development life cycle, from testing to production, and see where Docker fits in and how it can make your life easier. We make use of Docker to build test environments for new projects, demonstrate how to integrate Docker with continuous integration workflow, and then how to build application services and platforms. Finally, we show you how to use Docker's API and how to extend Docker yourself.

We teach you how to:

- Install Docker.
- Take your first steps with a Docker container.
- Build Docker images.
- Manage and share Docker images.
- Run and manage more complex Docker containers.
- Deploy Docker containers as part of your testing pipeline.
- Build multi-container applications and environments.
- Introduce the basics of Docker orchestration with Fig.
- Explore the Docker API.
- Getting Help and Extending Docker.

It is recommended that you read through every chapter. Each chapter builds on your Docker knowledge and introduce new features and functionality. By the end of the book you should have a solid understanding of how to work with Docker to build standard containers and deploy applications, test environments, and standalone services.

Docker resources

- [Docker homepage](#)
- [Docker Hub](#)
- [Docker blog](#)

- [Docker documentation](#)
- [Docker Getting Started Guide](#)
- [Docker code on GitHub](#)
- [Docker Forge](#) - collection of Docker tools, utilities, and services.
- [Docker mailing list](#)
- Docker on IRC: irc.freenode.net and channel #docker
- [Docker on Twitter](#)
- Get [Docker help](#) on StackOverflow
- [Docker.com](#)

In addition to these resources in Chapter 9 you'll get a detailed explanation of where and how to get help with Docker.

Chapter 2

Installing Docker

Installing Docker is quick and easy. Docker is currently supported on a wide variety of Linux platforms, including shipping as part of Ubuntu and Red Hat Enterprise Linux (RHEL). Also supported are various derivative and related distributions like Debian, CentOS, Fedora, Oracle Linux, and many others. Using a virtual environment, you can install and run Docker on OS X and Microsoft Windows.

Currently, the Docker team recommends deploying Docker on Ubuntu or RHEL hosts and makes available packages that you can use to do this. In this chapter, I'm going to show you how to install Docker in four different but complementary environments:

- On a host running Ubuntu.
- On a host running Red Hat Enterprise Linux or derivative distribution.
- On OS X using [Boot2Docker](#).
- On Microsoft Windows using [Boot2Docker](#).

TIP Boot2Docker is a tiny virtual machine shipped with a wrapper script to manage it. The virtual machine runs the daemon and provides a local Docker daemon on OS X and Microsoft Windows. The Docker client binary, `docker`, can be installed natively on these platforms and connected to the Docker daemon

running in the virtual machine.

Docker runs on a number of other platforms, including Debian, [SuSE](#), [Arch Linux](#), CentOS, and [Gentoo](#). It's also supported on several Cloud platforms including [Amazon EC2](#), [Rackspace Cloud](#), and [Google Compute Engine](#).

I've chosen these four methods because they represent the environments that are most commonly used in the Docker community. For example, your developers and sysadmins may wish to start with building Docker containers on their OS X or Windows workstations using Boot2Docker and then promote these containers to a testing, staging, or production environment running one of the other supported platforms.

I recommend you step through at least the Ubuntu or the RHEL installation to get an idea of Docker's prerequisites and an understanding of how to install it.

TIP As with all installation processes, I also recommend you look at using tools like [Puppet](#) or [Chef](#) to install Docker rather than using a manual process. For example, you can find a Puppet module to install Docker [here](#) and a Chef cookbook [here](#).

Requirements

For all of these installation types, Docker has some basic prerequisites. To use Docker you must:

- Be running a 64-bit architecture (currently x86_64 and amd64 only). 32-bit is **NOT** currently supported.
- Be running a Linux 3.8 or later kernel. Some earlier kernels from 2.6.x and later will run Docker successfully. Your results will greatly vary, though, and if you need support you will often be asked to run on a more recent

kernel.

- The kernel must support an appropriate storage driver. For example,
 - [Device Mapper](#)
 - [AUFS](#)
 - [vfs](#).
 - [btrfs](#)
 - The default storage driver is usually Device Mapper.
- [cgroups](#) and [namespaces](#) kernel features must be supported and enabled.

Installing on Ubuntu

Installing Docker on Ubuntu is currently officially supported on a selection of Ubuntu releases:

- Ubuntu Trusty 14.04 (LTS) (64-bit)
- Ubuntu Precise 12.04 (LTS) (64-bit)
- Ubuntu Raring 13.04 (64-bit)
- Ubuntu Saucy 13.10 (64-bit)

NOTE This is not to say Docker won't work on other Ubuntu (or Debian) versions that have appropriate kernel levels and the additional required support. They just aren't officially supported, so any bugs you encounter may result in a WONTFIX.

To begin our installation, we first need to confirm we've got all the required prerequisites. I've created a brand new Ubuntu 12.04 LTS 64-bit host on which to install Docker. We're going to call that host `darknight.example.com`.

Checking for prerequisites

Docker has a small but necessary list of prerequisites required to install and run on Ubuntu hosts.

Kernel

First, let's confirm we've got a sufficiently recent Linux kernel. We can do this using the `uname` command.

Listing 2.1: Checking for the Linux kernel version on Ubuntu

```
$ uname -a
Linux darknight.example.com 3.8.0-23-generic #34~precise1-Ubuntu ~
SMP Wed May 29 21:12:31 UTC 2013 x86_64 x86_64 x86_64 GNU/Linux
```

We can see that we've got a 3.8.0 x86_64 kernel installed. This is the default for Ubuntu 12.04.3 and later (and also for Ubuntu 13.04 Raring).

If, however, we're using an earlier release of Ubuntu 12.04 Precise, we may have a 3.2 kernel. We can easily upgrade our Ubuntu 12.04 to the later kernel; for example, at the time of writing, the 3.8 kernel was available to install via `apt-get`:

Listing 2.2: Installing a 3.8 kernel on Ubuntu Precise

```
$ sudo apt-get update
$ sudo apt-get install linux-headers-3.8.0-27-generic linux-image-3.8.0-27-generic linux-headers-3.8.0-27
```

NOTE Throughout this book we're going to use `sudo` to provide the required root privileges.

We can then update the Grub boot loader to load our new kernel.

Listing 2.3: Updating the boot loader on Ubuntu Precise

```
$ sudo update-grub
```

After installation, we'll need to reboot our host to enable the new 3.8 kernel.

Listing 2.4: Reboot the Ubuntu host

```
$ sudo reboot
```

After the reboot, we can then check that our host is running the right version using the same `uname -a` command we used above.

NOTE Remember: If installing on Ubuntu Raring, you won't need to update the kernel, as it already comes with a 3.8 kernel.

Checking for Device Mapper

We're going to make use of the Device Mapper storage driver. The Device Mapper framework has been in the Linux kernel since 2.6.9 and provides a method for mapping block devices into higher-level virtual devices. It supports a concept called '[thin-provisioning](#)' to store multiple virtual devices, the layers in our Docker images, on a filesystem. Hence, it is perfect for providing the storage that Docker requires.

Device Mapper should be installed on any Ubuntu 12.04 or later hosts, but we can confirm it is installed like so:

Listing 2.5: Checking for Device Mapper

```
$ ls -l /sys/class/misc/device-mapper
lrwxrwxrwx 1 root root 0 Oct  5 18:50 /sys/class/misc/device-mapper -> ../../devices/virtual/misc/device-mapper
```

We could also check in `/proc/devices` for a `device-mapper` entry.

Listing 2.6: Checking for Device Mapper in proc on Ubuntu

```
$ sudo grep device-mapper /proc/devices
```

If neither is present, we can also try to load the `dm_mod` module.

Listing 2.7: Loading the Device Mapper module

```
$ sudo modprobe dm_mod
```

Both cgroups and namespaces have also been longtime Linux kernel residents since the 2.6 version. Both are generally well supported and relatively bug free since about the 2.6.38 release of the kernel.

Installing Docker

Now we've got everything we need to add Docker to our host. To install Docker, we're going to use the Docker team's DEB packages.

First, we add the Docker APT repository. You may be prompted to confirm that you wish to add the repository and have the repositories GPG automatically added to your host.

Listing 2.8: Adding the Docker APT repository

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

First, we'll need to ensure the `curl` command is installed.

Listing 2.9: Testing for curl installation

```
$ whereis curl
curl: /usr/bin/curl /usr/bin/X11/curl /usr/share/man/man1/curl.1.gz
```

Then install curl if it's not found.

Listing 2.10: Install curl if needed

```
$ sudo apt-get -y install curl
```

Next, we need to add the Docker repository's GPG key.

Listing 2.11: Adding the Docker repository GPG key

```
$ curl -s https://get.docker.io/gpg | sudo apt-key add -
```

Now, we update our APT sources.

Listing 2.12: Updating APT sources

```
$ sudo apt-get update
```

We can now install the Docker package itself.

Listing 2.13: Installing the Docker packages on Ubuntu

```
$ sudo apt-get install lxc-docker
```

This will install Docker and a number of additional required packages.

We should now be able to confirm that Docker is installed and running using the `docker info` command.

Listing 2.14: Checking Docker is installed on Ubuntu

```
$ sudo docker info
Containers: 0
Images: 0
...
```

Docker and UFW

If you use the [UFW](#), or Uncomplicated Firewall, on Ubuntu, then you'll need to make a small change to get it to work with Docker. Docker uses a network bridge

to manage the networking on your containers. By default, UFW drops all forwarded packets. You'll need to enable forwarding in UFW for Docker to function correctly. We can do this by editing the `/etc/default/ufw` file. Inside this file, change:

Listing 2.15: Old UFW forwarding policy

```
DEFAULT_FORWARD_POLICY="DROP"
```

To:

Listing 2.16: New UFW forwarding policy

```
DEFAULT_FORWARD_POLICY="ACCEPT"
```

Save the update and reload UFW.

Listing 2.17: Reload the UFW firewall

```
$ sudo ufw reload
```

Installing on Red Hat and family

Installing Docker on Red Hat Enterprise Linux (or CentOS or Fedora) is currently only supported on a small selection of releases:

- Red Hat Enterprise Linux (and CentOS) 6 and later (64-bit)
- Fedora Core 19 and later (64-bit)
- Oracle Linux 6 and 7 with Unbreakable Enterprise Kernel Release 3 (3.8.13) or higher (64-bit)

TIP Docker is shipped by Red Hat as a native package on Red Hat Enterprise Linux 7 and later. Additionally, Red Hat Enterprise Linux 7 is the only release on which Red Hat officially supports Docker.

Checking for prerequisites

Docker has a small but necessary list of prerequisites required to install and run on Red Hat and the Red Hat-family of distributions.

Kernel

We need to confirm that we have a 3.8 or later kernel version. We can do this using the `uname` command like so:

Listing 2.18: Checking the Red Hat or Fedora kernel

```
$ uname -a
Linux darknight.example.com 3.10.9-200.fc19.x86_64 #1 SMP Wed Aug 21 19:27:58 UTC 2013 x86_64 x86_64 x86_64 GNU/Linux
```

All of the currently supported Red Hat and Red Hat-family platforms should have a kernel that supports Docker.

Checking for Device Mapper

We're going to use the Device Mapper storage driver to provide Docker's storage capabilities. Device Mapper should be installed on any Red Hat Enterprise Linux, CentOS 6+, or Fedora Core 19 or later hosts, but we can confirm it is installed like so:

Listing 2.19: Checking for Device Mapper

```
$ ls -l /sys/class/misc/device-mapper
lrwxrwxrwx 1 root root 0 Oct  5 18:50 /sys/class/misc/device-mapper -> ../../devices/virtual/misc/device-mapper
```

We could also check in `/proc/devices` for a `device-mapper` entry.

Listing 2.20: Checking for Device Mapper in proc on Red Hat

```
$ sudo grep device-mapper /proc/devices
```

If neither is present, we can also try to install the `device-mapper` package.

Listing 2.21: Installing the Device Mapper package

```
$ sudo yum install -y device-mapper
```

Then we can load the `dm_mod` kernel module.

Listing 2.22: Loading the Device Mapper module

```
$ sudo modprobe dm_mod
```

We should now be able to find the `/sys/class/misc/device-mapper` entry.

Installing Docker

The process for installing differs slightly between Red Hat variants. On Red Hat Enterprise Linux 6 and CentOS 6, we will need to add the EPEL package repositories first. On Fedora, we do not need the EPEL repositories enabled. There are also some package-naming differences between platforms and versions.

Installing on Red Hat Enterprise Linux 6 and CentOS 6

For Red Hat Enterprise Linux 6 and CentOS 6, we install EPEL by adding the following RPM.

Listing 2.23: Installing EPEL on Red Hat Enterprise Linux 6 and CentOS 6

```
$ sudo rpm -Uvh http://download.fedoraproject.org/pub/epel/6/i386<--  
/epel-release-6-8.noarch.rpm
```

Now we should be able to install the Docker package.

Listing 2.24: Installing the Docker package on Red Hat Enterprise Linux 6 and CentOS 6

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

Installing on Red Hat Enterprise Linux 7

With Red Hat Enterprise Linux 7 and later you can install Docker using [these instructions](#).

Listing 2.25: Installing Docker on RHEL 7

```
$ sudo subscription-manager repos --enable=rhel-7-server-extras-←
  rpms
$ sudo yum install -y docker
```

You'll need to be a Red Hat customer with an appropriate RHEL Server subscription entitlement to access the Red Hat Docker packages and documentation.

Installing on Fedora

There have been some package name changes across versions of Fedora. For Fedora 19, we need to install the `docker-io` package.

Listing 2.26: Installing the Docker package on Fedora 19

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

On Fedora 20 and later, the package has been renamed to `docker`.

Listing 2.27: Installing the Docker package on Fedora 20 and later

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

Starting the Docker daemon on the Red Hat family

Once the package is installed, we can start the Docker daemon. On Red Hat Enterprise Linux 6 and CentOS 6 you can use.

Listing 2.28: Starting the Docker daemon on Red Hat 6

```
$ sudo service docker start
```

If we want Docker to start at boot we should also:

Listing 2.29: Ensuring the Docker daemon starts at boot on Red Hat 6

```
$ sudo service docker enable
```

On Red Hat Enterprise Linux 7 and Fedora.

Listing 2.30: Starting the Docker daemon on Red Hat 7

```
$ sudo systemctl start docker
```

If we want Docker to start at boot we should also:

Listing 2.31: Ensuring the Docker daemon starts at boot on Red Hat 7

```
$ sudo systemctl enable docker
```

We should now be able to confirm Docker is installed and running using the `docker info` command.

Listing 2.32: Checking Docker is installed on the Red Hat family

```
$ sudo docker info
Containers: 0
Images: 0
.
.
```

Boot2Docker installation on OS X

If you're using OS X, you can quickly get started with Docker using the [Boot2Docker tool](#). Boot2Docker is a tiny virtual machine with a supporting command line tool that is installed on an OS X host and provides you with a Docker environment.

Boot2Docker ships with a couple of prerequisites too:

- VirtualBox.
- The Docker client.

Installing Boot2Docker on OS X

To install Boot2Docker on OS X we need to download its installer from Git Hub. You can find it [here](#).

Let's grab the current release:

Listing 2.33: Downloading the Boot2Docker PKG file

```
$ wget https://github.com/boot2docker/osx-installer/releases/←  
download/v1.2.0/Boot2Docker-1.2.0.pkg
```

Launch the downloaded installer and follow the instructions to install Boot2Docker.

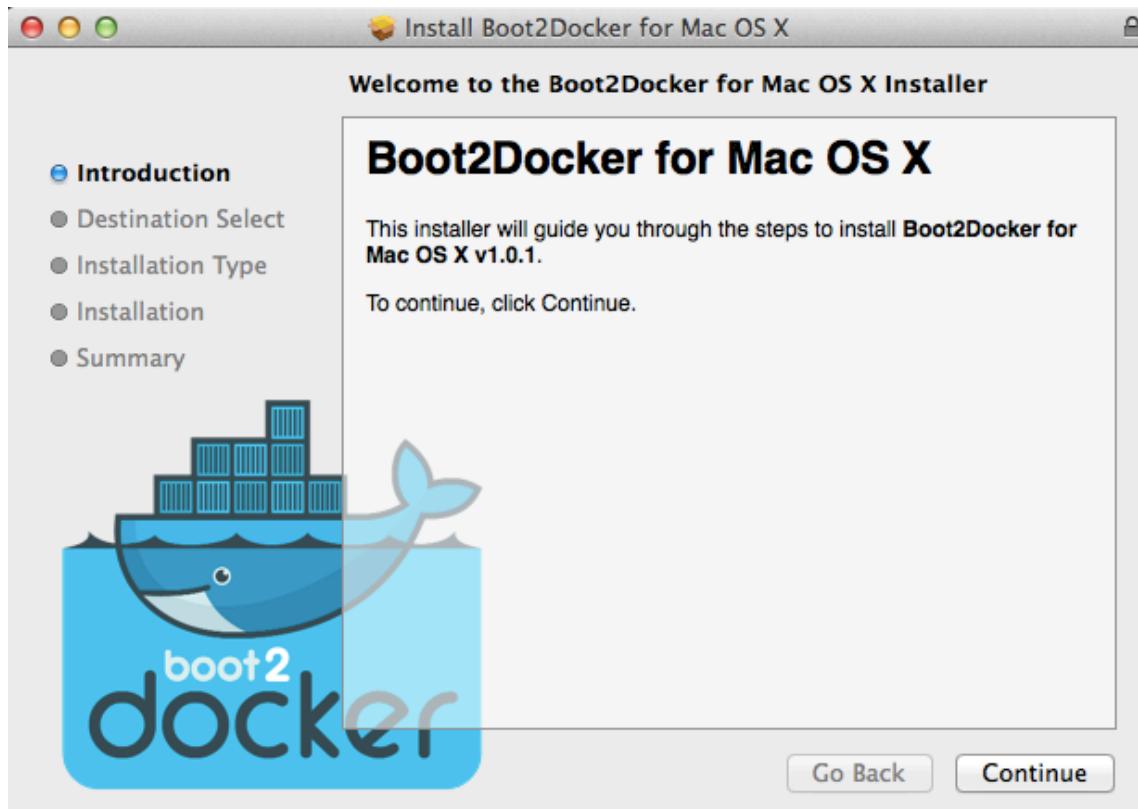


Figure 2.1: Installing Boot2Docker on OS X

Setting up Boot2Docker on OS X

Now that we've got all the pieces of Boot2Docker and its prerequisites installed, we can set it up and test it. To set it up, we run the Boot2Docker application.

Navigate to your OS X Application folder and click on the Boot2Docker icon to initialize and launch the Boot2Docker virtual machine.

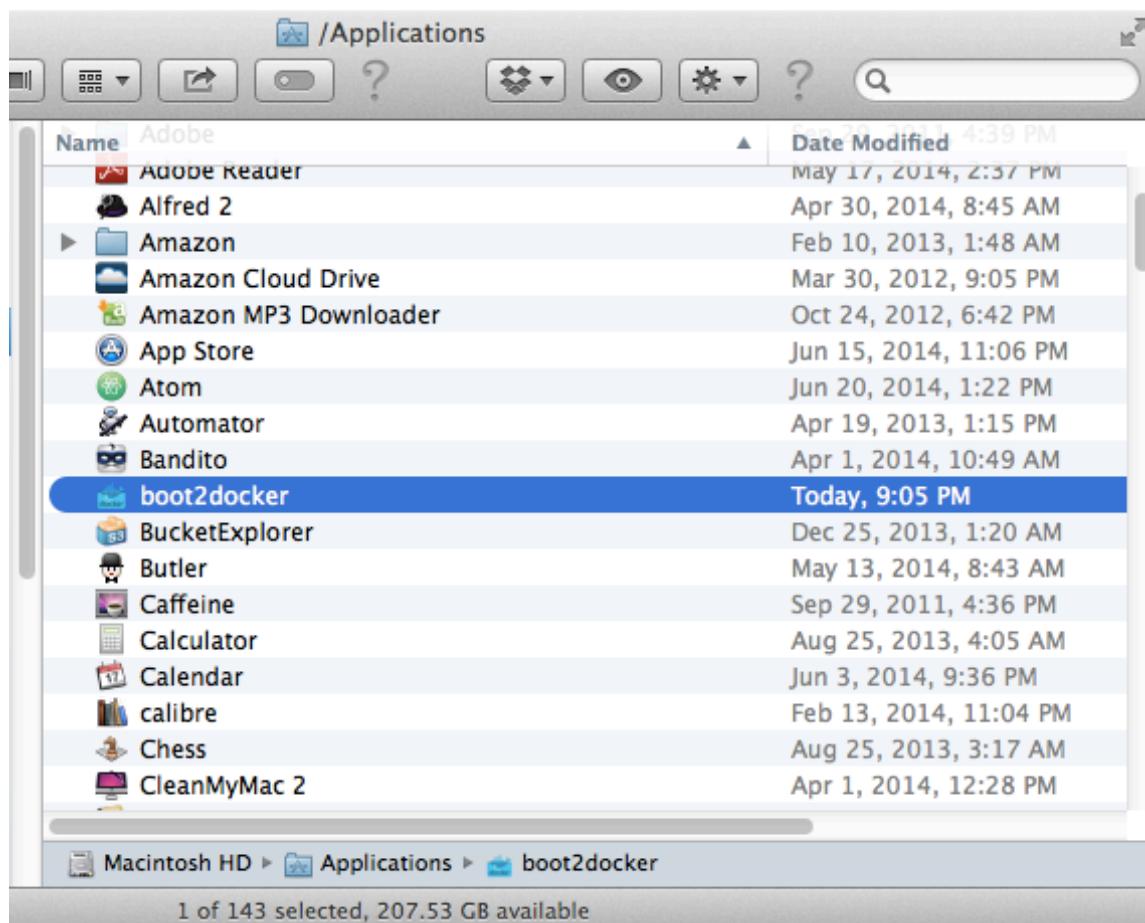


Figure 2.2: Running Boot2Docker on OS X

Testing Boot2Docker

We can now test that our Boot2Docker installation is working by trying to connect our local client to the Docker daemon running on the Boot2Docker virtual machine.

Listing 2.34: Testing Boot2Docker on OS X

```
$ docker info
Containers: 0
Images: 0
Driver: aufs
Root Dir: /mnt/sda1/var/lib/docker/aufs
Dirs: 0
.
.
.
Kernel Version: 3.13.3-tinycore64
```

And presto! We have Docker running locally on our OS X host.

Boot2Docker installation on Windows

If you're using Microsoft Windows, you can quickly get started with Docker using the [Boot2Docker tool](#). Boot2Docker is a tiny virtual machine with a supporting command line tool that is installed on a Windows host and provides you with a Docker environment.

Boot2Docker ships with a couple of prerequisites too:

- VirtualBox.
- The Docker client.

Installing Boot2Docker on Windows

To install Boot2Docker on Windows we need to download its installer from Git Hub. You can find it [here](#).

Let's grab the current release:

Listing 2.35: Downloading the Boot2Docker .EXE file

```
$ wget https://github.com/boot2docker/windows-installer/releases/←  
download/v1.2.0/docker-install.exe
```

Launch the downloaded installer and follow the instructions to install Boot2Docker.



Figure 2.3: Installing Boot2Docker on Windows

Setting up Boot2Docker on Windows

Once Boot2Docker is installed you can run the Boot2Docker Start script from the Desktop or Program Files > Boot2Docker for Windows.

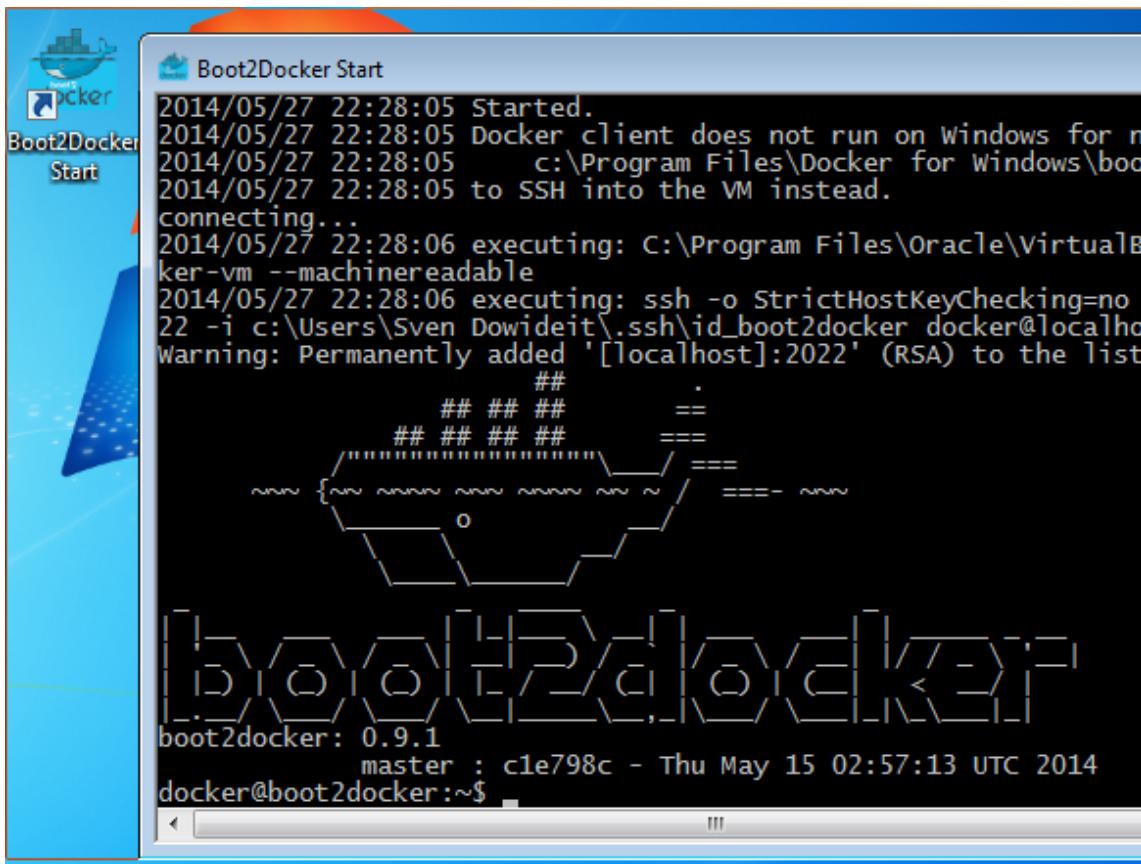


Figure 2.4: Running Boot2Docker on Windows

Testing Boot2Docker

We can now test that our Boot2Docker installation is working by trying to connect our local client to the Docker daemon running on the Boot2Docker virtual machine.

Listing 2.36: Testing Boot2Docker on Windows

```
$ docker info
Containers: 0
Images: 0
Driver: aufs
  Root Dir: /mnt/sda1/var/lib/docker/aufs
  Dirs: 0
...
Kernel Version: 3.13.3-tinycore64
```

And presto! We have Docker running locally on our Windows host.

Using Boot2Docker with this book

If you are following the examples in this book you will sometimes be asked to connect to a container via its network interface or a network port. Often this will be on the `localhost` or IP address of the Docker server. As Boot2Docker is a local virtual machine it has its own network interface and IP address. You will need to connect to that address rather than your `localhost` or host's own IP address.

To find the Boot2Docker IP address you can check the value of the `DOCKER_HOST` environment variable. You'll be prompted to set this variable when you start or install Boot2Docker with a message similar to:

Listing 2.37: Boot2Docker launch message

```
$ boot2docker start
2014/07/31 05:59:40 Waiting for VM to be started...
2014/07/31 05:59:46 Started.
2014/07/31 05:59:46 To connect the Docker client to the Docker ←
  daemon, please set:
2014/07/31 05:59:46      export DOCKER_HOST=tcp←
      ://192.168.59.103:2375
```

Or you can find the IP address by running the `boot2docker ip` command.

Listing 2.38: Getting the Boot2Docker IP address

```
$ boot2docker ip
The VM's Host only interface IP address is: 192.168.59.103
```

So with an example asking you to connect to a container on the `localhost` , for example using the `curl` command, you would replace `localhost` with the IP address provided.

So that:

Listing 2.39: Initial curl command

```
$ curl localhost:49155
```

Would become:

Listing 2.40: Updated curl command

```
$ curl 192.168.59.103:49155
```

Additionally, and importantly, any examples that use volumes or the `docker run` command with the `-v` flag to mount a local directory into a Docker container will not work with Boot2Docker. You can't mount a local directory on your OS X or Windows host into the Docker host running in the Boot2Docker virtual machine because they don't share a file system. Hence the current Boot2Docker release does not support the use of volumes. If you want to use any examples with volumes, such as those in Chapters 5 and 6, you will need to be running Docker on a Linux-based host. This is also true if running Docker in any virtual machine on Windows or OS X.

NOTE There is also a [great blog post](#) from Chris Jones that talks about these issues and suggests some workarounds.

Docker installation script

There is also an alternative method available to install Docker on an appropriate host using a remote installation script. To use this script we need to `curl` it from the get.docker.io website.

NOTE The script currently only supports Ubuntu, Fedora, Debian, and Gentoo installation. It may be updated shortly to include other distributions.

First, we'll need to ensure the `curl` command is installed.

Listing 2.41: Testing for curl

```
$ whereis curl
curl: /usr/bin/curl /usr/bin/X11/curl /usr/share/man/man1/curl.1.←
     gz
```

We can use `apt-get` to install `curl` if necessary.

Listing 2.42: Installing curl on Ubuntu

```
$ sudo apt-get -y install curl
```

Or we can use `yum` on Fedora.

Listing 2.43: Installing curl on Fedora

```
$ sudo yum -y install curl
```

Now we can use the script to install Docker.

Listing 2.44: Installing Docker from the installation script

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

This will ensure that the required dependencies are installed and check that our kernel is an appropriate version and that it supports an appropriate storage driver.

It will then install Docker and start the Docker daemon.

Binary installation

If we don't wish to use any of the package-based installation methods, we can download the latest binary version of Docker.

Listing 2.45: Downloading the Docker binary

```
$ wget http://get.docker.io/builds/Linux/x86_64/docker-latest.tgz
```

I recommend not taking this approach, as it reduces the maintainability of your Docker installation. Using packages is simpler and easier to manage, especially if using automation or configuration management tools.

The Docker daemon

After we've installed Docker, we need to confirm that the Docker daemon is running. Docker runs as a root-privileged daemon process to allow it to handle operations that can't be executed by normal users (e.g., mounting filesystems). The `docker` binary runs as a client of this daemon and also requires root privileges to run.

The Docker daemon should be started by default when the Docker package is installed. By default, the daemon listens on a Unix socket at `/var/run/docker.sock` for incoming Docker requests. If a group named `docker` exists on our system, Docker will apply ownership of the socket to that group. Hence, any user that belongs to the `docker` group can run Docker without needing to use the `sudo` command.

WARNING Remember that although the `docker` group makes life easier, it is still a security exposure. The `docker` group is root-equivalent and should be limited to those users and applications who absolutely need it.

Configuring the Docker daemon

We can change how the Docker daemon binds by adjusting the `-H` flag when the daemon is run.

We can use the `-H` flag to specify different interface and port configuration; for example, binding to the network:

Listing 2.46: Changing Docker daemon networking

```
$ sudo /usr/bin/docker -d -H tcp://0.0.0.0:2375
```

This would bind the Docker daemon to all interfaces on the host. Docker isn't automatically aware of networking changes on the client side. We will need to specify the `-H` option to point the docker client at the server; for example, `docker -H :4200` would be required if we had changed the port to 4200. Or, if we don't want to specify the `-H` on each client call, Docker will also honor the content of the `DOCKER_HOST` environment variable..

Listing 2.47: Using the DOCKER_HOST environment variable

```
$ export DOCKER_HOST="tcp://0.0.0.0:2375"
```

WARNING By default, Docker client-server communication is not authenticated. This means that if you bind Docker to an exposed network interface, anyone can connect to the daemon. There is, however, some TLS authentication available in Docker 0.9 and later. You'll see how to enable it when we look at the Docker API in Chapter 8.

We can also specify an alternative Unix socket path with the `-H` flag; for example, to use `unix://home/docker/docker.sock`:

Listing 2.48: Binding the Docker daemon to a different socket

```
$ sudo /usr/bin/docker -d -H unix://home/docker/docker.sock
```

Or we can specify multiple bindings like so:

Listing 2.49: Binding the Docker daemon to multiple places

```
$ sudo /usr/bin/docker -d -H tcp://0.0.0.0:2375 -H unix://home/←  
docker/docker.sock
```

We can also increase the verbosity of the Docker daemon by prefixing the daemon start command with `DEBUG=1`. Currently, Docker has limited log output. Indeed, if we are running on Ubuntu using Upstart, then generally only the output generated by the daemon is in `/var/log/upstart/docker.log`.

Listing 2.50: Turning on Docker daemon debug

```
DEBUG=1 /usr/bin/docker -d
```

If we want to make these changes permanent, we'll need to edit the various startup configurations. On Ubuntu, this is done by editing the `/etc/default/docker` file and changing the `DOCKER_OPTS` variable.

On Fedora and Red Hat distributions, this can be configured by editing the `/usr/←/lib/systemd/system/docker.service` file and adjusting the `ExecStart` line.

NOTE On other platforms, you can manage and update the Docker daemon's starting configuration via the appropriate `init` mechanism.

Checking that the Docker daemon is running

On Ubuntu, if Docker has been installed via package, we can check if the daemon is running with the Upstart status command:

Listing 2.51: Checking the status of the Docker daemon

```
$ sudo status docker
docker start/running, process 18147
```

We can then start or stop the Docker daemon with the Upstart start and stop commands, respectively.

Listing 2.52: Starting and stopping Docker with Upstart

```
$ sudo stop docker
docker stop/waiting
$ sudo start docker
docker start/running, process 18192
```

On Red Hat and Fedora, we can do similarly using the service shortcuts.

Listing 2.53: Starting and stopping Docker on Red Hat and Fedora

```
$ sudo service docker stop
Redirecting to /bin/systemctl stop docker.service
$ sudo service docker start
Redirecting to /bin/systemctl start docker.service
```

If the daemon isn't running, then the docker binary client will fail with an error message similar to this:

Listing 2.54: The Docker daemon isn't running

```
2014/05/18 20:08:32 Cannot connect to the Docker daemon. Is '←
docker -d' running on this host?
```

NOTE Prior to version 0.4.0 of Docker, the docker binary had a stand-alone mode, meaning it would run without the Docker daemon running. This mode has now been deprecated.

Upgrading Docker

After you've installed Docker, it is also easy to upgrade it when required. If you installed Docker using native packages via `apt-get` or `yum`, then you can also use these channels to upgrade it.

For example, run the `apt-get update` command and then install the new version of Docker.

Listing 2.55: Upgrade docker

```
$ sudo apt-get update
$ sudo apt-get install lxc-docker
```

Docker user interfaces

You can also potentially use a graphical user interface to manage Docker once you've got it installed. Currently, there are a small number of Docker user interfaces and web consoles available in various states of development, including:

- [Shipyard](#) - Shipyard gives you the ability to manage Docker resources, including containers, images, hosts, and more from a single management interface. It's open source, and the code is available [here](#).
- [DockerUI](#) - DockerUI is a web interface that allows you to interact with the Docker Remote API. It's written in JavaScript using the AngularJS framework.
- [maDocker](#) - A Web UI written in NodeJS and Backbone (in early stages of development).

Summary

In this chapter, we've seen how to install Docker on a variety of platforms. We've also seen how to manage the Docker daemon.

Chapter 2: Installing Docker

In the next chapter, we're going to start using Docker. We'll begin with container basics to give you an introduction to basic Docker operations. If you're all set up and ready to go then jump onto Chapter 3.

Chapter 3

Getting Started with Docker

In the last chapter, we saw how to install Docker and ensure the Docker daemon is up and running. In this chapter we're going to see how to take our first steps with Docker and work with our first container. This chapter will provide you with the basics of how to interact with Docker.

Ensuring Docker is ready

We're going to start with checking that Docker is working correctly, and then we're going to take a look at the basic Docker workflow: creating and managing containers. We'll take a container through its typical lifecycle from creation to a managed state and then stop and remove it.

Firstly, let's check that the `docker` binary exists and is functional:

Listing 3.1: Checking the docker binary works

```
$ sudo docker info
Containers: 0
Images: 0
Storage Driver: aufs
Root Dir: /var/lib/docker/aufs
Dirs: 144
Execution Driver: native-0.1
Kernel Version: 3.8.0-29-generic
Registry: [https://index.docker.io/v1/]
```

Here, we've passed the `info` command to the `docker` binary, which returns a list of any containers, any images (the building blocks Docker uses to build containers), the execution and storage drivers Docker is using, and its basic configuration.

As we've learned in previous chapters, Docker has a client-server architecture. It has a single binary, `docker`, that can act as both client and server. As a client, the `docker` binary passes requests to the Docker daemon (e.g., asking it to return information about itself), and then processes those requests when they are returned.

Running our first container

Now let's try and launch our first container with Docker. We're going to use the `docker run` command to create a container. The `docker run` command provides all of the "launch" capabilities for Docker. We'll be using it a lot to create new containers.

TIP You can find a full list of the available Docker commands [here](#) or by typing `docker help`. You can also use the Docker `man` pages (e.g., `man docker-run`).

Listing 3.2: Creating our first container

```
$ sudo docker run -i -t ubuntu /bin/bash
Pulling repository ubuntu from https://index.docker.io/v1
Pulling image 8<-
  dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c <-
    (precise) from ubuntu
Pulling 8<-
  dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c <-
    metadata
Pulling 8<-
  dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c <-
    fs layer
Downloading 58337280/? (n/a)
Pulling image <-
  b750fe79269d2ec9a3c593ef05b4332b1d1a02a62b4accb2c21d589ff2f5f2dc <-
    (quantal) from ubuntu
Pulling image 27cf784147099545 () from ubuntu
root@fcd78e1a3569:/#
```

Wow. A bunch of stuff happened here when we ran this command. Let's look at each piece.

Listing 3.3: The docker run command

```
$ sudo docker run -i -t ubuntu /bin/bash
```

First, we told Docker to run a command using `docker run`. We passed it two command line flags: `-i` and `-t`. The `-i` flag keeps STDIN open from the container, even if we're not attached to it. This persistent standard input is one half of what we need for an interactive shell. The `-t` flag is the other half and tells Docker to assign a pseudo-tty to the container we're about to create. This provides us with an interactive shell in the new container. This line is the base configuration needed to create a container with which we plan to interact on the command line rather than run as a daemonized service.

TIP You can find a full list of the available Docker run flags [here](#) or by typing `docker help run`. You can also use the Docker `man` pages (e.g., `example man docker-run`.)

Next, we told Docker which image to use to create a container, in this case the `ubuntu` image. The `ubuntu` image is a stock image, also known as a "base" image, provided by Docker, Inc., on the [Docker Hub](#) registry. You can use base images like the `ubuntu` base image (and the similar `fedora`, `debian`, `centos`, etc., images) as the basis for building your own images on the operating system of your choice. For now, we're just running the base image as the basis for our container and not adding anything to it.

So what was happening in the background here? Firstly, Docker checked locally for the `ubuntu` image. If it can't find the image on our local Docker host, it will reach out to the [Docker Hub](#) registry run by Docker, Inc., and look for it there. Once Docker had found the image, it downloaded the image and stored it on the local host.

Docker then used this image to create a new container inside a filesystem. The container has a network, IP address, and a bridge interface to talk to the local host. Finally, we told Docker which command to run in our new container, in this case launching a Bash shell with the `/bin/bash` command.

When the container had been created, Docker ran the `/bin/bash` command inside it; the container's shell was presented to us like so:

Listing 3.4: Our first container's shell

```
root@f7cbdac22a02:/#
```

NOTE We're going to see in Chapter 4 how to build our own images to use as the basis for containers.

Working with our first container

We are now logged into a new container, with the catchy ID of `f7cbdac22a02`, as the `root` user. This is a fully fledged Ubuntu host, and we can do anything we like in it. Let's explore it a bit, starting with asking for its hostname.

Listing 3.5: Checking the container's hostname

```
root@f7cbdac22a02:/# hostname
f7cbdac22a02
```

We can see that our container's hostname is the container ID. Let's have a look at the `/etc/hosts` file too.

Listing 3.6: Checking the container's /etc/hosts

```
root@f7cbdac22a02:/# cat /etc/hosts
172.17.0.4 f7cbdac22a02
127.0.0.1 localhost
::1 localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

Docker has also added a host entry for our container with its IP address. Let's also check out its networking configuration.

Listing 3.7: Checking the container's interfaces

```
root@f7cbdac22a02:/# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        inet6 ::1/128 scope host
            valid_lft forever preferred_lft forever
899: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 16:50:3a:b6:f2:cc brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.4/16 scope global eth0
        inet6 fe80::1450:3aff:feb6:f2cc/64 scope link
            valid_lft forever preferred_lft forever
```

As we can see, we have the `lo` loopback interface and the standard `eth0` network interface with an IP address of `172.17.0.4`, just like any other host. We can also check its running processes.

Listing 3.8: Checking container's processes

```
root@f7cbdac22a02:/# ps -aux
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME
COMMAND
root        1  0.0  0.0  18156  1936 ?          Ss  May30   0:00
/bin/bash
root       21  0.0  0.0  15568  1100 ?          R+  02:38   0:00
ps -aux
```

Now, how about we install a package?

Listing 3.9: Installing a package in our first container

```
root@f7cbdac22a02:/# apt-get update && apt-get install vim
```

We'll now have Vim installed in our container.

You can keep playing with the container for as long as you like. When you're done, type `exit`, and you'll return to the command prompt of your Ubuntu host.

So what's happened to our container? Well, it has now stopped running. The container only runs for as long as the command we specified, `/bin/bash`, is running. Once we exited the container, that command ended, and the container was stopped.

The container still exists; we can show a list of current containers using the `docker ps -a` command.

ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
f7cbdac22a02	ubuntu:12.04	/bin/bash	22 seconds ago	Exit 0		gray_cat
69d37e230d99	jamtur01/wordpress:latest	/bin/bash /start.sh	27 hours ago	Up 27 hours	0.0.0.0:49153->22/tcp, 0.0.0.0:49154->80/tcp	blue_bird
a0ef25127b65	jamtur01/wordpress:latest	/bin/bash /start.sh	27 hours ago	Exit 137		blue_wolf

Figure 3.1: Listing Docker containers

By default, when we run just `docker ps`, we will only see the running containers. When we specify the `-a` flag, the `docker ps` command will show us all containers, both stopped and running.

TIP You can also use the `docker ps` command with the `-l` flag to show the last container that was run, whether it is running or stopped.

We can see quite a bit of information about our container: its ID, the image used to create it, the command it last ran, when it was created, and its exit status (in our case, 0, because it was exited normally using the `exit` command). We can also see that each container has a name.

NOTE There are three ways containers can be identified: a short UUID (like `f7cbdac22a02`), a longer UUID (like `f7cbdac22a02e03c9438c729345e54db9d20cf` `a2ac1fc3494b6eb60872e74778`), and a name (like `gray_cat`).

Container naming

Docker will automatically generate a name at random for each container we create. We can see that the container we've just created is called `gray_cat`. If we want to specify a particular container name in place of the automatically generated name, we can do so using the `--name` flag.

Listing 3.10: Naming a container

```
$ sudo docker run --name bob_the_container -i -t ubuntu /bin/bash
root@aa3f365f0f4e:/# exit
```

This would create a new container called `bob_the_container`. A valid container name can contain the following characters: a to z, A to Z, the digits 0 to 9, the underscore, period, and dash (or, expressed as a regular expression: `[a-zA-Z0-9_.-]`).

We can use the container name in place of the container ID in most Docker commands, as we'll see. Container names are useful to help us identify and build logical connections between containers and applications. It's also much easier to remember a specific container name (e.g., `web` or `db`) than a container ID or even a random name. I recommend using container names to make managing your containers easier.

Names are unique. If we try to create two containers with the same name, the command will fail. We need to delete the previous container with the same name before we can create a new one. We can do so with the `docker rm` command.

Starting a stopped container

So what to do with our now-stopped `bob_the_container` container? Well, if we want, we can restart a stopped container like so:

Listing 3.11: Starting a stopped container

```
$ sudo docker start bob_the_container
```

We could also refer to the container by its container ID instead.

Listing 3.12: Starting a stopped container by ID

```
$ sudo docker start aa3f365f0f4e
```

TIP We can also use the `docker restart` command.

Now if we run the `docker ps` command without the `-a` flag, we'll see our running container.

Attaching to a container

Our container will restart with the same options we'd specified when we launched it with the `docker run` command. So there is an interactive session waiting on our running container. We can reattach to that session using the `docker attach` command.

Listing 3.13: Attaching to a running container

```
$ sudo docker attach bob_the_container
```

or via its container ID:

Listing 3.14: Attaching to a running container via ID

```
$ sudo docker attach aa3f365f0f4e
```

and we'll be brought back to our container's Bash prompt:

TIP You might need to hit `Enter` to bring up the prompt

Listing 3.15: Inside our re-attached container

```
root@aa3f365f0f4e:/#
```

If we exit this shell, our container will again be stopped.

Creating daemonized containers

In addition to these interactive containers, we can create longer-running containers. Daemonized containers don't have the interactive session we've just used and are ideal for running applications and services. Most of the containers you're likely to run will probably be daemonized. Let's start a daemonized container now.

Listing 3.16: Creating a long running container

```
$ sudo docker run --name daemon_dave -d ubuntu /bin/sh -c "while <-
  true; do echo hello world; sleep 1; done"
1333bb1a66af402138485fe44a335b382c09a887aa9f95cb9725e309ce5b7db3
```

Here, we've used the `docker run` command with the `-d` flag to tell Docker to detach the container to the background.

We've also specified a `while` loop as our container command. Our loop will echo `hello world` over and over again until the container is stopped or the process stops.

With this combination of flags, you'll see that, instead of being attached to a shell like our last container, the `docker run` command has instead returned a container ID and returned us to our command prompt. Now if we run `docker ps`, we can see a running container.

Listing 3.17: Viewing our running `daemon_dave` container

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
1333bb1a66af	ubuntu:14.04	/bin/sh -c 'while tr 32 secs ago Up 27<- daemon_dave				

Seeing what's happening inside our container

We now have a daemonized container running our while loop; let's take a look inside the container and see what's happening. To do so, we can use the `docker logs` command. The `docker logs` command fetches the logs of a container.

Listing 3.18: Fetching the logs of our daemonized container

```
$ sudo docker logs daemon_dave
hello world
.
.
```

Here we can see the results of our while loop echoing `hello world` to the logs. Docker will output the last few log entries and then return. We can also monitor the container's logs much like the `tail -f` binary operates using the `-f` flag..

Listing 3.19: Tailing the logs of our daemonized container

```
$ sudo docker logs -f daemon_dave
hello world
.
.
```

TIP Use `Ctrl-C` to exit from the log tail.

You can also tail a portion of the logs of a container, again much like the `tail` command with the `-f` `--lines` flags. For example, you can get the last ten lines of a log by using `docker logs --tail 10 daemon_dave`. You can also follow the logs of a container without having to read the whole log file with `docker logs --tail 0 -f daemon_dave`.

To make debugging a little easier, we can also add the `-t` flag to prefix our log entries with timestamps.

Listing 3.20: Tailing the logs of our daemonized container

```
$ sudo docker logs -ft daemon_dave
[May 10 13:06:17.934] hello world
[May 10 13:06:18.935] hello world
[May 10 13:06:19.937] hello world
[May 10 13:06:20.939] hello world
[May 10 13:06:21.942] hello world
. . .
```

TIP Again, use `Ctrl-C` to exit from the log tail.

Inspecting the container's processes

In addition to the container's logs we can also inspect the processes running inside the container. To do this, we use the `docker top` command.

Listing 3.21: Inspecting the processes of the daemonized container

```
$ sudo docker top daemon_dave
```

We can then see each process (principally our `while` loop), the user it is running

as, and the process ID.

Listing 3.22: The docker top output

```
PID  USER  COMMAND
 977  root  /bin/sh -c while true; do echo hello world; sleep 1; ↵
      done
 1123  root  sleep 1
```

Running a process inside a container

We can also run additional processes inside our containers using the docker `exec` command. There are two types of commands we can run inside a container: background and interactive. Background tasks run inside the container without interaction and interactive tasks remain in the foreground. Interactive tasks are useful for tasks like opening a shell inside a container. Let's look at an example of a background task.

Listing 3.23: Running a background task inside a container

```
$ sudo docker exec -d daemon_dave touch /etc/new_config_file
```

Here the `-d` flag indicates we're running a background process. We then specify the name of the container to run the command inside and the command to be executed. In this case our command will create a new empty file called `/etc/new_config_file` inside our `daemon_dave` container. We can use a docker `exec` background command to run maintenance, monitoring or management tasks inside a running container.

We can also run interactive tasks like opening a shell inside our `daemon_dave` container.

Listing 3.24: Running an interactive command inside a container

```
$ sudo docker exec -t -i daemon_dave /bin/bash
```

The `-t` and `-i` flags, like the flags used when running an interactive container, create a TTY and capture STDIN for our executed process. We then specify the name of the container to run the command inside and the command to be executed. In this case our command will create a new bash session inside the container `daemon_dave`. We could then use this session to issue other commands inside our container.

NOTE The `docker exec` command was introduced in Docker 1.3 and is not available in earlier releases. For earlier Docker releases you should see the `nsenter` command explained in Chapter 6.

Stopping a daemonized container

If we wish to stop our daemonized container, we can do it with the `docker stop` command, like so:

Listing 3.25: Stopping the running Docker container

```
$ sudo docker stop daemon_dave
```

or again via its container ID.

Listing 3.26: Stopping the running Docker container by ID

```
$ sudo docker stop c2c4e57c12c4
```

NOTE The `docker stop` command sends a SIGTERM signal to the Docker container's running process. If you want to stop a container a bit more enthusiastically, you can use the `docker kill` command, which will send a SIGKILL signal to the container's process.

Run `docker ps` to check the status of the now-stopped container. Useful here is the `docker ps -n x` flag which shows the last x containers, running or stopped.

Automatic container restarts

If your container has stopped because of a failure you can configure Docker to restart it using the `--restart` flag. The `--restart` flag checks for the container's exit code and makes a decision whether or not to restart it. The default behavior is to not restart containers at all.

You specify the `--restart` flag with the `docker run` command.

Listing 3.27: Automatically restarting containers

```
$ sudo docker run --restart=always --name daemon_dave -d ubuntu /<br/>bin/sh -c "while true; do echo hello world; sleep 1; done"
```

In this example the `--restart` flag has been set to `always`. Docker will try to restart the container no matter what exit code is returned. Alternatively, we can specify a value of `on-failure` which restarts the container if it exits with a non-zero exit code. The `on-failure` flag also accepts an optional restart count.

Listing 3.28: On-failure restart count

```
--restart=on-failure:5
```

This will attempt to restart the container a maximum of five times if a non-zero exit code is received.

NOTE The `--restart` flag was introduced in Docker 1.2.0.

Finding out more about our container

In addition to the information we retrieved about our container using the `docker ps` command, we can get a whole lot more information using the `docker inspect` command.

Listing 3.29: Inspecting a container

```
$ sudo docker inspect daemon_dave
[{
  "ID": "c2c4e57c12c4c142271c031333823af95d64b20b5d607970c334784430bcbd0f",
  "Created": "2014-05-10T11:49:01.902029966Z",
  "Path": "/bin/sh",
  "Args": [
    "-c",
    "while true; do echo hello world; sleep 1; done"
  ],
  "Config": {
    "Hostname": "c2c4e57c12c4",
    ...
  }
}]
```

The `docker inspect` command will interrogate our container and return its configuration information, including names, commands, networking configuration, and a wide variety of other useful data.

We can also selectively query the inspect results hash using the `-f` or `--format` flag.

Listing 3.30: Selectively inspecting a container

```
$ sudo docker inspect --format='{{ .State.Running }}' daemon_dave
false
```

This will return the running state of the container, which in our case is `false`. We can also get useful information like the container's IP address.

Listing 3.31: Inspecting the container's IP address

```
$ sudo docker inspect --format '{{ .NetworkSettings.IPAddress }}' \
 \
daemon_dave
172.17.0.2
```

TIP The --format or -f flag is a bit more than it seems on the surface. It's actually a full Go template being exposed. You can make use of all [the capabilities of a Go template](#) when querying it.

We can also list multiple containers and receive output for each.

Listing 3.32: Inspecting multiple containers

```
$ sudo docker inspect --format '{{.Name}} {{.State.Running}}' \
daemon_dave bob_the_container
/daemon_dave false
/bob_the_container false
```

We can select any portion of the inspect hash to query and return.

NOTE In addition to inspecting containers, you can see a bit more about how Docker works by exploring the /var/lib/docker directory. This directory holds your images, containers, and container configuration. You'll find all your containers in the /var/lib/docker/containers directory.

Deleting a container

If you are finished with a container, you can delete it using the `docker rm` command.

NOTE It's important to note that you can't delete a running Docker container. You must stop it first using the `docker stop` command or `docker kill` command.

Listing 3.33: Deleting a container

```
$ sudo docker rm 80430f8d0921
80430f8d0921
```

There isn't currently a way to delete all containers, but you can slightly cheat with a command like the following:

Listing 3.34: Deleting all containers

```
docker rm `docker ps -a -q`
```

This command will list all of the current containers using the `docker ps` command. The `-a` flag lists all containers, and the `-q` flag only returns the container IDs rather than the rest of the information about your containers. This list is then passed to the `docker rm` command, which deletes each container.

Summary

We've now been introduced to the basic mechanics of how Docker containers work. This information will form the basis for how we'll learn to use Docker in the rest of the book.

In the next chapter, we're going to explore building our own Docker images and working with Docker repositories and registries.

Chapter 4

Working with Docker images and repositories

In Chapter 2, we learned how to install Docker. In Chapter 3, we learned how to use a variety of commands to manage Docker containers, including the `docker run` command.

Let's see the `docker run` command again.

Listing 4.1: Revisiting creating a basic Docker container

```
$ sudo docker run -i -t --name another_container_mum ubuntu \
/bin/bash
root@b415b317ac75:/#
```

This command will launch a new container called `another_container_mum` from the `ubuntu` image and open a Bash shell.

In this chapter, we're going to explore Docker images: the building blocks from which we launch containers. We'll learn a lot more about Docker images, what they are, how to manage them, how to modify them, and how to create, store, and share your own images. We'll also examine the repositories that hold images and the registries that store repositories.

What is a Docker image?

Let's continue our journey with Docker by learning a bit more about Docker images. A Docker image is made up of filesystems layered over each other. At the base is a boot filesystem, `bootfs`, which resembles the typical Linux/Unix boot filesystem. A Docker user will probably never interact with the boot filesystem. Indeed, when a container has booted, it is moved into memory, and the boot filesystem is unmounted to free up the RAM used by the `initrd` disk image.

So far this looks pretty much like a typical Linux virtualization stack. Indeed, Docker next layers a root filesystem, `rootfs`, on top of the boot filesystem. This `rootfs` can be one or more operating systems (e.g., a Debian or Ubuntu filesystem).

In a more traditional Linux boot, the root filesystem is mounted read-only and then switched to read-write after boot and an integrity check is conducted. In the Docker world, however, the root filesystem stays in read-only mode, and Docker takes advantage of a [union mount](#) to add more read-only filesystems onto the root filesystem. A union mount is a mount that allows several filesystems to be mounted at one time but appear to be one filesystem. The union mount overlays the filesystems on top of one another so that the resulting filesystem may contain files and subdirectories from any or all of the underlying filesystems.

Docker calls each of these filesystems images. Images can be layered on top of one another. The image below is called the parent image and you can traverse each layer until you reach the bottom of the image stack where the final image is called the base image. Finally, when a container is launched from an image, Docker mounts a read-write filesystem on top of any layers below. This is where whatever processes we want our Docker container to run will execute.

This sounds confusing, so perhaps it is best represented by a diagram.

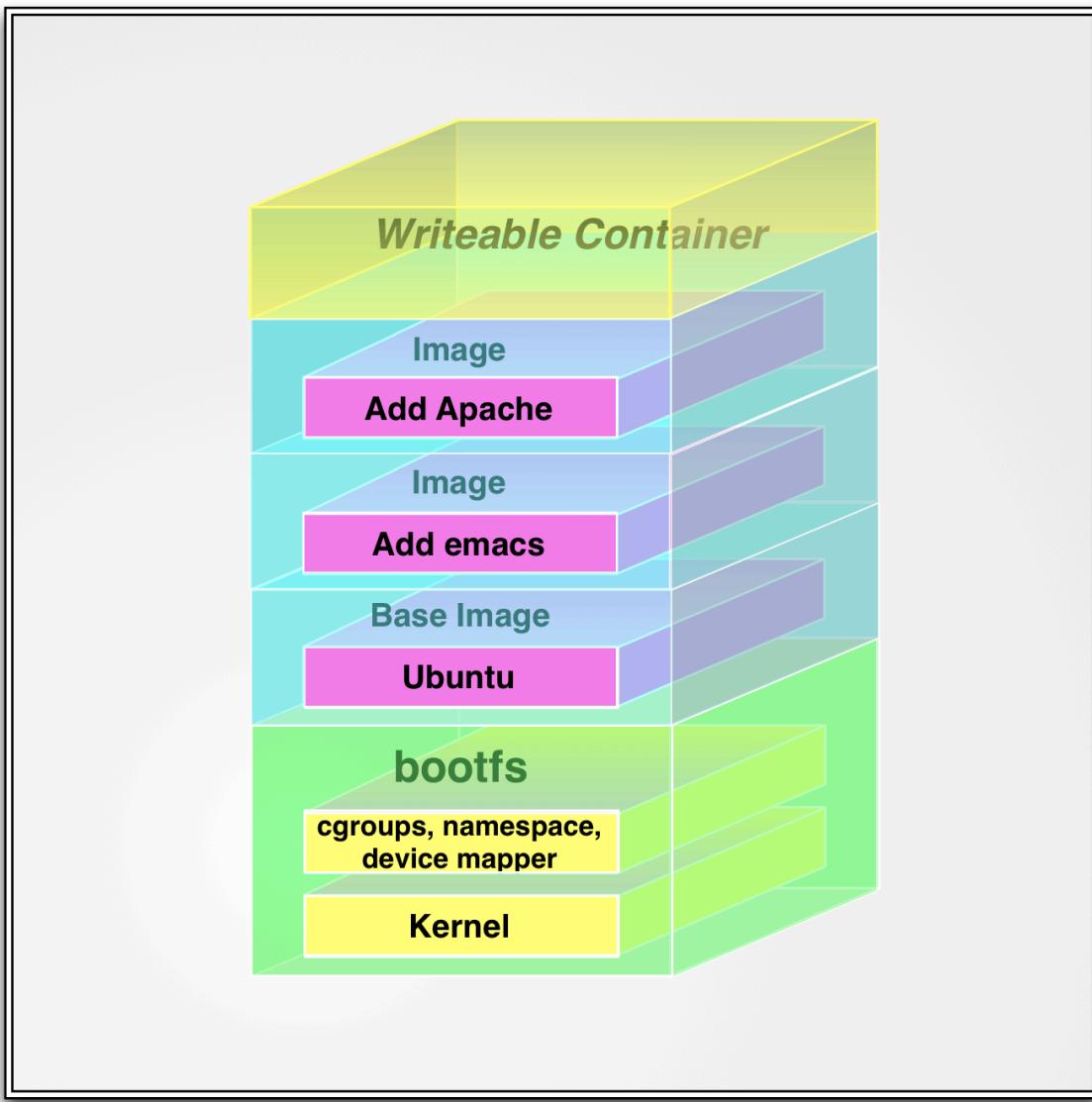


Figure 4.1: The Docker filesystem layers

When Docker first starts a container, the initial read-write layer is empty. As changes occur, they are applied to this layer; for example, if you want to change a file, then that file will be copied from the read-only layer below into the read-write layer. The read-only version of the file will still exist but is now hidden underneath the copy.

This pattern is traditionally called "copy on write" and is one of the features that makes Docker so powerful. Each read-only image layer is read-only; this image never changes. When a container is created, Docker builds from the stack of images and then adds the read-write layer on top. That layer, combined with the knowledge of the image layers below it and some configuration data, form the container. As we discovered in the last chapter, containers can be changed, they have state, and they can be started and stopped. This, and the image-layering framework, allows us to quickly build images and run containers with our applications and services.

List of Docker images

Let's get started with Docker images by looking at what images are available to us on our Docker host. We can do this using the `docker images` command.

Listing 4.2: Listing Docker images

```
$ sudo docker images
REPOSITORY TAG      IMAGE ID      CREATED      VIRTUAL SIZE
ubuntu      latest  c4ff7513909d 6 days ago  225.4 MB
```

We can see that we've got an image, from a repository called `ubuntu`. So where does this image come from? Remember in Chapter 3, when we ran the `docker-run` command, that part of the process was downloading an image? In our case, it's the `ubuntu` image.

NOTE Local images live on our local Docker host in the `/var/lib/docker` directory. Each image will be inside a directory named for your storage driver; for example, `aufs` or `devicemapper`. You'll also find all your containers in the `/var/lib/docker/containers` directory.

That image was downloaded from a repository. Images live inside repositories,

Chapter 4: Working with Docker images and repositories

and repositories live on registries. The default registry is the public registry managed by Docker, Inc., [Docker Hub](#).

TIP The Docker registry code is open source. You can also run your own registry, as we'll see later in this chapter.

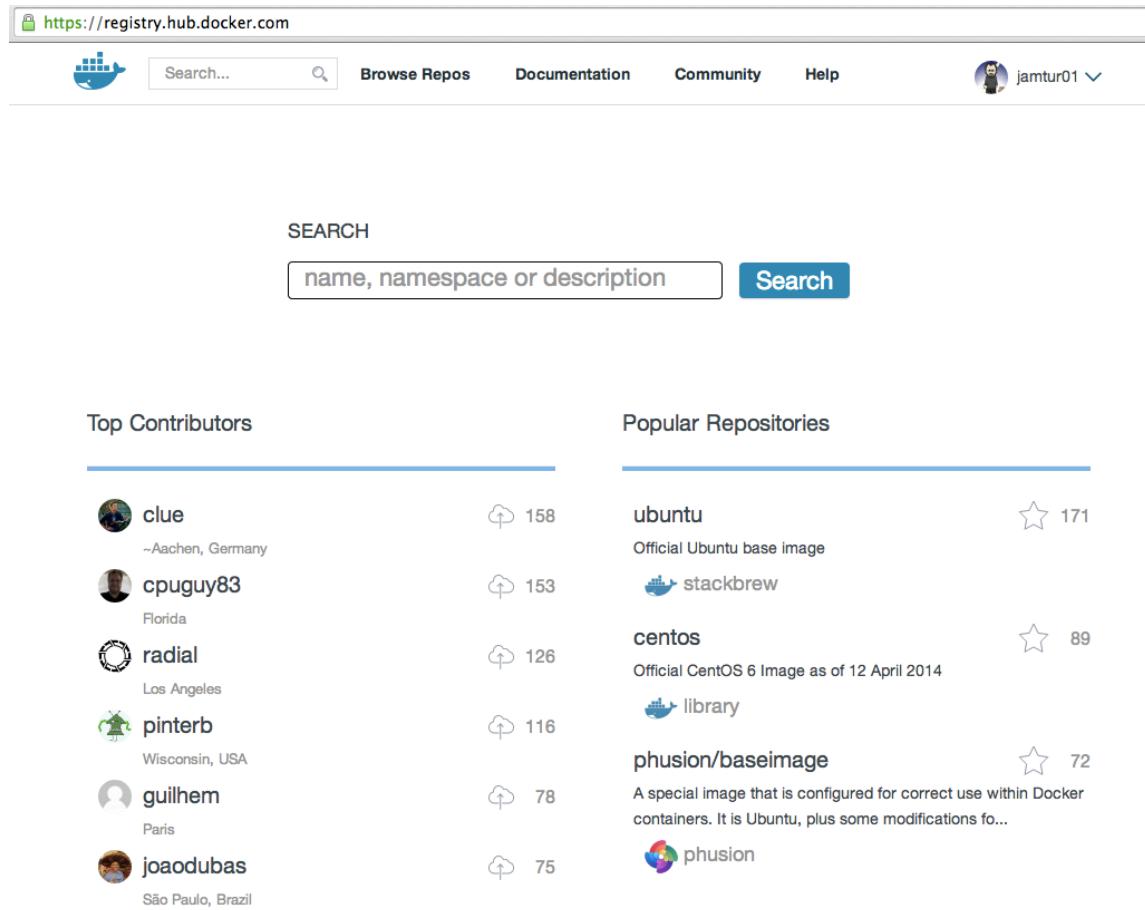


Figure 4.2: Docker Hub

Inside [Docker Hub](#) (or on a Docker registry you run yourself), images are stored in repositories. You can think of an image repository as being much like a Git repository. It contains images, layers, and metadata about those images.

Chapter 4: Working with Docker images and repositories

Each repository can contain multiple images (e.g., the `ubuntu` repository contains images for Ubuntu 12.04, 12.10, 13.04, 13.10, and 14.04). Let's get the rest of the images in the `ubuntu` repository now.

Listing 4.3: Pulling the Ubuntu image

```
$ sudo docker pull ubuntu
Pulling repository ubuntu
c4ff7513909d: Pulling dependent layers
3db9c44f4520: Pulling dependent layers
75204fdb260b: Pulling dependent layers
.
.
```

Here we've used the `docker pull` command to pull down the entire contents of the `ubuntu` repository.

Let's see what our `docker images` command reveals now.

Listing 4.4: Listing all the ubuntu Docker images

```
$ sudo docker images
REPOSITORY TAG      IMAGE ID      CREATED      VIRTUAL SIZE
ubuntu      13.10      5e019ab7bf6d  2 weeks ago  180 MB
ubuntu      saucy       5e019ab7bf6d  2 weeks ago  180 MB
ubuntu      12.04       74fe38d11401  2 weeks ago  209.6 MB
ubuntu      precise     74fe38d11401  2 weeks ago  209.6 MB
ubuntu      12.10       a7cf8ae4e998  2 weeks ago  171.3 MB
ubuntu      quantal     a7cf8ae4e998  2 weeks ago  171.3 MB
ubuntu      14.04       99ec81b80c55  2 weeks ago  266 MB
ubuntu      latest      c4ff7513909d  6 days ago   225.4 MB
ubuntu      trusty      99ec81b80c55  2 weeks ago  266 MB
ubuntu      raring      316b678ddf48  2 weeks ago  169.4 MB
ubuntu      13.04       316b678ddf48  2 weeks ago  169.4 MB
ubuntu      10.04       3db9c44f4520  3 weeks ago  183 MB
ubuntu      lucid       3db9c44f4520  3 weeks ago  183 MB
```

You can see we've now got a series of `ubuntu` images. We can see that the `ubuntu` image is actually a series of images collected under a single repository. In this

case, when we pulled the `ubuntu` image, we actually got several versions of the Ubuntu operating system, including 10.04, 12.04, 13.04, and 14.04.

NOTE We call it the Ubuntu operating system, but really it is not the full operating system. It's a very cut-down version with the bare runtime required to run the distribution.

We identify each image inside that repository by what Docker calls tags. Each image is being listed by the tags applied to it, so, for example, 12.10, 12.04, quantal, or precise and so on. Each tag marks together a series of image layers that represent a specific image (e.g., the 12.04 tag collects together all the layers of the Ubuntu 12.04 image). This allows us to store more than one image inside a repository.

We can refer to a specific image inside a repository by suffixing the repository name with a colon and a tag name, for example:

Listing 4.5: Running a tagged Docker image

```
$ sudo docker run -t -i --name new_container ubuntu:12.04 /bin/←  
  bash  
root@79e36bfff89b4:/#
```

This launches a container from the `ubuntu:12.04` image, which is an Ubuntu 12.04 operating system. We can also see that some images with the same ID (see image ID `74fe38d11401`) are tagged more than once. Image ID `74fe38d11401` is actually tagged both 12.04 and precise: the version number and code name for that Ubuntu release, respectively.

It's always a good idea to build a container from specific tags. That way we'll know exactly what the source of our container is. There are differences, for example, between Ubuntu 12.04 and 14.04, so it would be useful to specifically state that we're using `ubuntu:12.04` so we know exactly what we're getting.

There are two types of repositories: user repositories, which contain images contributed by Docker users, and top-level repositories, which are controlled by the

people behind Docker.

A user repository takes the form of a username and a repository name; for example, `jamtur01/puppet`.

- Username: `jamtur01`
- Repository name: `puppet`

Alternatively, a top-level repository only has a repository name like `ubuntu`. The top-level repositories are managed by Docker Inc and by selected vendors who provide curated base images that you can build upon (e.g., the Fedora team provides a `fedora` image). The top-level repositories also represent a commitment from vendors and Docker Inc that the images contained in them are well constructed, secure, and up to date.

WARNING User-contributed images are built by members of the Docker community. You should use them at your own risk: they are not validated or verified in any way by Docker Inc.

Pulling images

When we run a container from images with the `docker run` command, if the image isn't present locally already then Docker will download it from the Docker Hub. By default, if you don't specify a specific tag, Docker will download the `latest` tag, for example:

Listing 4.6: Docker run and the default latest tag

```
$ sudo docker run -t -i --name next_container ubuntu /bin/bash
root@23a42cee91c3:/#
```

will download the `ubuntu:latest` image if it isn't already present on the host.

Chapter 4: Working with Docker images and repositories

Alternatively, we can use the `docker pull` command to pull images down ourselves. Using `docker pull` saves us some time launching a container from a new image. Let's see that now by pulling down the `fedora` base image.

Listing 4.7: Pulling the fedora image

```
$ sudo docker pull fedora
Pulling repository fedora
5cc9e91966f7: Download complete
b7de3133ff98: Download complete
511136ea3c5a: Download complete
ef52fb1fe610: Download complete
```

Let's see this new image on our Docker host using the `docker images` command. This time, however, let's narrow our review of the images to only the `fedora` images. To do so, we can specify the image name after the `docker images` command.

Listing 4.8: Viewing the fedora image

```
$ sudo docker images fedora
REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE
fedora rawhide 5cc9e91966f7 6 days ago 372.7 MB
fedora 20 b7de3133ff98 3 weeks ago 372.7 MB
fedora heisenbug b7de3133ff98 3 weeks ago 372.7 MB
fedora latest b7de3133ff98 3 weeks ago 372.7 MB
```

We can see that the `fedora` image contains the development Rawhide release as well as Fedora 20. We can also see that the Fedora 20 release is tagged in three ways -- `20`, `heisenbug`, and `latest` -- but it is the same image (we can see all three entries have an ID of `b7de3133ff98`). If we wanted the Fedora 20 image, therefore, we could use any of the following:

- `fedora:20`
- `fedora:heisenbug`
- `fedora:latest`

We could have also just downloaded one tagged image using the `docker pull` command.

Listing 4.9: Pulling a tagged fedora image

```
$ sudo docker pull fedora:20
```

This would have just pulled the `fedora:20` image.

Searching for images

We can also search all of the publicly available images on [Docker Hub](#) using the `docker search` command:

Listing 4.10: Searching for images

```
$ sudo docker search puppet
NAME                  DESCRIPTION STARS OFFICIAL AUTOMATED
wfarr/puppet-module...
jamtur01/puppetmaster
...
```

TIP You can also browse the available images online at [Docker Hub](#).

Here, we've searched the Docker Hub for the term `puppet`. It'll search images and return:

- Repository names
- Image descriptions
- Stars - these measure the popularity of an image
- Official - an image managed by the upstream developer (e.g., the `fedora` image managed by the Fedora team)
- Automated - an image built by the Docker Hub's Automated Build process

NOTE We'll see more about Automated Builds later in this chapter.

Let's pull down one of these images.

Listing 4.11: Pulling down the `jamtur01/puppetmaster` image

```
$ sudo docker pull jamtur01/puppetmaster
```

This will pull down the `jamtur01/puppetmaster` image (which, by the way, contains a pre-installed Puppet master).

We can then use this image to build a new container. Let's do that now using the `docker run` command again.

Listing 4.12: Creating a Docker container from the Puppet master image

```
$ sudo docker run -i -t jamtur01/puppetmaster /bin/bash
root@4655dee672d3:/# facter
architecture => amd64
augeasversion => 1.2.0
.
.
.
root@4655dee672d3:/# puppet --version
3.4.3
```

You can see we've launched a new container from our `jamtur01/puppetmaster` image. We've launched the container interactively and told the container to run the Bash shell. Once inside the container's shell, we've run Facter (Puppet's inventory application), which was pre-installed on our image. From inside the container, we've also run the `puppet` binary to confirm it is installed.

Building our own images

So we've seen that we can pull down pre-prepared images with custom contents. How do we go about modifying our own images and updating and managing them?

There are two ways to create a Docker image:

- Via the `docker commit` command
- Via the `docker build` command with a Dockerfile

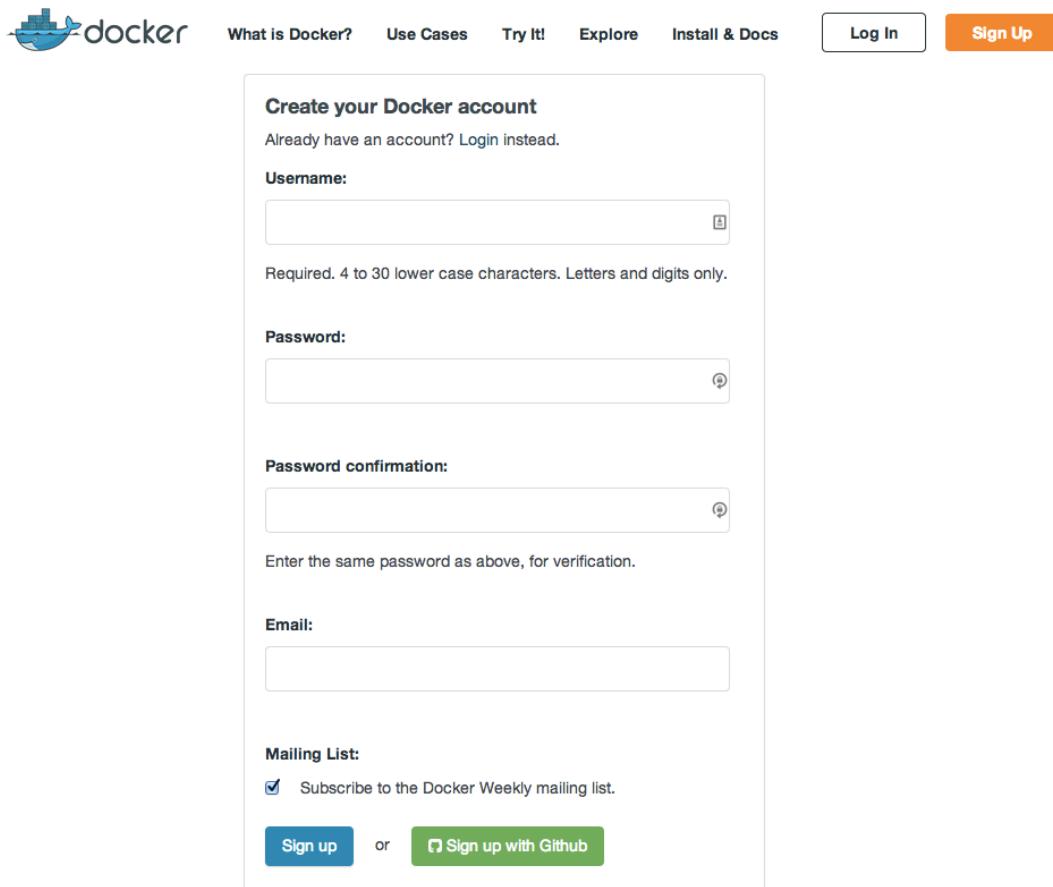
The `docker commit` method is not currently recommended, as building with a Dockerfile is far more flexible and powerful, but we'll demonstrate it to you for the sake of completeness. After that, we'll focus on the recommended method of building Docker images: writing a Dockerfile and using the `docker build` command.

NOTE We don't generally actually "create" new images; rather, we build new images from existing base images, like the `ubuntu` or `fedora` images we've already seen. If you want to build an entirely new base image, you can see some information on this [here](#).

Creating a Docker Hub account

A big part of image building is sharing and distributing your images. We do this by pushing them to the [Docker Hub](#) or your own registry. To facilitate this, let's start by creating an account on the Docker Hub. You can join Docker Hub [here](#).

Chapter 4: Working with Docker images and repositories



The screenshot shows the Docker Hub sign-up page. At the top, there is a navigation bar with links: 'What is Docker?', 'Use Cases', 'Try It!', 'Explore', 'Install & Docs', 'Log In' (in a grey box), and 'Sign Up' (in an orange box). The main form is titled 'Create your Docker account' and includes a link 'Already have an account? Login instead.' Below the title, there is a 'Username:' field with a note 'Required. 4 to 30 lower case characters. Letters and digits only.' Below it is a 'Password:' field with a note 'Enter the same password as above, for verification.' Below that is an 'Email:' field. At the bottom of the form, there is a 'Mailing List:' section with a checked checkbox 'Subscribe to the Docker Weekly mailing list.' Below the checkbox are two buttons: 'Sign up' (blue) and 'Sign up with Github' (green).

Figure 4.3: Creating a Docker Hub account.

Create an account and verify your email address from the email you'll receive after signing up.

Now let's test our new account from Docker. To sign into the Docker Hub you can use the `docker login` command.

Listing 4.13: Logging into the Docker Hub

```
$ sudo docker login
Username: jamtur01
Password:
Email: james@lovedthanlost.net
Login Succeeded
```

This command will log you into the Docker Hub and store your credentials for future use.

NOTE Your credentials will be stored in the `$HOME/.dockercfg` file.

Using Docker commit to create images

The first method of creating images used the `docker commit` command. You can think about this method as much like making a commit in a version control system. We create a container, make changes to that container as you would change code, and then commit those changes to a new image.

Let's start by creating a container from the `ubuntu` image we've used in the past.

Listing 4.14: Creating a custom container to modify

```
$ sudo docker run -i -t ubuntu /bin/bash
root@4aab3ce3cb76:/#
```

Next, we'll install Apache into our container.

Listing 4.15: Adding the Apache package

```
root@4aab3ce3cb76:/# apt-get -yqq update
. . .
root@4aab3ce3cb76:/# apt-get -y install apache2
. . .
```

We've launched our container and then installed Apache within it. We're going to use this container as a web server, so we'll want to save it in its current state. That will save us from having to rebuild it with Apache every time we create a new container. To do this we exit from the container, using the `exit` command, and use the `docker commit` command.

Listing 4.16: Committing the custom container

```
$ sudo docker commit 4aab3ce3cb76 jamtur01/apache2  
8ce0ea7a1528
```

You can see we've used the `docker commit` command and specified the ID of the container we've just changed (to find that ID you could use the `docker ps -l -q` command to return the ID of the last created container) as well as a target repository and image name, here `jamtur01/apache2`. Of note is that the `docker commit` command only commits the differences between the image the container was created from and the current state of the container. This means updates are very lightweight.

Let's look at our new image.

Listing 4.17: Reviewing our new image

```
$ sudo docker images jamtur01/apache2  
.  
.  
jamtur01/apache2  latest  8ce0ea7a1528  13 seconds ago  90.63 MB
```

We can also provide some more data about our changes when committing our image, including tags. For example:

Listing 4.18: Committing another custom container

```
$ sudo docker commit -m="A new custom image" --author="James Turnbull" \  
4aab3ce3cb76 jamtur01/apache2:webserver  
f99ebb6fed1f559258840505a0f5d5b6173177623946815366f3e3acff01adef
```

Here, we've specified some more information while committing our new image. We've added the `-m` option which allows us to provide a commit message explaining our new image. We've also specified the `--author` option to list the author of the image. We've then specified the ID of the container we're committing. Finally, we've specified the username and repository of the image, `jamtur01/apache2`, and we've added a tag, `webserver`, to our image.

We can view this information about our image using the `docker inspect` command.

Listing 4.19: Inspecting our committed image

```
$ sudo docker inspect jamtur01/apache2:webserver
[{
    "Architecture": "amd64",
    "Author": "James Turnbull",
    "Comment": "A new custom image",
    ...
}]
```

TIP You can find a full list of the `docker commit` flags [here](#).

If we want to run a container from our new image, we can do so using the `docker run` command.

Listing 4.20: Running a container from our committed image

```
$ sudo docker run -t -i jamtur01/apache2:webserver /bin/bash
```

You'll note that we've specified our image with the full tag: `jamtur01/apache2:webserver`.

Building images with a Dockerfile

We don't recommend the `docker commit` approach. Instead, we recommend that you build images using a definition file called a **Dockerfile** and the `docker build` command. The **Dockerfile** uses a basic DSL with instructions for building Docker images. We then use the `docker build` command to build a new image from the instructions in the **Dockerfile**.

Our first Dockerfile

Let's now create a directory and an initial Dockerfile. We're going to build a Docker image that contains a simple web server.

Listing 4.21: Creating a sample repository

```
$ mkdir static_web
$ cd static_web
$ touch Dockerfile
```

We've created a directory called `static_web` to hold our Dockerfile. This directory is our build environment, which is what Docker calls a context or build context. Docker will upload the build context, as well as any files and directories contained in it, to our Docker daemon when the build is run. This provides the Docker daemon with direct access to any code, files or other data you might want to include in the image.

We've also created an empty Dockerfile file to get started. Now let's look at an example of a Dockerfile to create a Docker image that will act as a Web server.

Listing 4.22: Our first Dockerfile

```
# Version: 0.0.1
FROM ubuntu:14.04
MAINTAINER James Turnbull "james@example.com"
RUN apt-get update
RUN apt-get install -y nginx
RUN echo 'Hi, I am in your container' \
    >/usr/share/nginx/html/index.html
EXPOSE 80
```

The Dockerfile contains a series of instructions paired with arguments. Each instruction, for example `FROM`, should be in upper-case and be followed by an argument: `FROM ubuntu:14.04`. Instructions in the Dockerfile are processed from the top down, so you should order them accordingly.

Each instruction adds a new layer to the image and then commits the image.

Docker executing instructions roughly follow a workflow:

- Docker runs a container from the image.
- An instruction executes and makes a change to the container.
- Docker runs the equivalent of `docker commit` to commit a new layer.
- Docker then runs a new container from this new image.
- The next instruction in the file is executed, and the process repeats until all instructions have been executed.

This means that if your Dockerfile stops for some reason (for example, if an instruction fails to complete), you will be left with an image you can use. This is highly useful for debugging: you can run a container from this image interactively and then debug why your instruction failed using the last image created.

NOTE The Dockerfile also supports comments. Any line that starts with a `#` is considered a comment. You can see an example of this in the first line of our Dockerfile.

The first instruction in a Dockerfile should always be `FROM`. The `FROM` instruction specifies an existing image that the following instructions will operate on; this image is called the base image.

In our sample Dockerfile we've specified the `ubuntu:14.04` image as our base image. This specification will build an image on top of an Ubuntu 14.04 base operating system. As with running a container, you should always be specific about exactly from which base image you are building.

Next, we've specified the `MAINTAINER` instruction, which tells Docker who the author of the image is and what their email address is. This is useful for specifying an owner and contact for an image.

We've followed these instructions with three `RUN` instructions. The `RUN` instruction executes commands on the current image. The commands in our example: updating the installed APT repositories, installing the `nginx` package, then creating the

`/usr/share/nginx/html/index.html` file containing some example text. As we've discovered, each of these instructions will create a new layer and, if successful, will commit that layer and then execute the next instruction.

By default, the `RUN` instruction executes inside a shell using the command wrapper `/bin/sh -c`. If you are running the instruction on a platform without a shell or you wish to execute without a shell (for example, to avoid shell string munging), you can specify the instruction in `exec` format:

Listing 4.23: The `RUN` instruction in `exec` form

```
RUN [ "apt-get", "install", "-y", "nginx" ]
```

We use this format to specify an array containing the command to be executed and then each parameter to pass to the command.

Next, we've specified the `EXPOSE` instruction, which tells Docker that the application in this container will use this specific port on the container. That doesn't mean you can automatically access whatever service is running on that port (here, port 80) on the container. For security reasons, Docker doesn't open the port automatically, but waits for you to do it when you run the container using the `docker run` command. We'll see this shortly when we create a new container from this image.

You can specify multiple `EXPOSE` instructions to mark multiple ports to be exposed.

NOTE Docker also uses the `EXPOSE` instruction to help link together containers, which we'll see in Chapter 5.

Building the image from our `Dockerfile`

All of the instructions will be executed and committed and a new image returned when we run the `docker build` command. Let's try that now:

Listing 4.24: Running the Dockerfile

```
$ cd static_web
$ sudo docker build -t="jamtur01/static_web" .
Sending build context to Docker daemon 2.56 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu:14.04
--> ba5877dc9bec
Step 1 : MAINTAINER James Turnbull "james@example.com"
--> Running in b8ffa06f9274
--> 4c66c9dcee35
Removing intermediate container b8ffa06f9274
Step 2 : RUN apt-get update
--> Running in f331636c84f7
--> 9d938b9e0090
Removing intermediate container f331636c84f7
Step 3 : RUN apt-get install -y nginx
--> Running in 4b989d4730dd
--> 93fb180f3bc9
Removing intermediate container 4b989d4730dd
Step 4 : RUN echo 'Hi, I am in your container'      >/usr/share/←
nginx/html/index.html
--> Running in b51bacc46eb9
--> b584f4ac1def
Removing intermediate container b51bacc46eb9
Step 5 : EXPOSE 80
--> Running in 7ff423bd1f4d
--> 22d47c8cb6e5
Successfully built 22d47c8cb6e5
```

We've used the `docker build` command to build our new image. We've specified the `-t` option to mark our resulting image with a repository and a name, here the `jamtur01` repository and the image name `static_web`. I strongly recommend you always name your images to make it easier to track and manage them.

You can also tag images during the build process by suffixing the tag after the

image name with a colon, for example:

Listing 4.25: Tagging a build

```
$ sudo docker build -t="jamtur01/static_web:v1" .
```

TIP If you don't specify any tag, Docker will automatically tag your image as latest.

The trailing . tells Docker to look in the local directory to find the Dockerfile. You can also specify a Git repository as a source for the Dockerfile as we can see here:

Listing 4.26: Building from a Git repository

```
$ sudo docker build -t="jamtur01/static_web:v1" \
git@github.com:jamtur01/docker-static_web
```

Here Docker assumes that there is a Dockerfile located in the root of the Git repository.

But back to our docker build process. You can see that the build context has been uploaded to the Docker daemon.

Listing 4.27: Uploading the build context to the daemon

```
Sending build context to Docker daemon 2.56 kB
Sending build context to Docker daemon
```

TIP If a file named .dockerignore exists in the root of the build context then it is interpreted as a newline-separated list of exclusion patterns. Much like a .gitignore file it excludes the listed files from being uploaded to the build context. Globbing can be done using Go's [filepath](#).

Next, you can see that each instruction in the Dockerfile has been executed with the image ID, 22d47c8cb6e5, being returned as the final output of the build process. Each step and its associated instruction are run individually, and Docker has committed the result of each operation before outputting that final image ID.

What happens if an instruction fails?

Earlier, we talked about what happens if an instruction fails. Let's look at an example: let's assume that in Step 4 we got the name of the required package wrong and instead called it `ngin`.

Let's run the build again and see what happens when it fails.

Listing 4.28: Managing a failed instruction

```
$ cd static_web
$ sudo docker build -t="jamtur01/static_web" .
Sending build context to Docker daemon 2.56 kB
Sending build context to Docker daemon
Step 1 : FROM ubuntu:14.04
--> 8dbd9e392a96
Step 2 : MAINTAINER James Turnbull "james@example.com"
--> Running in d97e0c1cf6ea
--> 85130977028d
Step 3 : RUN apt-get update
--> Running in 85130977028d
--> 997485f46ec4
Step 4 : RUN apt-get install -y ngin
--> Running in ffca16d58fd8
Reading package lists...
Building dependency tree...
Reading state information...
E: Unable to locate package ngin
2014/06/04 18:41:11 The command [/bin/sh -c apt-get install -y ←
ngin] returned a non-zero code: 100
```

Let's say I want to debug this failure. I can use the `docker run` command to create a container from the last step that succeeded in my Docker build, here the image ID `997485f46ec4`.

Listing 4.29: Creating a container from the last successful step

```
$ sudo docker run -t -i 997485f46ec4 /bin/bash
dcge12e59fe8:/#
```

I can then try to run the `apt-get install -y nginx` step again with the right package name or conduct some other debugging to determine what went wrong. Once I've identified the issue, I can exit the container, update my `Dockerfile` with the right package name, and retry my build.

Dockerfiles and the build cache

As a result of each step being committed as an image, Docker is able to be really clever about building images. It will treat previous layers as a cache. If, in our debugging example, we did not need to change anything in Steps 1 to 3, then Docker would use the previously built images as a cache and a starting point. Essentially, it'd start the build process straight from Step 4. This can save you a lot of time when building images if a previous step has not changed. If, however, you did change something in Steps 1 to 3, then Docker would restart from the first changed instruction.

Sometimes, though, you want to make sure you don't use the cache. For example, if you'd cached Step 3 above, `apt-get update`, then it wouldn't refresh the APT package cache. You might want it to do this to get a new version of a package. To skip the cache, we can use the `--no-cache` flag with the `docker build` command..

Listing 4.30: Bypassing the Dockerfile build cache

```
$ sudo docker build --no-cache -t="jamtur01/static_web" .
```

Using the build cache for templating

As a result of the build cache, you can build your Dockerfiles in the form of simple templates (e.g., adding a package repository or updating packages near the top of the file to ensure the cache is hit). I generally have the same template set of instructions in the top of my Dockerfile, for example for Ubuntu:

Listing 4.31: A template Ubuntu Dockerfile

```
FROM ubuntu:14.04
MAINTAINER James Turnbull "james@example.com"
ENV REFRESHED_AT 2014-07-01
RUN apt-get -qq update
```

Let's step through this new Dockerfile. Firstly, I've used the `FROM` instruction to specify a base image of `ubuntu:14.04`. Next I've added my `MAINTAINER` instruction to provide my contact details. I've then specified a new instruction, `ENV`. The `ENV` instruction sets environment variables in the image. In this case, I've specified the `ENV` instruction to set an environment variable called `REFRESHED_AT`, showing when the template was last updated. Lastly, I've specified the `apt-get -qq` update command in a `RUN` instruction. This refreshes the APT package cache when it's run, ensuring that the latest packages are available to install.

With my template, when I want to refresh the build, I change the date in my `ENV` instruction. Docker then resets the cache when it hits that `ENV` instruction and runs every subsequent instruction anew without relying on the cache. This means my `RUN apt-get update` instruction is rerun and my package cache is refreshed with the latest content. You can extend this template example for your target platform or to fit a variety of needs. For example, for a `fedora` image we might:

Listing 4.32: A template Fedora Dockerfile

```
FROM fedora:20
MAINTAINER James Turnbull "james@example.com"
ENV REFRESHED_AT 2014-07-01
RUN yum -q makecache
```

which performs a very similar function for Fedora using Yum.

Viewing our new image

Now let's take a look at our new image. We can do this using the `docker images` command.

Listing 4.33: Listing our new Docker image

```
$ sudo docker images jamtur01/static_web
REPOSITORY          TAG      ID           CREATED        SIZE
jamtur01/static_web latest  22d47c8cb6e5  24 seconds ago  12.29 kB<-
(virtual 326 MB)
```

If we want to drill down into how our image was created, we can use the `docker history` command.

Listing 4.34: Using the docker history command

```
$ sudo docker history 22d47c8cb6e5
IMAGE      CREATED      CREATED BY <-
           SIZE
22d47c8cb6e5  6 minutes ago  /bin/sh -c #(nop) EXPOSE map[80/tcp<-
:{}]          0 B
b584f4ac1def  6 minutes ago  /bin/sh -c echo 'Hi, I am in your <-
container'    27 B
93fb180f3bc9  6 minutes ago  /bin/sh -c apt-get install -y nginx <-
18.46 MB
9d938b9e0090  6 minutes ago  /bin/sh -c apt-get update <-
20.02 MB
4c66c9dcee35  6 minutes ago  /bin/sh -c #(nop) MAINTAINER James <-
Turnbull " 0 B
. . .
```

We can see each of the image layers inside our new `jamtur01/static_web` image and the `Dockerfile` instruction that created them.

Launching a container from our new image

We can also now launch a new container using our new image and see if what we've built has worked.

Listing 4.35: Launching a container from our new image

```
$ sudo docker run -d -p 80 --name static_web jamtur01/static_web ←
  \
  nginx -g "daemon off;" ←
6751b94bb5c001a650c918e9a7f9683985c3eb2b026c2f1776e61190669494a8
```

Here I've launched a new container called `static_web` using the `docker run` command and the name of the image we've just created. We've specified the `-d` option, which tells Docker to run detached in the background. This allows us to run long-running processes like the Nginx daemon. We've also specified a command for the container to run: `nginx -g "daemon off;"`. This will launch Nginx in the foreground to run our web server.

We've also specified a new flag, `-p`. The `-p` flag manages which network ports Docker exposes at runtime. When you run a container, Docker has two methods of assigning ports on the Docker host:

- Docker can randomly assign a high port from the range 49000 to 49900 on the Docker host that maps to port 80 on the container.
- You can specify a specific port on the Docker host that maps to port 80 on the container.

This will open a random port on the Docker host that will connect to port 80 on the Docker container.

Let's look at what port has been assigned using the `docker ps` command.

Listing 4.36: Viewing the Docker port mapping

```
$ sudo docker ps -l
CONTAINER ID  IMAGE          NAMES
6751b94bb5c0  jamtur01/static_web:latest ... 0.0.0.0:49154->80/tcp
          static_web
```

We can see that port 49154 is mapped to the container port of 80. We can get the same information with the `docker port` command.

Listing 4.37: The docker port command

```
$ sudo docker port 6751b94bb5c0 80
0.0.0.0:49154
```

We've specified the container ID and the container port for which we'd like to see the mapping, 80, and it has returned the mapped port, 49154.

The `-p` option also allows us to be flexible about how a port is exposed to the host. For example, we can specify that Docker bind the port to a specific port:

Listing 4.38: Exposing a specific port with `-p`

```
$ sudo docker run -d -p 80:80 --name static_web jamtur01/
  static_web \
  nginx -g "daemon off;"
```

This will bind port 80 on the container to port 80 on the local host. Obviously, it's important to be wary of this direct binding: if you're running multiple containers, only one container can bind a specific port on the local host. This can limit Docker's flexibility.

To avoid this, we could bind to a different port.

Listing 4.39: Binding to a different port

```
$ sudo docker run -d -p 8080:80 --name static_web jamtur01/←  
static_web \  
nginx -g "daemon off;"
```

This would bind port 80 on the container to port 8080 on the local host.

We can also bind to a specific interface.

Listing 4.40: Binding to a specific interface

```
$ sudo docker run -d -p 127.0.0.1:80:80 --name static_web ←  
jamtur01/static_web \  
nginx -g "daemon off;"
```

Here we've bound port 80 of the container to port 80 on the 127.0.0.1 interface on the local host. We can also bind to a random port using the same structure.

Listing 4.41: Binding to a random port on a specific interface

```
$ sudo docker run -d -p 127.0.0.1::80 --name static_web jamtur01/←  
static_web \  
nginx -g "daemon off;"
```

Here we've removed the specific port to bind to on 127.0.0.1. We would now use the `docker inspect` or `docker port` command to see which random port was assigned to port 80 on the container.

TIP You can bind UDP ports by adding the suffix `/udp` to the port binding.

Docker also has a shortcut, `-P`, that allows us to expose all ports we've specified via `EXPOSE` instructions in our `Dockerfile`.

Listing 4.42: Exposing a port with docker run

```
$ sudo docker run -d -P --name static_web jamtur01/static_web \
nginx -g "daemon off;"
```

This would expose port 80 on a random port on our local host. It would also expose any additional ports we had specified with other EXPOSE instructions in the Dockerfile that built our image.

TIP You can find more information on port redirection [here](#).

With this port number, we can now view the web server on the running container using the IP address of our host or the localhost on 127.0.0.1.

NOTE You can find the IP address of your local host with the `ifconfig` or `ip addr` command.

Listing 4.43: Connecting to the container via curl

```
$ curl localhost:49154
Hi, I am in your container
```

Now we've got a very simple Docker-based web server.

Dockerfile instructions

We've already seen some of the available Dockerfile instructions, like RUN and EXPOSE. But there are also a variety of other instructions we can put in our Dockerfile. These include CMD, ENTRYPOINT, ADD, COPY, VOLUME, WORKDIR, USER, ONBUILD, and ENV. You can see a full list of the available Dockerfile instructions [here](#).

We'll also see a lot more Dockerfiles in the next few chapters and see how to build some cool applications into Docker containers.

CMD

The `CMD` instruction specifies the command to run when a container is launched. It is similar to the `RUN` instruction, but rather than running the command when the container is being built, it will specify the command to run when the container is launched, much like specifying a command to run when launching a container with the `docker run` command, for example:

Listing 4.44: Specifying a specific command to run

```
$ sudo docker run -i -t jamtur01/static_web /bin/true
```

This would be articulated in the Dockerfile as:

Listing 4.45: Using the CMD instruction

```
CMD ["/bin/true"]
```

You can also specify parameters to the command, like so:

Listing 4.46: Passing parameters to the CMD instruction

```
CMD ["/bin/bash", "-l"]
```

Here we're passing the `-l` flag to the `/bin/bash` command.

WARNING You'll note that the command is contained in an array. This tells Docker to run the command 'as-is'. You can also specify the `CMD` instruction without an array, in which case Docker will prepend `/bin/sh -c` to the command. This may result in unexpected behavior when the command is executed. As a result, it is recommended that you always use the array syntax.

Lastly, it's important to understand that we can override the `CMD` instruction using

the `docker run` command. If we specify a `CMD` in our `Dockerfile` and one on the `docker run` command line, then the command line will override the `Dockerfile`'s `CMD` instruction.

NOTE It's also important to understand the interaction between the `CMD` instruction and the `ENTRYPOINT` instruction. We'll see some more details of this below.

Let's look at this process a little more closely. Let's say our `Dockerfile` contains the `CMD`:

Listing 4.47: Overriding `CMD` instructions in the `Dockerfile`

```
CMD [ "/bin/bash" ]
```

We can build a new image (let's call it `jamtur01/test`) using the `docker build` command and then launch a new container from this image.

Listing 4.48: Launching a container with a `CMD` instruction

```
$ sudo docker run -t -i jamtur01/test
root@e643e6218589:/#
```

Notice something different? We didn't specify the command to be executed at the end of the `docker run`. Instead, Docker used the command specified by the `CMD` instruction.

If, however, I did specify a command, what would happen?

Listing 4.49: Overriding a command locally

```
$ sudo docker run -i -t jamtur01/test /bin/ps
PID TTY      TIME CMD
1 ? 00:00:00 ps
$
```

You can see here that we have specified the `/bin/ps` command to list running processes. Instead of launching a shell, the container merely returned the list

of running processes and stopped, overriding the command specified in the `CMD` instruction.

TIP You can only specify one `CMD` instruction in a `Dockerfile`. If more than one is specified, then the last `CMD` instruction will be used. If you need to run multiple processes or commands as part of starting a container you should use a service management tool like [Supervisor](#).

ENTRYPOINT

Closely related to the `CMD` instruction, and often confused with it, is the `ENTRYPOINT` instruction.. So what's the difference between the two, and why are they both needed? As we've just discovered, we can override the `CMD` instruction on the `docker run` command line. Sometimes this isn't great when we want a container to behave in a certain way. The `ENTRYPOINT` instruction provides a command that isn't as easily overridden. Instead, any arguments we specify on the `docker run` command line will be passed as arguments to the command specified in the `ENTRYPOINT`. Let's see an example of an `ENTRYPOINT` instruction.

Listing 4.50: Specifying an ENTRYPOINT

```
ENTRYPOINT ["/usr/sbin/nginx"]
```

Like the `CMD` instruction, we also specify parameters by adding to the array. For example:

Listing 4.51: Specifying an ENTRYPOINT parameter

```
ENTRYPOINT ["/usr/sbin/nginx", "-g", "daemon off;"]
```

NOTE As with the `CMD` instruction above, you can see that we've specified the `ENTRYPOINT` command in an array to avoid any issues with the command being

prepended with `/bin/sh -c`.

Now let's rebuild our image with an `ENTRYPOINT` of `ENTRYPOINT ["/usr/sbin/←
nginx"]`.

Listing 4.52: Rebuilding static_web with a new ENTRYPOINT

```
$ sudo docker build -t="jamtur01/static_web" .
```

And then launch a new container from our `jamtur01/static_web` image.

Listing 4.53: Using docker run with ENTRYPOINT

```
$ sudo docker run -t -i jamtur01/static_web -g "daemon off;"
```

As we can see, we've rebuilt our image and then launched an interactive container. We specified the argument `-g "daemon off;"`. This argument will be passed to the command specified in the `ENTRYPOINT` instruction, which will thus become `/←
usr/sbin/nginx -g "daemon off;"`. This command would then launch the Nginx daemon in the foreground and leave the container running as a web server server. We can also combine `ENTRYPOINT` and `CMD` to do some neat things. For example, we might want to specify the following in our Dockerfile.

Listing 4.54: Using ENTRYPOINT and CMD together

```
ENTRYPOINT [ "/usr/sbin/nginx" ]
CMD [ "-h" ]
```

Now when we launch a container, any option we specify will be passed to the Nginx daemon; for example, we could specify `-g "daemon off;"`; as we did above to run the daemon in the foreground. If we don't specify anything to pass to the container, then the `-h` is passed by the `CMD` instruction and returns the Nginx help text: `/usr/sbin/nginx -h`.

This allows us to build in a default command to execute when our container is run combined with overridable options and flags on the `docker run` command line.

TIP If required at runtime, you can override the `ENTRYPOINT` instruction using the `docker run` command with `--entrypoint` flag.

WORKDIR

The `WORKDIR` instruction provides a way to set the working directory for the container and the `ENTRYPOINT` and/or `CMD` to be executed when a container is launched from the image.

We can use it to set the working directory for a series of instructions or for the final container. For example, to set the working directory for a specific instruction we might:

Listing 4.55: Using the `WORKDIR` instruction

```
WORKDIR /opt/webapp/db
RUN bundle install
WORKDIR /opt/webapp
ENTRYPOINT [ "rakeup" ]
```

Here we've changed into the `/opt/webapp/db` directory to run `bundle install` and then changed into the `/opt/webapp` directory prior to specifying our `ENTRYPOINT` instruction of `rakeup`.

You can override the working directory at runtime with the `-w` flag, for example:

Listing 4.56: Overriding the working directory

```
$ sudo docker run -ti -w /var/log ubuntu pwd
/var/log
```

This will set the container's working directory to `/var/log`.

ENV

The ENV instruction is used to set environment variables during the image build process. For example:

Listing 4.57: Setting an environment variable in Dockerfile

```
ENV RVM_PATH /home/rvm/
```

This new environment variable will be used for any subsequent RUN instructions, as if we had specified an environment variable prefix to a command like so:

Listing 4.58: Prefixing a RUN instruction

```
RUN gem install unicorn
```

would be executed as:

Listing 4.59: Executing with an ENV prefix

```
RVM_PATH=/home/rvm/ gem install unicorn
```

These environment variables will also be persisted into any containers created from your image. So, if we were to run `env` in a container build with the `ENV RVM_PATH /home/rvm/` instruction we'd see:

Listing 4.60: Persistent environment variables in Docker containers

```
root@bf42aadc7f09:~# env
. . .
RVM_PATH=/home/rvm/
. . .
```

You can also pass environment variables on the `docker run` command line using the `-e` flag. These variables will only apply at runtime, for example:

Listing 4.61: Runtime environment variables

```
$ sudo docker run -ti -e "WEB_PORT=8080" ubuntu env
HOME=/
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=792b171c5e9f
TERM=xterm
WEB_PORT=8080
```

We can see that our container has the `WEB_PORT` environment variable set to 8080.

USER

The `USER` instruction specifies a user that the image should be run as; for example:

Listing 4.62: Using the USER instruction

```
USER nginx
```

This will cause containers created from the image to be run by the `nginx` user. We can specify a username or a UID and group or GID. Or even a combination thereof, for example:

Listing 4.63: Specifying USER and GROUP variants

```
USER user
USER user:group
USER uid
USER uid:gid
USER user:gid
USER uid:group
```

You can also override this at runtime by specifying the `-u` flag with the `docker run` command.

TIP The default user if you don't specify the `USER` instruction is `root`.

VOLUME

The `VOLUME` instruction adds volumes to any container created from the image. A volume is a specially designated directory within one or more containers that bypasses the Union File System to provide several useful features for persistent or shared data:

- Volumes can be shared and reused between containers.
- A container doesn't have to be running to share its volumes.
- Changes to a volume are made directly.
- Changes to a volume will not be included when you update an image.
- Volumes persist until no containers use them.

This allows us to add data (like source code), a database, or other content into an image without committing it to the image and allows us to share that data between containers. This can be used to do testing with containers and an application's code, manage logs, or handle databases inside a container. We'll see examples of this in Chapters 5 and 6.

You can use the `VOLUME` instruction like so:

Listing 4.64: Using the VOLUME instruction

```
VOLUME [ "/opt/project" ]
```

This would attempt to create a mount point `/opt/project` to any container created from the image.

Or we can specify multiple volumes by specifying an array:

Listing 4.65: Using multiple VOLUME instructions

```
VOLUME [ "/opt/project", "/data" ]
```

TIP We'll see a lot more about volumes and how to use them in Chapters 5 and 6. If you're curious in the meantime, you can read more about volumes [here](#).

ADD

The ADD instruction adds files and directories from our build environment into our image; for example, when installing an application. The ADD instruction specifies a source and a destination for the files, like so:

Listing 4.66: Using the ADD instruction

```
ADD software.lic /opt/application/software.lic
```

This ADD instruction will copy the file `software.lic` from the build directory to `/opt/application/software.lic` in the image. The source of the file can be a URL, filename, or directory as long as it is inside the build context or environment. You can cannot ADD files from outside the build directory or context.

When ADD'ing files Docker uses the ending character of the destination to determine what the source is. If the destination ends in a `/`, then it considers the source a directory. If it doesn't end in a `/`, it considers the source a file.

The source of the file can also be a URL; for example:

Listing 4.67: URL as the source of an ADD instruction

```
ADD http://wordpress.org/latest.zip /root/wordpress.zip
```

Lastly, the ADD instruction has some special magic for taking care of local tar archives. If a tar archive (valid archive types include gzip, bzip2, xz) is specified as the source file, then Docker will automatically unpack it for you:

Listing 4.68: Archive as the source of an ADD instruction

```
ADD latest.tar.gz /var/www/wordpress/
```

This will unpack the `latest.tar.gz` archive into the `/var/www/wordpress/` directory. The archive is unpacked with the same behavior as running tar with the

`-x` option: the output is the union of whatever exists in the destination plus the contents of the archive. If a file or directory with the same name already exists in the destination, it will not be overwritten.

WARNING Currently this will not work with a tar archive specified in a URL. This is somewhat inconsistent behavior and may change in a future release.

Finally, if the destination doesn't exist, Docker will create the full path for us, including any directories. New files and directories will be created with a mode of 0755 and a UID and GID of 0.

NOTE It's also important to note that the build cache can be invalidated by ADD instructions. If the files or directories added by an ADD instruction change then this will invalidate the cache for all following instructions in the Dockerfile.

COPY

The COPY instruction is closely related to the ADD instruction. The key difference is that the COPY instruction is purely focused on copying local files from the build context and does not have any extraction or decompression capabilities.

Listing 4.69: Using the COPY instruction

```
COPY conf.d/ /etc/apache2/
```

This will copy files from the `conf.d` directory to the `/etc/apache2/` directory.

The source of the files must be the path to a file or directory relative to the build context, the local source directory in which your Dockerfile resides. You cannot copy anything that is outside of this directory, because the build context is uploaded to the Docker daemon, and the copy takes place there. Anything outside

of the build context is not available. The destination should be an absolute path inside the container.

Any files and directories created by the copy will have a UID and GID of 0.

If the source is a directory, the entire directory is copied, including filesystem metadata; if the source is any other kind of file, it is copied individually along with its metadata. In our example, the destination ends with a trailing slash /, so it will be considered a directory and copied to the destination directory.

If the destination doesn't exist, it is created along with all missing directories in its path, much like how the `mkdir -p` command works.

ONBUILD

The `ONBUILD` instruction adds triggers to images. A trigger is executed when the image is used as the basis of another image (e.g., if you have an image that needs source code added from a specific location that might not yet be available, or if you need to execute a build script that is specific to the environment in which the image is built).

The trigger inserts a new instruction in the build process, as if it were specified right after the `FROM` instruction. The trigger can be any build instruction. For example:

Listing 4.70: Adding ONBUILD instructions

```
ONBUILD ADD . /app/src
ONBUILD RUN cd /app/src && make
```

This would add an `ONBUILD` trigger to the image being created, which we can see when we run `docker inspect` on the image.

Listing 4.71: Showing ONBUILD instructions with docker inspect

```
$ sudo docker inspect 508efa4e4bf8
...
"OnBuild": [
    "ADD . /app/src",
    "RUN cd /app/src/ && make"
]
...
```

For example, we'll build a new Dockerfile for an Apache2 image that we'll call `jamtur01/apache2`.

Listing 4.72: A new ONBUILD image Dockerfile

```
FROM ubuntu:14.04
MAINTAINER James Turnbull "james@example.com"
RUN apt-get update
RUN apt-get install -y apache2
ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2
ONBUILD ADD . /var/www/
EXPOSE 80
ENTRYPOINT ["/usr/sbin/apache2"]
CMD ["-D", "FOREGROUND"]
```

Now we'll build this image.

Listing 4.73: Building the apache2 image

```
$ sudo docker build -t="jamtur01/apache2" .
...
Step 7 : ONBUILD ADD . /var/www/
--> Running in 0e117f6ea4ba
--> a79983575b86
Successfully built a79983575b86
```

We now have an image with an `ONBUILD` instruction that uses the `ADD` instruction to add the contents of the directory we're building from to the `/var/www/` directory in our image. This could readily be our generic web application template from which I build web applications.

Let's try this now by building a new image called `webapp` from the following `Dockerfile`:

Listing 4.74: The webapp Dockerfile

```
FROM jamtur01/apache2
MAINTAINER James Turnbull "james@example.com"
ENV APPLICATION_NAME webapp
ENV ENVIRONMENT development
```

Let's look at what happens when I build this image.

Listing 4.75: Building our webapp image

```
$ sudo docker build -t="jamtur01/webapp" .
...
Step 0 : FROM jamtur01/apache2
# Executing 1 build triggers
Step onbuild-0 : ADD . /var/www/
--> 1a018213a59d
--> 1a018213a59d
Step 1 : MAINTAINER James Turnbull "james@example.com"
...
Successfully built 04829a360d86
```

We can see that straight after the `FROM` instruction, Docker has inserted the `ADD` instruction, specified by the `ONBUILD` trigger, and then proceeded to execute the remaining steps. This would allow me to always add the local source and, as I've done here, specify some configuration or build information for each application; hence, this becomes a useful template image.

The `ONBUILD` triggers are executed in the order specified in the parent image and are only inherited once (i.e., by children and not grandchildren). If we built an-

other image from this new image, a grandchild of the `jamtur01/apache2` image, then the triggers would not be executed when that image is built.

NOTE There are several instructions you can't `ONBUILD: FROM`, `MAINTAINER`, and `ONBUILD` itself. This is done to prevent inception-like recursion in Dockerfile builds.

Pushing images to the Docker Hub

Once we've got an image, we can upload it to the [Docker Hub](#). This allows us to make it available for others to use. For example, we could share it with others in our organization or make it publicly available.

NOTE The Docker Hub also has the option of private repositories. These are a paid-for feature that allows you to store an image in a private repository that is only available to you or anyone with whom you share it. This allows you to have private images containing proprietary information or code you might not want to share publicly.

We push images to the Docker Hub using the `docker push` command.

Let's try a push now.

Listing 4.76: Trying to push a root image

```
$ sudo docker push static_web
2013/07/01 18:34:47 Impossible to push a "root" repository. ←
  Please rename your repository in <user>/<repo> (ex: jamtur01/←
    static_web)
```

What's gone wrong here? We've tried to push our image to the repository `static_web`, but Docker knows this is a root repository. Root repositories are managed only by the Docker, Inc., team and will reject our attempt to write to them. Let's try again.

Listing 4.77: Pushing a Docker image

```
$ sudo docker push jamtur01/static_web
The push refers to a repository [jamtur01/static_web] (len: 1)
Processing checksums
Sending image list
Pushing repository jamtur01/static_web to registry-1.docker.io (1←
  tags)
.
.
```

This time, our push has worked, and we've written to a user repository, `jamtur01←/static_web`. We would write to your own user ID, which we created earlier, and to an appropriately named image (e.g., `youruser/yourimage`).

We can now see our uploaded image on the [Docker Hub](#).

Chapter 4: Working with Docker images and repositories

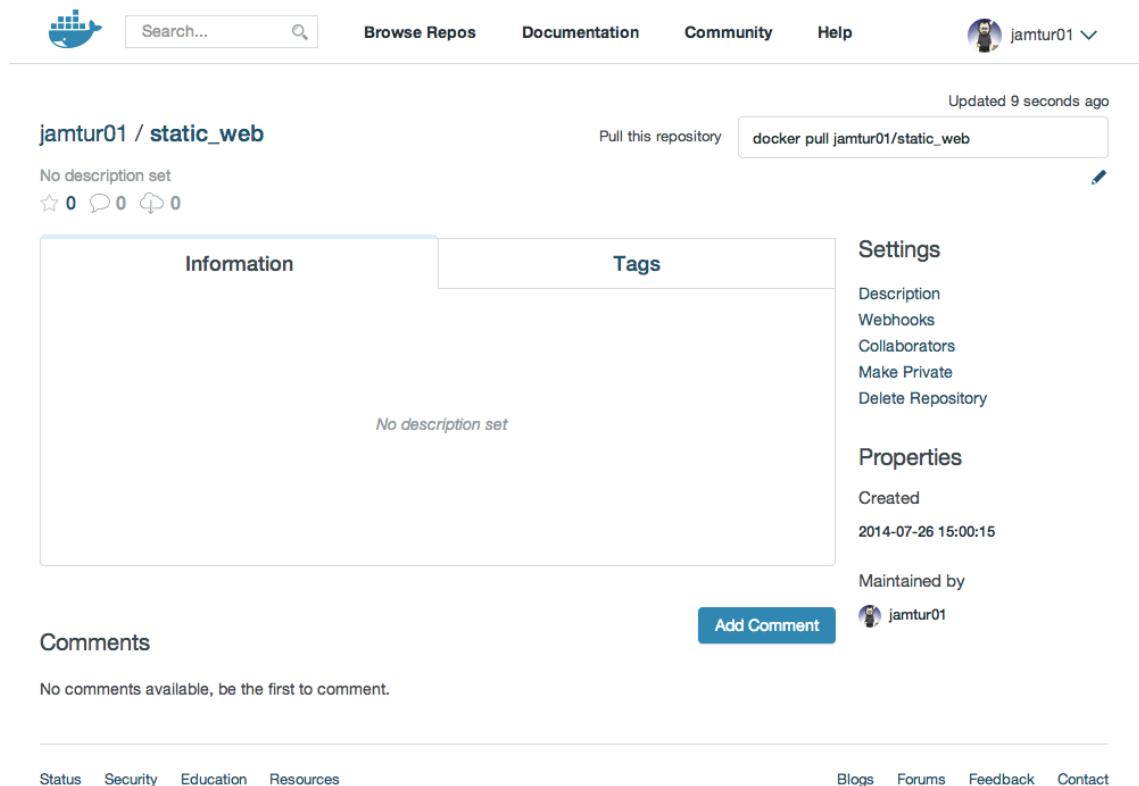


Figure 4.4: Your image on the Docker Hub.

TIP You can find documentation and more information on the features of the Docker Hub [here](#).

Automated Builds

In addition to being able to build and push our images from the command line, the Docker Hub also allows us to define Automated Builds. We can do so by connecting a [GitHub](#) or [BitBucket](#) repository containing a Dockerfile to the [Docker Hub](#). When we push to this repository, an image build will be triggered and a new image created. This was previously also known as a Trusted Build.

NOTE Automated Builds also work for private GitHub and BitBucket repositories.

The first step in adding an Automated Build to the Docker Hub is to connect your GitHub account or BitBucket to your Docker Hub account. To do this, navigate to Docker Hub, sign in, click on your profile link, then click the `Add Repository` \leftrightarrow `Automated Build` button.

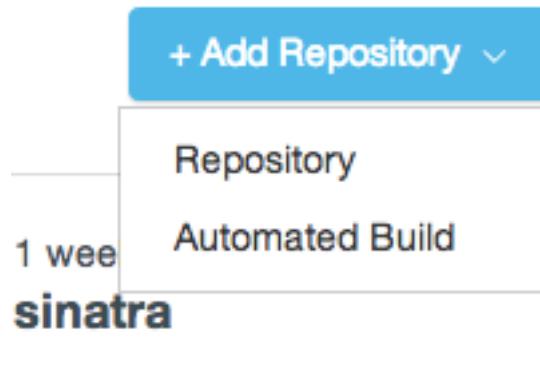


Figure 4.5: The Add Repository button.

You will see a page that shows your options for linking to either GitHub or BitBucket.

Chapter 4: Working with Docker images and repositories

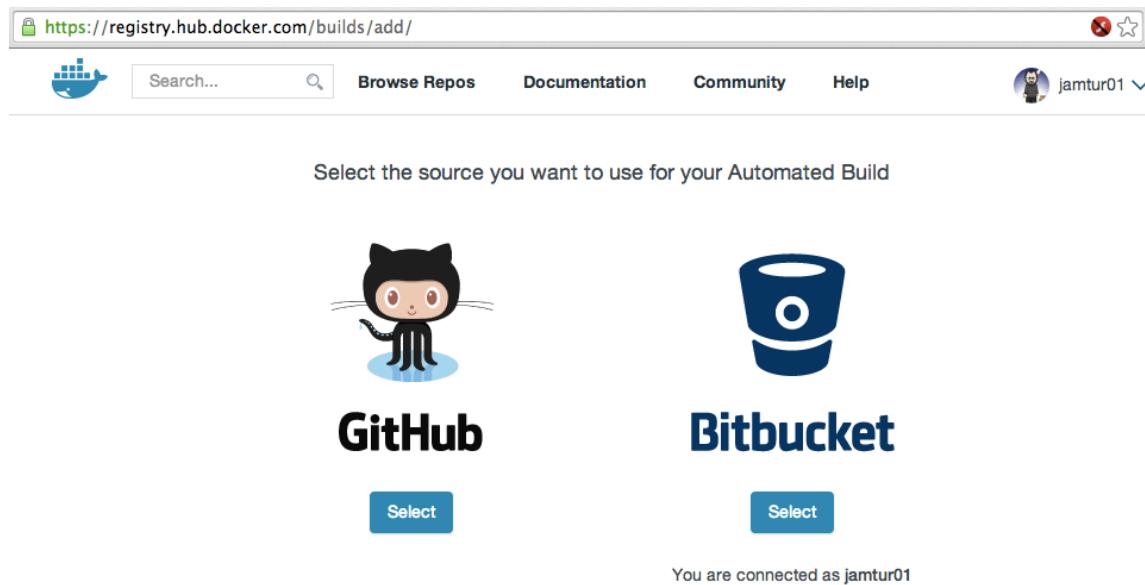


Figure 4.6: Account linking options.

Click the Select button under the GitHub logo to initiate the account linkage. You will be taken to GitHub and asked to authorize access for Docker Hub.

Chapter 4: Working with Docker images and repositories

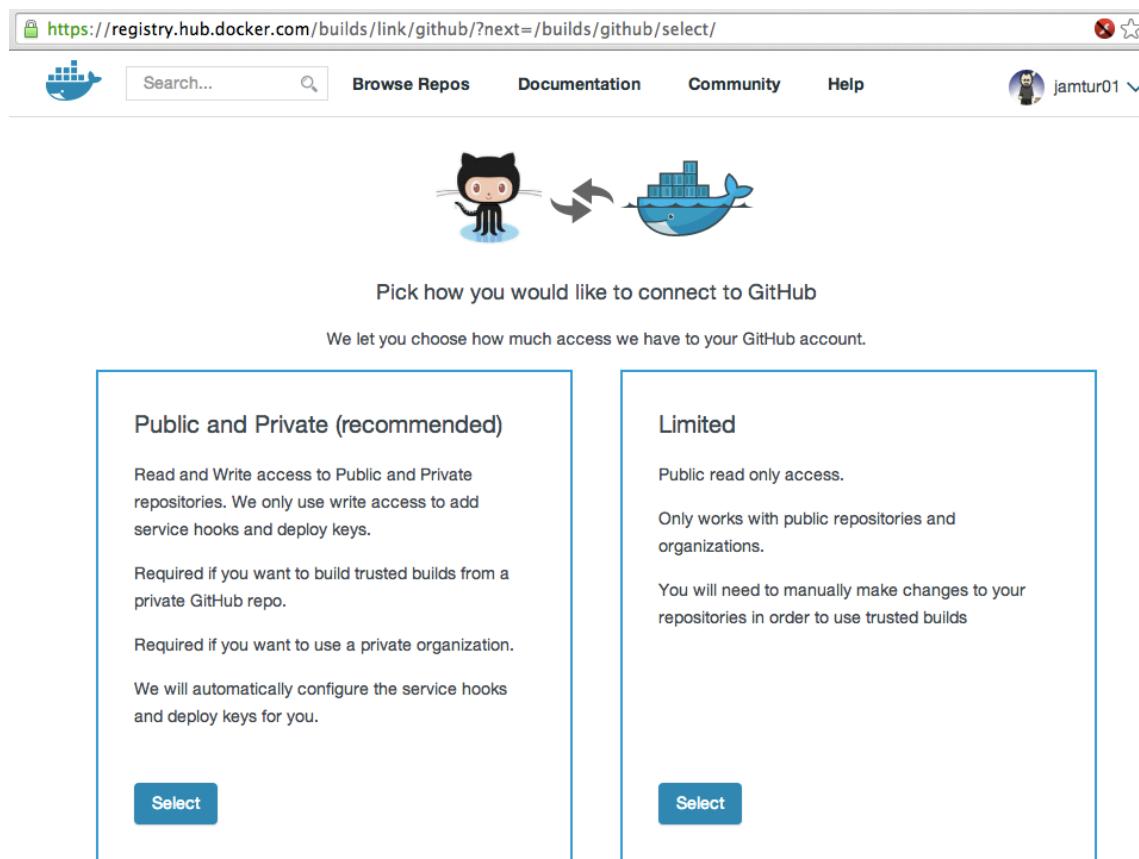
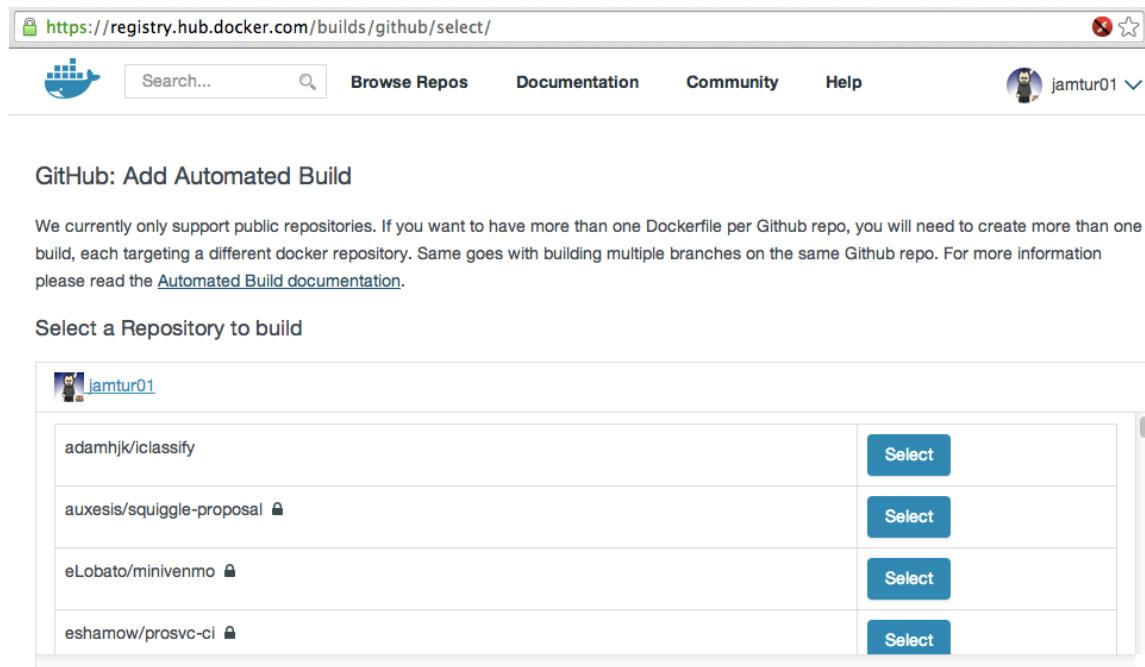


Figure 4.7: Linking your GitHub account

You have two options: **Public and Private (recommended)** and **Limited**. Select **Public and Private (recommended)**, and click **Allow Access** to complete the authorization. You may be prompted to input your GitHub password to confirm the access.

From here, you will be prompted to select the organization and repository from which you want to construct an Automated Build.

Chapter 4: Working with Docker images and repositories



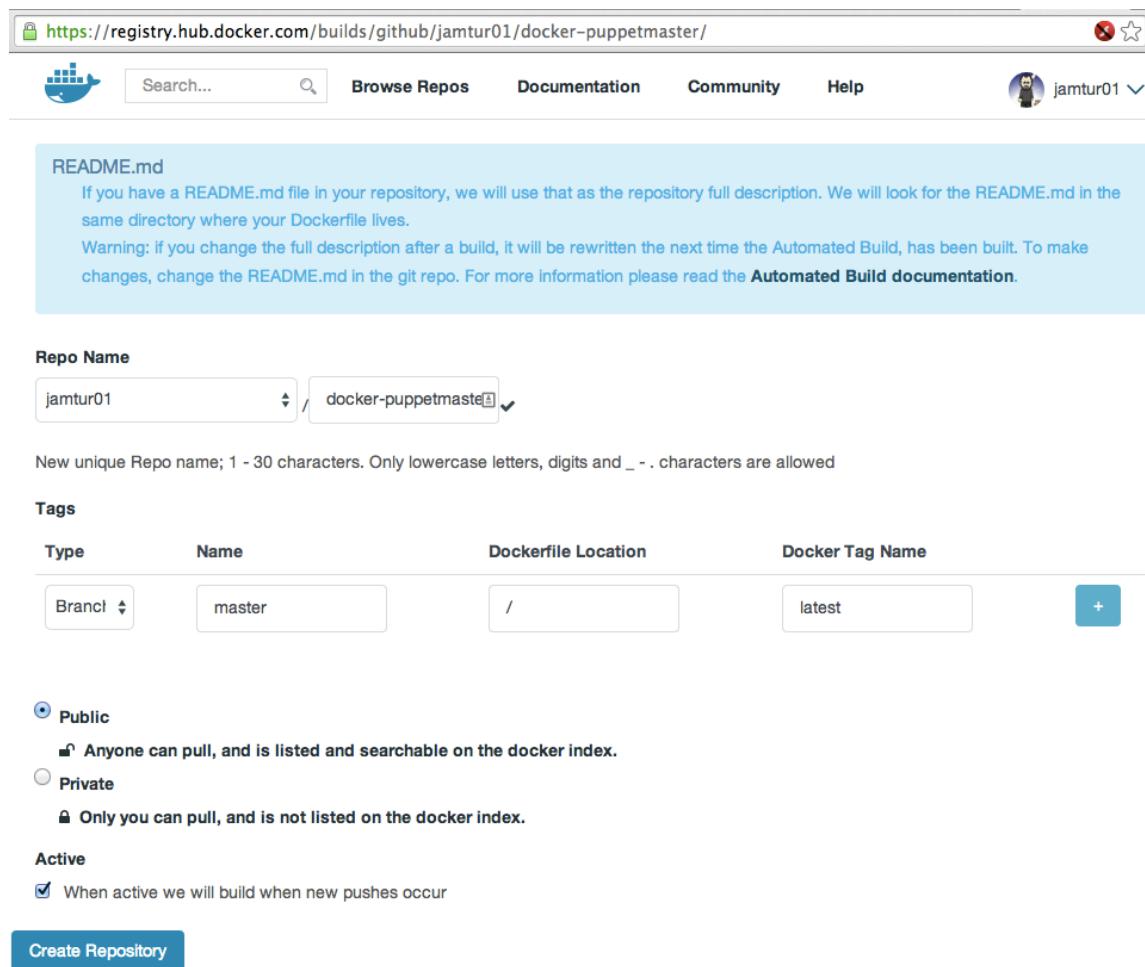
The screenshot shows a web browser window with the URL <https://registry.hub.docker.com/builds/github/select/>. The page title is "GitHub: Add Automated Build". The header includes a Docker logo, a search bar, and navigation links for "Browse Repos", "Documentation", "Community", and "Help". A user profile for "jamtur01" is visible on the right. The main content area is titled "Select a Repository to build" and shows a list of repositories under the user "jamtur01":

Repository	Select
adamhjk/iclassify	Select
auxesis/squiggle-proposal	Select
eLobato/minivenmo	Select
eshamow/prosvc-ci	Select

Figure 4.8: Selecting your repository.

Select the repository from which you wish to create an Automated Build by clicking the Select button next to the required repository, and then configure the build.

Chapter 4: Working with Docker images and repositories



The screenshot shows the Docker Hub automated build configuration interface. At the top, the URL is <https://registry.hub.docker.com/builds/github/jamtur01/docker-puppetmaster/>. The page includes a search bar, navigation links for 'Browse Repos', 'Documentation', 'Community', and 'Help', and a user profile for 'jamtur01'. A 'README.md' section provides instructions for using the README file. The 'Repo Name' field is set to 'jamtur01/docker-puppetmaster'. Below this, a note specifies a unique repository name of 1-30 characters. The 'Tags' section shows a table with columns 'Type', 'Name', 'Dockerfile Location', and 'Docker Tag Name'. A dropdown for 'Type' is set to 'Branch', and the 'Name' is 'master'. The 'Dockerfile Location' is '/'. The 'Docker Tag Name' is 'latest', with a '+' button for adding more. Under 'Visibility', 'Public' is selected, with a note that anyone can pull. Under 'Active', the 'When active we will build when new pushes occur' checkbox is checked. A 'Create Repository' button is at the bottom.

Figure 4.9: Configuring your Automated Build.

Specify the default branch you wish to use, and confirm the repository name.

Specify a tag you wish to apply to any resulting build, then specify the location of the Dockerfile. The default is assumed to be the root of the repository, but you can override this with any path.

Finally, click the Create Repository button to add your Automated Build to the Docker Hub.

Chapter 4: Working with Docker images and repositories

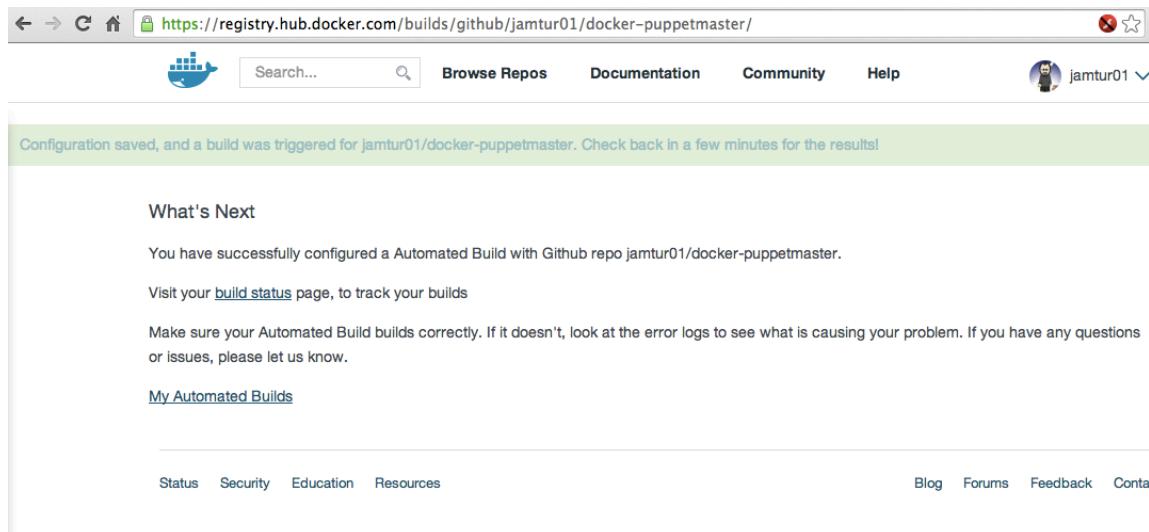


Figure 4.10: Creating your Automated Build.

You will now see your Automated Build submitted. Click on the [Build Status](#) link to see the status of the last build, including log output showing the build process and any errors. A build status of **Done** indicates the Automated Build is up to date. An **Error** status indicates a problem; you can click through to see the log output.

NOTE You can't push to an Automated Build using the `docker push` command. You can only update it by pushing updates to your GitHub or BitBucket repository.

Deleting an image

We can also delete images when we don't need them anymore. To do this, we'll use the `docker rmi` command.

Listing 4.78: Deleting a Docker image

```
$ sudo docker rmi jamtur01/static_web
Untagged: 06c6c1f81534
Deleted: 06c6c1f81534
Deleted: 9f551a68e60f
Deleted: 997485f46ec4
Deleted: a101d806d694
Deleted: 85130977028d
```

Here we've deleted the `jamtur01/static_web` image. You can see Docker's layer filesystem at work here: each of the `Deleted:` lines represents an image layer being deleted.

NOTE This only deletes the image locally. If you've previously pushed that image to the Docker Hub, it'll still exist there.

If you want to delete an image's repository on the Docker Hub, you'll need to sign in and [delete it there](#) using the `Delete repository` link.

Chapter 4: Working with Docker images and repositories

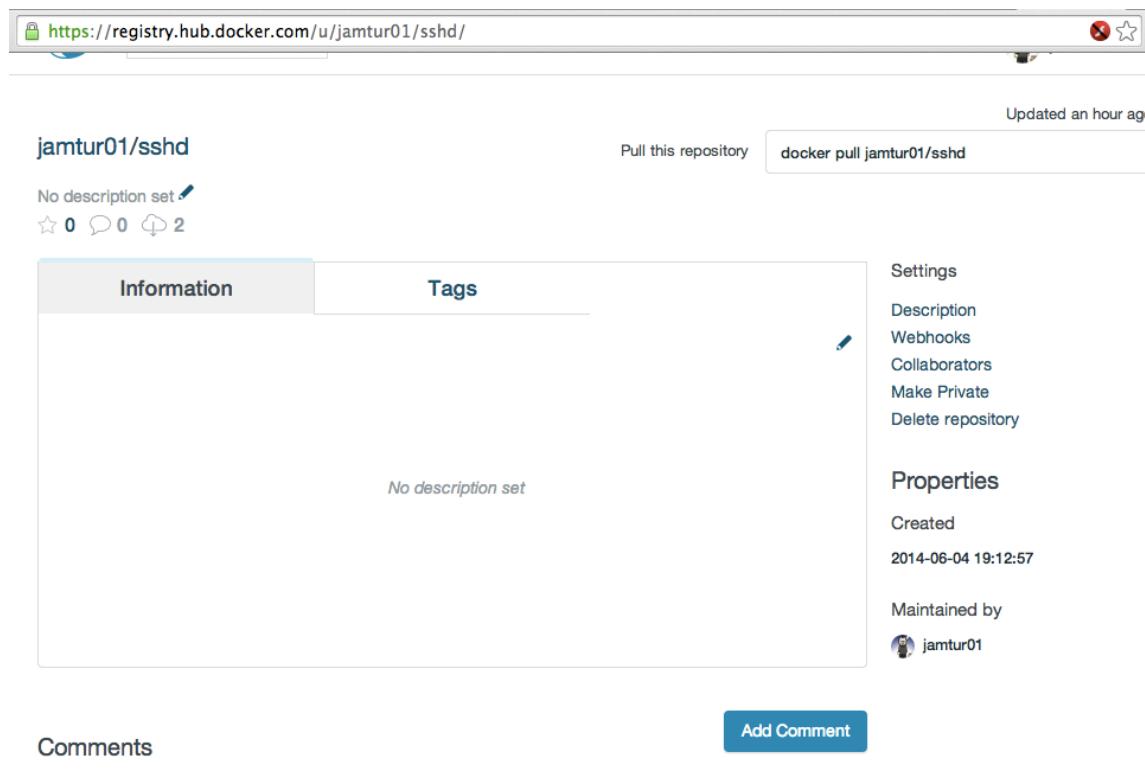


Figure 4.11: Deleting a repository.

We can also delete more than one image by specifying a list on the command line.

Listing 4.79: Deleting multiple Docker images

```
$ sudo docker rmi jamtur01/apache2 jamtur01/puppetmaster
```

or, like the `docker rm` command cheat we saw in Chapter 3, we can do the same with the `docker rmi` command:

Listing 4.80: Deleting all images

```
$ sudo docker rmi `docker images -a -q`
```

Running your own Docker registry

Obviously, having a public registry of Docker images is highly useful. Sometimes, however, we are going to want to build and store images that contain information or data that we don't want to make public. There are two choices in this situation:

- Make use of private [repositories on the Docker Hub](#).
- Run your own registry behind the firewall.

Thankfully, the team at Docker, Inc., have [open-sourced the code](#) they use to run a Docker registry, thus allowing us to build our own internal registry.

NOTE The registry does not currently have a user interface and is only made available as an API server.

Running a registry from a container

Installing a registry from a Docker container is very simple. Just run the Docker-provided container like so:

Listing 4.81: Running a container-based registry

```
$ sudo docker run -p 5000:5000 registry
```

This will launch a container running the registry application and bind port 5000 to the local host.

Testing the new registry

So how can we make use of our new registry? Let's see if we can upload one of our existing images, the `jamtur01/static_web` image, to our new registry. First, let's identify the image's ID using the `docker images` command.

Listing 4.82: Listing the jamtur01 static_web Docker image

```
$ sudo docker images jamtur01/static_web
REPOSITORY          TAG      ID      CREATED      SIZE
jamtur01/static_web  latest   22d47c8cb6e5  24 seconds ago  12.29 <-
                           kB (virtual 326 MB)
```

Next we take our image ID, 22d47c8cb6e5, and tag it for our new registry. To specify the new registry destination, we prefix the image name with the hostname and port of our new registry. In our case, our new registry has a hostname of docker.example.com.

Listing 4.83: Tagging our image for our new registry

```
$ sudo docker tag 22d47c8cb6e5 docker.example.com:5000/jamtur01/<-
                           static_web
```

After tagging our image, we can then push it to the new registry using the docker push command:

Listing 4.84: Pushing an image to our new registry

```
$ sudo docker push docker.example.com:5000/jamtur01/static_web
The push refers to a repository [docker.example.com:5000/jamtur01<-
                           /static_web] (len: 1)
Processing checksums
Sending image list
Pushing repository docker.example.com:5000/jamtur01/static_web (1<-
                           tags)
Pushing 22<-
                           d47c8cb6e556420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c
Buffering to disk 58375952/? (n/a)
Pushing 58.38 MB/58.38 MB (100%)
. . .
```

The image is then posted in the local registry and available for us to build new containers using the docker run command.

Listing 4.85: Building a container from our local registry

```
$ sudo docker run -t -i docker.example.com:5000/jamtur01/←  
static_web /bin/bash
```

This is the simplest deployment of the Docker registry behind your firewall. It doesn't explain how to configure the registry or manage it. To find out details like configuring authentication, how to manage the backend storage for your images and how to manage your registry see the full configuration and deployments details in the [Docker Registry documentation](#).

Alternative Indexes

There are a variety of other services and companies out there starting to provide custom Docker registry services.

Quay

The [Quay](#) service provides a private hosted registry that allows you to upload both public and private containers. Unlimited public repositories are currently free. Private repositories are available in a series of scaled plans. The Quay product has recently been acquired by [CoreOS](#) and will be integrated into that product.

Summary

In this chapter, we've seen how to use and interact with Docker images and the basics of modifying, updating, and uploading images to the Docker Index. We've also learned about using a Dockerfile to construct our own custom images. Finally, we've discovered how to run our own local Docker registry and some hosted alternatives. This gives us the basis for starting to build services with Docker.

We'll use this knowledge in the next chapter to see how we can integrate Docker into a testing workflow and into a Continuous Integration lifecycle.

Chapter 5

Testing with Docker

We've learned a lot about the basics of Docker in the previous chapters. We've learned about images, the basics of launching, and working with containers. Now that we've got those basics down, let's try to use Docker in earnest. We're going to start by using Docker to help us make our development and testing processes a bit more streamlined and efficient.

To demonstrate this, we're going to look at three use cases:

- Using Docker to test a static website.
- Using Docker to build and test a web application.
- Using Docker for Continuous Integration.

NOTE We're using Jenkins for CI because it's the platform I have the most experience with, but you can adapt most of the ideas contained in those sections to any CI platform.

In the first two use cases, we're going to focus on local, developer-centric developing and testing, and in the last use case, we'll see how Docker might be used in a broader multi-developer lifecycle for build and test.

This chapter will introduce you to using Docker as part of your daily life and workflow. It also contains a lot of useful information on how to run and manage Docker in general, and I recommend you read it even if these use cases aren't immediately relevant to you.

Using Docker to test a static website

One of the simplest use cases for Docker is as a local web development environment. An environment that allows you to replicate your production environment and ensure what you develop will run in the environment to which you want to deploy it. We're going to start with installing Nginx into a container to run a simple website. Our website is originally named Sample.

An initial Dockerfile for the Sample website

To do this, let's start with a simple Dockerfile. We start by creating a directory to hold our Dockerfile first.

Listing 5.1: Creating a directory for our Sample website Dockerfile

```
$ mkdir sample
$ cd sample
$ touch Dockerfile
```

We're also going to need some Nginx configuration files to run our website. Let's create a directory called `nginx` inside our `sample` directory to hold them. We can download some example files I've prepared earlier from Git Hub.

Listing 5.2: Getting our Nginx configuration files

```
$ cd sample
$ mkdir nginx && cd nginx
$ wget https://raw.githubusercontent.com/jamtur01/dockerbook-code<-
  /master/code/5/sample/nginx/global.conf
$ wget https://raw.githubusercontent.com/jamtur01/dockerbook-code<-
  /master/code/5/sample/nginx/nginx.conf
$ cd ..
```

Now let's look at the Dockerfile for our Sample website.

Listing 5.3: Our basic Dockerfile for the Sample website

```
FROM ubuntu:14.04
MAINTAINER James Turnbull "james@example.com"
ENV REFRESHED_AT 2014-06-01
RUN apt-get update
RUN apt-get -y -q install nginx
RUN mkdir -p /var/www/html
ADD nginx/global.conf /etc/nginx/conf.d/
ADD nginx/nginx.conf /etc/nginx/nginx.conf
EXPOSE 80
```

Here we've written a simple Dockerfile that:

- Installs Nginx.
- Creates a directory, `/var/www/html`, in the container.
- Adds the Nginx configuration from the local files we downloaded to our image.
- Exposes port 80 on the image.

Our two Nginx configuration files configure Nginx for running our Sample website. The `nginx/global.conf` file is copied into the `/etc/nginx/conf.d/` directory by the ADD instruction. The `global.conf` configuration file specifies:

Listing 5.4: The global.conf

```
server {  
    listen      0.0.0.0:80;  
    server_name _;  
  
    root        /var/www/html/website;  
    index       index.html index.htm;  
  
    access_log  /var/log/nginx/default_access.log;  
    error_log   /var/log/nginx/default_error.log;  
}
```

This sets Nginx to listen on port 80 and sets the root of our webserver to `/var/www/html/website`, the directory we just created with a `RUN` instruction.

We also need to configure Nginx to run non-daemonized in order to allow it to work inside our Docker container. To do this, the `nginx/nginx.conf` file is copied into the `/etc/nginx` directory and contains:

Listing 5.5: The nginx.conf configuration file

```

user www-data;
worker_processes 4;
pid /run/nginx.pid;
daemon off;

events { }

http {
    sendfile on;
    tcp_nopush on;
    tcp_nodelay on;
    keepalive_timeout 65;
    types_hash_max_size 2048;
    include /etc/nginx/mime.types;
    default_type application/octet-stream;
    access_log /var/log/nginx/access.log;
    error_log /var/log/nginx/error.log;
    gzip on;
    gzip_disable "msie6";
    include /etc/nginx/conf.d/*.conf;
}

```

In this configuration file, the `daemon off;` option stops Nginx from going into the background and forces it to run in the foreground. This is because Docker containers rely on the running process inside them to remain active. By default, Nginx daemonizes itself when started, which would cause the container to run briefly and then stop when the daemon was forked and launched and the original process that forked it stopped.

This file is copied to `/etc/nginx/nginx.conf` by the ADD instruction.

You'll also see a subtle difference between the destinations of the two ADD instructions. The first ends in the directory, `/etc/nginx/conf.d/`, and the second in a specific file `/etc/nginx/nginx.conf`. Both styles are accepted ways of copying

files into a Docker image.

NOTE You can find all the code and sample configuration files for this at [The Docker Book Code site](#) or the [GitHub repository](#). You will need to specifically download or copy and paste the `nginx.conf` and `global.conf` configuration files into the `nginx` directory we created to make them available for the `docker build`.

Building our Sample website and Nginx image

From this Dockerfile, we can build ourselves a new image with the `docker build` command; we'll call it `jamtur01/nginx`.

Listing 5.6: Building our new Nginx image

```
$ sudo docker build -t jamtur01/nginx .
```

This will build and name our new image, and you should see the build steps execute. We can take a look at the steps and layers that make up our new image using the `docker history` command.

Listing 5.7: Showing the history of the Nginx image

```
$ sudo docker history jamtur01/nginx
IMAGE          CREATED      CREATED BY ←
                SIZE
7eae7a24daba 2 minutes ago /bin/sh -c #(nop) EXPOSE map[80/tcp←
:{}]          0 B
bfea01c931f1 2 minutes ago /bin/sh -c #(nop) ADD file:5545063←
a4ee791201a 415 B
3dd97f7c6e01 2 minutes ago /bin/sh -c #(nop) ADD file:3←
c22f6ad1b04b40761 286 B
9e2ba0dbe0ce 2 minutes ago /bin/sh -c mkdir -p /var/www/html ←
0 B
a34fb588afa3 2 minutes ago /bin/sh -c apt-get -y -q install nginx←
18.43 MB
8df0d38229b7 3 minutes ago /bin/sh -c apt-get update ←
75.49 MB
51e7e1e3a370 3 minutes ago /bin/sh -c #(nop) ENV REFRESHED_AT←
=2014-06-01 0 B
2e4137b1b4ae 3 minutes ago /bin/sh -c #(nop) MAINTAINER James ←
Turnbull " 0 B
99ec81b80c55 6 weeks ago  /bin/sh -c apt-get update && apt-get ←
install 73.33 MB
d4010efcf86 6 weeks ago  /bin/sh -c sed -i 's/^#\|s*\|(deb.*←
universe|)$/ 1.903 kB
4d26dd3ebc1c 6 weeks ago  /bin/sh -c echo '#!/bin/sh' > /usr/←
sbin/polic 194.5 kB
5e66087f3ffe 6 weeks ago  /bin/sh -c #(nop) ADD file:175959←
bb3b959f73e9 192.5 MB
511136ea3c5a 11 months ago ←
0 B
```

The history starts with the final layer, our new `jamtur01/nginx`, image and works backward to the original parent image, `ubuntu:14.04`. Each step in between shows the new layer and the instruction from the Dockerfile that generated it.

Building containers from our Sample website and Nginx image

We can now take our `jamtur01/nginx` image and start to build containers from it, which will allow us to test our Sample website. Firstly, though we need that Sample website to test. Let's download some code for that now.

Listing 5.8: Downloading our Sample website

```
$ cd sample
$ mkdir website && cd website
$ wget https://raw.githubusercontent.com/jamtur01/dockerbook-code-
  /master/code/5/sample/website/index.html
$ cd ..
```

This will create a directory called `website` inside the `sample` directory. We then download an `index.html` file for our Sample website into that `website` directory.

Now let's look at how we might create a container using the `docker run` command.

Listing 5.9: Building our first Nginx testing container

```
$ sudo docker run -d -p 80 --name website \
  -v $PWD/website:/var/www/html/website \
  jamtur01/nginx nginx
```

NOTE You can see we've passed the `nginx` command to `docker run`. Normally this wouldn't make Nginx run interactively. In the configuration we supplied to Docker, though, we've added the directive `daemon off`. This directive causes Nginx to run interactively in the foreground when launched.

You can see we've used the `docker run` command to build a container from our `jamtur01/nginx` image called `website`. You will have seen most of the options before, but the `-v` option is new. This new option allows us to create a volume in our container from a directory on the host.

Let's take a brief digression into volumes, as they are important and useful in

Docker. Volumes are specially designated directories within one or more containers that bypass the layered Union File System to provide persistent or shared data for Docker. This means that changes to a volume are made directly and bypass the image. They will not be included when we commit or build an image.

TIP Volumes can also be shared between containers and can persist even when containers are stopped. We'll see how to make use of this for data management in later chapters.

In our immediate case, we can see the value of volumes when we don't want to bake our application or code into an image. For example:

- We want to work on and test it simultaneously.
- It changes frequently, and we don't want to rebuild the image during our development process.
- We want to share the code between multiple containers.

The `-v` option works by specifying a source directory or mount on the local host separated from the destination on the container with a `:`. If the destination directory doesn't exist Docker will create it.

We can also specify the read/write status of the destination by adding either `rw` or `ro` after that destination, like so:

Listing 5.10: Controlling the write status of a volume

```
$ sudo docker run -d -p 80 --name website \
-v $PWD/website:/var/www/html/website:ro \
jamtur01/nginx nginx
```

This would make the destination `/var/www/html/website` read-only.

In our Nginx website container, we've mounted a local website we're developing. To do this we've mounted, as a volume, the directory `$PWD/website` to `/var/www/html/website` in our container. In our Nginx configuration (in the `/etc/nginx` directory), we've specified the `root` directive to point to the mounted volume. This means that any changes we make to the local website will be reflected in the container.

`nginx/conf.d/global.conf` configuration file), we've specified this directory as the location to be served out by the Nginx server.

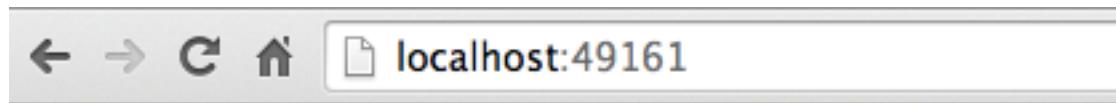
TIP The website directory we're using is contained in the source code for this book [here](#) and on [GitHub](#). You can see the `index.html` file we downloaded inside that directory.

Now, if we look at our running container using the `docker ps` command, we can see that it is active, it is named `website`, and port 80 is mapped to port 49161.

Listing 5.11: Viewing the Sample website container

```
$ sudo docker ps -l
CONTAINER ID  IMAGE          ... PORTS          ↪
 NAMES
6751b94bb5c0  jamtur01/nginx:latest ... 0.0.0.0:49161->80/tcp ↪
 website
```

If we browse to port 49161 on our Docker host, we'll be able to see our Sample website displayed.



This is a test website

Figure 5.1: Browsing the Sample website.

Editing our website

Neat! We've got a site live. Now what happens if we edit our website? Let's open up the `index.html` file in the `website` folder on our local host and edit it.

Listing 5.12: Editing our Sample website

```
$ vi $PWD/website/index.html
```

We'll change the title from:

Listing 5.13: Old title

```
This is a test website
```

To:

Listing 5.14: New title

```
This is a test website for Docker
```

Let's refresh our browser and see what we've got now.



Figure 5.2: Browsing the edited Sample website.

We can see that our Sample website has been updated. Obviously, this is an incredibly simple example of editing a website, but you can see how you could easily do so much more. More importantly, you're testing a site that reflects production reality. You can now have containers for each type of production web-serving environment (e.g., Apache, Nginx), for running varying versions of development frameworks like PHP or Ruby on Rails, or for database back ends, etc.

Using Docker to build and test a web application

Now let's look at a more complex example of testing a larger web application. We're going to test a Sinatra-based web application instead of a static website and then develop that application whilst testing in Docker. Our application is going to take incoming parameters and output them as a JSON hash.

Building our Sinatra application

Let's start with a `Dockerfile` to build the basic image that we will use to develop our Sinatra web application.

Listing 5.15: Dockerfile for web application testing

```
FROM ubuntu:14.04
MAINTAINER James Turnbull "james@example.com"
ENV REFRESHED_AT 2014-06-01

RUN apt-get update
RUN apt-get -y install ruby ruby-dev build-essential redis-tools
RUN gem install --no-rdoc --no-ri sinatra json redis

RUN mkdir -p /opt/webapp

EXPOSE 4567

CMD [ "/opt/webapp/bin/webapp" ]
```

You can see that we've created another Ubuntu-based image, installed Ruby and RubyGems, and then used the `gem` binary to install the `sinatra`, `json`, and `redis` gems. We've also created a directory to hold our new web application and exposed the default WEBrick port of 4567.

Finally, we've specified a `CMD` of `/opt/webapp/bin/webapp`, which will be the binary that launches our web application.

Let's build this new image now using the `docker build` command.

Listing 5.16: Building our new Sinatra image

```
$ sudo docker build -t jamtur01/sinatra .
```

Creating our Sinatra container

We've built our image. Let's now download our Sinatra web application's source code. You can find the code for this Sinatra application [here](#) or in the [GitHub repository](#). The application is made up of the `bin` and `lib` directories from the `webapp` directory.

Let's download it now.

Listing 5.17: Download our Sinatra web application

```
$ wget --cut-dirs=3 -nH -r --no-parent http://dockerbook.com/code-  
/5/sinatra/webapp/  
$ ls -l webapp  
..
```

We also need to ensure that the webapp/bin/webapp is executable prior to using it using the `chmod` command.

Listing 5.18: Making webapp/bin/webapp executable

```
$ chmod +x $PWD/webapp/bin/webapp
```

Now let's create a new container from our image using the `docker run` command.

Listing 5.19: Launching our first Sinatra container

```
$ sudo docker run -d -p 4567 --name webapp \  
-v $PWD/webapp:/opt/webapp jamtur01/sinatra
```

Here we've launched a new container from our `jamtur01/sinatra` image, called `webapp`. We've specified a new volume, `$PWD/webapp`, that holds our new Sinatra web application, and we've mounted it to the directory we created in the Dockerfile: `/opt/webapp`.

We've not provided a command to run on the command line; instead, we provided the command in the `CMD` in the Dockerfile of the image.

Listing 5.20: The `CMD` instruction in our Dockerfile

```
..  
CMD [ "/opt/webapp/bin/webapp" ]  
..
```

This command will be executed when a container is launched from this image.

We can also use the `docker logs` command to see what happened when our command was executed.

Listing 5.21: Checking the logs of our Sinatra container

```
$ sudo docker logs webapp
[2013-08-05 02:22:14] INFO  WEBrick 1.3.1
[2013-08-05 02:22:14] INFO  ruby 1.8.7 (2011-06-30) [x86_64-linux]
[2013-08-05 02:22:14] == Sinatra/1.4.3 has taken the stage on 4567 for development with-
[2013-08-05 02:22:14] == backup from WEBrick
[2013-08-05 02:22:14] INFO  WEBrick::HTTPServer#start: pid=1 port=4567
```

By adding the `-f` flag to the `docker logs` command, you can get similar behavior to the `tail -f` command and continuously stream new output from the `STDERR` and `STDOUT` of the container.

Listing 5.22: Tailing the logs of our Sinatra container

```
$ sudo docker logs -f webapp
. . .
```

We can also see the running processes of our Sinatra Docker container using the `docker top` command.

Listing 5.23: Using docker top to list our Sinatra processes

```
$ sudo docker top webapp
UID  PID  PPID  C  STIME  TTY  TIME      CMD
root 21506  15332  0  20:26  ?    00:00:00  /usr/bin/ruby /opt/webapp/bin/webapp
```

We can see from the logs that Sinatra has been launched and the WEBrick server is waiting on port 4567 in the container for us to test our application. Let's check to which port on our local host that port is mapped:

Listing 5.24: Checking the Sinatra port mapping

```
$ sudo docker port webapp 4567
0.0.0.0:49160
```

Right now, our basic Sinatra application doesn't do much. As we saw above, it just takes incoming parameters, turns them into JSON, and then outputs them. We can now use the `curl` command to test our application.

Listing 5.25: Testing our Sinatra application

```
$ curl -i -H 'Accept: application/json' \
-d 'name=Foo&status=Bar' http://localhost:49160/json
HTTP/1.1 200 OK
X-Content-Type-Options: nosniff
Content-Length: 29
X-Frame-Options: SAMEORIGIN
Connection: Keep-Alive
Date: Mon, 05 Aug 2013 02:22:21 GMT
Content-Type: text/html; charset=utf-8
Server: WEBrick/1.3.1 (Ruby/1.8.7/2011-06-30)
X-Xss-Protection: 1; mode=block
{"name":"Foo", "status": "Bar"}
```

We can see that we've passed some parameters to our Sinatra application and seen them returned to us as a JSON hash: `{"name": "Foo", "status": "Bar"}`.

Neat! But let's see if we can extend our example application container to an actual application stack by adding a service running in another container.

Building a Redis image and container

We're going to extend our Sinatra application now by adding a Redis back end and storing our incoming parameters in a Redis database. To do this, we're going to build a whole new image and container to run our Redis database, then we'll make use of Docker's capabilities to connect the two containers.

To build our Redis database, we're going to create a new image. We start with a new `Dockerfile` on which Redis will run.

Listing 5.26: Dockerfile for Redis image

```
FROM ubuntu:14.04
MAINTAINER James Turnbull "james@example.com"
ENV REFRESHED_AT 2014-06-01
RUN apt-get update
RUN apt-get -y install redis-server redis-tools
EXPOSE 6379
ENTRYPOINT [ "/usr/bin/redis-server" ]
CMD []
```

We've specified the installation of the Redis server, exposed port 6379, and specified an ENTRYPOINT that will launch that Redis server. Let's now build that image and call it `jamtur01/redis`.

Listing 5.27: Building our Redis image

```
$ sudo docker build -t jamtur01/redis .
```

Now let's create a container from our new image.

Listing 5.28: Launching a Redis container

```
$ sudo docker run -d -p 6379 --name redis jamtur01/redis
0a206261f079
```

We can see we've launched a new container named `redis` from our `jamtur01/redis` image. Note that we've specified the `-p` flag to expose port 6379. Let's see what port it's running on.

Listing 5.29: Launching a Redis container

```
$ sudo docker port redis 6379
0.0.0.0:49161
```

We can see our Redis port is exposed on port 49161. Let's try to connect to that Redis instance now.

We'll need to install the Redis client locally to do the test. This is usually the

`redis-tools` package.

Listing 5.30: Installing the `redis-tools` package on Ubuntu

```
$ sudo apt-get -y install redis-tools
```

Then we can use the `redis-cli` command to check our Redis server.

Listing 5.31: Testing our Redis connection

```
$ redis-cli -h 127.0.0.1 -p 49161
redis 127.0.0.1:49161>
```

Here we've connected the Redis client to 127.0.0.1 on port 49161 and verified that our Redis server is working.

Connecting to the Redis container

Let's now update our Sinatra application to connect to Redis and store our incoming parameters. In order to do that, we're going to need to be able to talk to the Redis server. There are several ways we could do this; let's explore and see the pros and cons of each.

The first method involves Docker's own network stack. So far, we've seen Docker containers exposing ports and binding interfaces so that container services are exposed on the local Docker host's external network (e.g., binding port 80 inside a container to a high port on the local host). In addition to this capability, Docker has a facet we haven't yet seen: internal networking.

Every Docker container is assigned an IP address, provided through an interface created when we installed Docker. That interface is called `docker0`. Let's look at that interface on our Docker host now.

Listing 5.32: The docker0 interface

```
$ ip a show docker0
4: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc ←
    noqueue state UP
        link/ether 06:41:69:71:00:ba brd ff:ff:ff:ff:ff:ff
        inet 172.17.42.1/16 scope global docker0
            inet6 fe80::1cb3:6eff:fee2:2df1/64 scope link
                valid_lft forever preferred_lft forever
        . . .
```

We can see that the docker0 interface has an RFC1918 private IP address in the 172.16-172.30 range. This address, 172.17.42.1, will be the gateway address for the Docker network and all our Docker containers.

TIP Docker will default to 172.17.x.x as a subnet unless that subnet is already in use, in which case it will try to acquire another in the 172.16-172.30 ranges.

The docker0 interface is a virtual Ethernet bridge that connects our containers and the local host network. If we look further at the other interfaces on our Docker host, we'll find a series of interfaces starting with veth.

Listing 5.33: The veth interfaces

```
vethec6a  Link encap:Ethernet  HWaddr 86:e1:95:da:e2:5a
          inet6 addr: fe80::84e1:95ff:fed:a25a/64 Scope:Link
          . . .
```

Every time Docker creates a container, it creates a pair of peer interfaces that are like opposite ends of a pipe (i.e., a packet send on one will be received on the other). It gives one of the peers to the container to become its eth0 interface and keeps the other peer, with a unique name like vethec6a, out on the host machine. You can think of a veth interface as one end of a virtual network cable. One end is plugged into the docker0 bridge, and the other end is plugged into the container.

By binding every `veth*` interface to the `docker0` bridge, Docker creates a virtual subnet shared between the host machine and every Docker container.

Let's look inside a container now and see the other end of this pipe.

Listing 5.34: The eth0 interface in a container

```
$ sudo docker run -t -i ubuntu /bin/bash
root@b9107458f16a:/# ip a show eth0
1483: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast ←
    state UP group default qlen 1000
    link/ether f2:1f:28:de:ee:a7 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.29/16 scope global eth0
        inet6 fe80::f01f:28ff:fed:ee7/64 scope link
            valid_lft forever preferred_lft forever
```

We can see that Docker has assigned an IP address, `172.17.0.29`, for our container that will be peered with a virtual interface on the host side, allowing communication between the host network and the container.

Let's trace a route out of our container and see this now.

Listing 5.35: Tracing a route out of our container

```
root@b9107458f16a:/# apt-get -yqq update && apt-get install -yqq ←
traceroute
. .
root@b9107458f16a:/# traceroute google.com
traceroute to google.com (74.125.228.78), 30 hops max, 60 byte ←
packets
1 172.17.42.1 (172.17.42.1) 0.078 ms 0.026 ms 0.024 ms
. .
15 iad23s07-in-f14.1e100.net (74.125.228.78) 32.272 ms 28.050 ←
ms 25.662 ms
```

We can see that the next hop from our container is the `docker0` interface gateway IP `172.17.42.1` on the host network.

But there's one other piece of Docker networking that enables this connectivity:

firewall rules and NAT configuration allow Docker to route between containers and the host network. Let's look at the IPTables NAT configuration on our Docker host.

Listing 5.36: Docker iptables and NAT

```
$ sudo iptables -t nat -L -n
Chain PREROUTING (policy ACCEPT)
target  prot opt source          destination
DOCKER  all  --  0.0.0.0/0    0.0.0.0/0      ADDRTYPE match dst-←
      type LOCAL

Chain OUTPUT (policy ACCEPT)
target  prot opt source          destination
DOCKER  all  --  0.0.0.0/0    !127.0.0.0/8  ADDRTYPE match dst-←
      type LOCAL

Chain POSTROUTING (policy ACCEPT)
target      prot opt source          destination
MASQUERADE all  --  172.17.0.0/16  !172.17.0.0/16

Chain DOCKER (2 references)
target  prot opt source          destination
DNAT    tcp  --  0.0.0.0/0    0.0.0.0/0      tcp dpt:49161 to←
      :172.17.0.18:6379
```

Here we have several interesting IPTables rules. Firstly, we can note that there is no default access into our containers. We specifically have to open up ports to communicate to them from the host network. We can see one example of this in the DNAT, or destination NAT, rule that routes traffic from our container to port 49161 on the Docker host.

TIP To learn more about advanced networking configuration for Docker, [this article is useful](#).

Our Redis connection

Let's examine our new Redis container and see its networking configuration using the `docker inspect` command.

Listing 5.37: Redis container's networking configuration

```
$ sudo docker inspect redis
...
"NetworkSettings": {
  "Bridge": "docker0",
  "Gateway": "172.17.42.1",
  "IPAddress": "172.17.0.18",
  "IPPrefixLen": 16,
  "PortMapping": null,
  "Ports": {
    "6379/tcp": [
      {
        "HostIp": "0.0.0.0",
        "HostPort": "49161"
      }
    ]
  }
},
...
```

The `docker inspect` command shows the details of a Docker container, including its configuration and networking. We've truncated much of this information in the example above and only shown the networking configuration. We could also use the `-f` flag to only acquire the IP address.

Listing 5.38: Finding the Redis container's IP address

```
$ sudo docker inspect -f '{{ .NetworkSettings.IPAddress }}' redis
172.17.0.18
```

We can see that the container has an IP address of 172.17.0.18 and uses the

gateway address of the `docker0` interface. We can also see that the 6379 port is mapped to port 49161 on the local host, but, because we're on the local Docker host, we don't have to use that port mapping. We can instead use the `172.17.0.18` address to communicate with the Redis server on port 6379 directly.

Listing 5.39: Talking directly to the Redis container

```
$ redis-cli -h 172.17.0.18
redis 172.17.0.18:6379>
```

NOTE Docker binds exposed ports on all interfaces by default; therefore, the Redis server will also be available on the `localhost` or `127.0.0.1`.

So, while this initially looks like it might be a good solution for connecting our containers together, sadly, this approach has two big rough edges: Firstly, we'd need to hard-code the IP address of our Redis container into our applications. Secondly, if we restart the container, Docker changes the IP address. Let's see this now using the `docker restart` command (we'll get the same result if we kill our container using the `docker kill` command).

Listing 5.40: Restarting our Redis container

```
$ sudo docker restart redis
```

Let's inspect its IP address.

Listing 5.41: Finding the restarted Redis container's IP address

```
$ sudo docker inspect -f '{{ .NetworkSettings.IPAddress }}' redis
172.17.0.19
```

We can see that our new Redis container has a new IP address, `172.17.0.19`, which means that if we'd hard-coded our Sinatra application, it would no longer be able to connect to the Redis database. That's not very helpful.

So what do we do instead? Thankfully, Docker comes with a useful feature called

links that allows us to link together one or more Docker containers and have them communicate.

Linking Docker containers

Linking one container to another is a simple process involving container names. Let's start by creating a new Redis container (or we could reuse the one we created earlier).

Listing 5.42: Starting another Redis container

```
$ sudo docker run -d --name redis jamtur01/redis
```

TIP Remember that container names are unique: if you recreate the container, you will need to delete the old `redis` container using the `docker rm` command before you can create another container called `redis`.

Here we've launched a Redis instance in our new container. We've named the new container `redis` using the `--name` flag.

NOTE You can also see that we've not exposed any ports on the container. The "why" of this will become clear shortly.

Now let's launch a container with our web application in it and link it to our new Redis container.

Listing 5.43: Linking our Redis container

```
$ sudo docker run -p 4567 \
--name webapp --link redis:db -t -i \
-v $PWD/webapp:/opt/webapp jamtur01/sinatra \
/bin/bash
root@811bd6d588cb:/#
```

TIP You'll have to stop and remove any previous `webapp` containers you have running with `docker rm`.

There's a lot going on in this command, so let's break it down. Firstly, we're exposing port 4567 using the `-p` flag so we can access our web application externally.

We've also named our container `webapp` using the `--name` flag and mounted our web application as a volume using the `-v` flag.

This time, however, we've used a new flag called `--link`. The `--link` flag creates a parent-child link between two containers. The flag takes two arguments: the container name to link and an alias for the link. In this case, we're creating a child relationship with the `redis` container with an alias of `db`. The alias allows us to consistently access the exposed information without needing to be concerned about the underlying container name. The link gives the parent container the ability to communicate with the child container and shares some connection details with it to help you configure applications to make use of the link.

We also get a security-related benefit from this linkage. You'll note that when we launched our Redis container, we didn't expose the Redis port with the `-p` flag. We don't need to. By linking the containers together, we're allowing the parent container to communicate to any open ports on the child container (i.e., our parent `webapp` container can connect to port 6379 on our child `redis` container). But even better, only containers explicitly linked to this container using the `--link` flag can connect to this port. Given that the port is not exposed to the local host, we now have a very strong security model for limiting the attack surface and network exposure of a containerized application.

TIP If you wish, for security reasons (for example), you can force Docker to only allow communication between containers if a link exists. To do this, you can start the Docker daemon with the `--icc=false` flag. This turns off communications between all containers unless a link exists.

You can link multiple containers together. For example, if we wanted to use our Redis instance for multiple web applications, we could link each web application container to the same `redis` container.

Listing 5.44: Linking our Redis container

```
$ sudo docker run -p 4567 --name webapp2 --link redis:db ...
...
$ sudo docker run -p 4567 --name webapp3 --link redis:db ...
...
```

TIP Container linking currently only works on a single Docker host. You can't link between containers on separate Docker hosts.

Finally, instead of running the container as a daemon, we've launched a shell. We've done this so we can see how our containers are now linked. Docker links populate information about the parent container in two places:

- The `/etc/hosts` file.
- Environmental variables that contain connection information.

Let's look first at the `/etc/hosts` file.

Listing 5.45: The webapp's /etc/hosts file

```
root@811bd6d588cb:/# cat /etc/hosts
172.17.0.33 811bd6d588cb
...
172.17.0.31 db
```

We can see that we have some useful entries in here. The first one is the container's own IP address and hostname (here, the short ID of the container). The second entry has been generated by our link; it's the IP address of the `redis` container with the hostname `db` derived from the link alias. Let's try and ping that container now.

TIP The container's hostname doesn't have to be the short ID. You can use the `-h` or `--hostname` flag with the `docker run` command to set a specific hostname for the container.

Listing 5.46: Pinging the db container

```
root@811bd6d588cb:/# ping db
PING db (172.17.0.31) 56(84) bytes of data.
64 bytes from db (172.17.0.31): icmp_seq=1 ttl=64 time=0.623 ms
64 bytes from db (172.17.0.31): icmp_seq=2 ttl=64 time=0.132 ms
64 bytes from db (172.17.0.31): icmp_seq=3 ttl=64 time=0.095 ms
64 bytes from db (172.17.0.31): icmp_seq=4 ttl=64 time=0.155 ms
...
```

TIP Remember how we mentioned that container IP addresses change when a container is restarted? Well since Docker version 1.3 if the linked container is restarted then the IP address in the `/etc/hosts` file will be updated with the new IP address.

We have connectivity to our Redis database, but before we make use of it, let's look at the other connection information contained in our environment variables.

Let's run the `env` command to see the environment variables.

Listing 5.47: Showing linked environment variables

```
root@811bd6d588cb:/# env
HOSTNAME=811bd6d588cb
DB_NAME=/webapp/db
DB_PORT_6379_TCP_PORT=6379
DB_PORT=tcp://172.17.0.31:6379
DB_PORT_6379_TCP=tcp://172.17.0.31:6379
DB_ENV_REFRESHED_AT=2014-06-01
DB_PORT_6379_TCP_ADDR=172.17.0.31
DB_PORT_6379_TCP_PROTO=tcp
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
REFRESHED_AT=2014-06-01
. . .
```

We can see a bunch of environment variables here, including some prefixed with `DB`. Docker automatically creates these variables when we link the `webapp` and `redis` containers. They start with `DB` because that is the alias we used when we created our link.

The automatically created environment variables include a variety of information:

- The name of the child container.
- The protocol, IP, and port of the service running in the container.
- Specific protocols, IP, and ports of various services running in the container.
- The values of any Docker-specified environment variables on the container.

The precise variables will vary from container to container depending on what is configured on that container (e.g., what has been defined by the `ENV` and `EXPOSE` instructions in the container's `Dockerfile`). More importantly, they include information we can use inside our applications to consistently link between containers.

Using our container link to communicate

So how can we make use of this link? Let's look at our Sinatra application and add some connection information for Redis. There are two ways we could connect the application to Redis:

- Using some of the connection information in our environment variables.
- Using DNS and the `/etc/hosts` information.

Let's look at the first method by seeing how our web app's `lib/app.rb` file might look using our new environment variables.

Listing 5.48: The linked via env variables Redis connection

```
require 'uri'
.
.
.
uri=URI.parse(ENV['DB_PORT'])
redis = Redis.new(:host => uri.host, :port => uri.port)
.
.
```

Here, we are parsing the `DB_PORT` environment variable using the Ruby `URI` module. We're then using the resulting host and port output to configure our Redis connection. Our application can now use this connection information to find Redis in a linked container. This abstracts away the need to hard-code an IP address and port to provide connectivity. It's a very crude form of service discovery.

Alternatively, we could use local DNS.

TIP You can also configure the DNS of your individual containers using the `--dns` and `--dns-search` flags on the `docker run` command. This allows you to set the local DNS resolution path and search domains. You can read about this [here](#). In the absence of both of these flags, Docker will set DNS resolution to match that of the Docker host. You can see the DNS resolution configuration in the `/etc/resolv.conf` file.

Listing 5.49: The linked via hosts Redis connection

```
redis = Redis.new(:host => 'db', :port => '6379')
```

NOTE You can see and download our updated Redis-enabled Sinatra application [here](#) or in the [GitHub repository](#). You'll need to download or edit the existing code for the example to work.

This will cause our application to look up the host db locally; it will find the entry in the /etc/hosts file and resolve the host to the correct IP address, again abstracting away the need to hard-code an IP address.

Let's try it with the DNS local resolution by starting our application inside the container.

Listing 5.50: Starting the Redis-enabled Sinatra application

```
root@811bd6d588cb:/# nohup /opt/webapp/bin/webapp &
nohup: ignoring input and appending output to 'nohup.out'
```

Here we've backgrounded our Sinatra application and started it. Now let's test our application from the Docker host using the `curl` command again.

Listing 5.51: Testing our Redis-enabled Sinatra application

```
$ curl -i -H 'Accept: application/json' \
-d 'name=Foo&status=Bar' http://localhost:49161/json
HTTP/1.1 200 OK
X-Content-Type-Options: nosniff
Content-Length: 29
X-Frame-Options: SAMEORIGIN
Connection: Keep-Alive
Date: Mon, 01 Jun 2014 02:22:21 GMT
Content-Type: text/html; charset=utf-8
Server: WEBrick/1.3.1 (Ruby/1.8.7/2011-06-30)
X-Xss-Protection: 1; mode=block
{"name": "Foo", "status": "Bar"}
```

And now let's confirm that our Redis instance has received the update.

Listing 5.52: Confirming Redis contains data

```
$ curl -i http://localhost:49161/json
[{"name": "Foo", "status": "Bar"}]
```

Here we've connected to our application, which has connected to Redis, checked a list of keys to find that we have a key called `params`, and then queried that key to see that our parameters (`name=Foo` and `status=Bar`) have both been stored in Redis. Our application works!

We now have a fully functional representation of our web application stack consisting of:

- A web server container running Sinatra.
- A Redis database container.
- A secure connection between the two containers.

You can see how easy it would be to extend this concept to provide any number of applications stacks and manage complex local development with them, like:

- Wordpress, HTML, CSS, JavaScript.
- Ruby on Rails.
- Django and Flask.
- Node.js.
- Play!
- Or any other framework that you like!

This way you can build, replicate, and iterate on production applications, even complex multi-tier applications, in your local environment.

Using Docker for continuous integration

Up until now, all our testing examples have been very local, single developer-centric examples (i.e., how a local developer might make use of Docker to test a local website or application). Let's look at using Docker's capabilities in a multi-developer [continuous integration](#) testing scenario.

Docker excels at quickly generating and disposing of one or multiple containers. There's an obvious synergy with Docker's capabilities and the concept of continuous integration testing. Often in a testing scenario you need to install software or deploy multiple hosts frequently, run your tests, and then clean up the hosts to be ready to run again.

In a continuous integration environment, you might need these installation steps and hosts multiple times a day. This adds a considerable build and configuration overhead to your testing lifecycle. Package and installation steps can also be time-consuming and annoying, especially if requirements change frequently or steps require complex or time-consuming processes to clean up or revert.

Docker makes the deployment and cleanup of these steps and hosts cheap. To demonstrate this, we're going to build a testing pipeline in stages using [Jenkins CI](#): Firstly, we're going to build a Jenkins server that also runs Docker. To make it even more interesting, we're going to be very recursive and run Docker INSIDE Docker. Turtles all the way down!

TIP You can read more about Docker-in-Docker [here](#).

Once we've got Jenkins running, we'll demonstrate a basic single-container test run. Finally, we'll look at a multi-container test scenario.

TIP There are a number of continuous integration tool alternatives to Jenkins, including [Strider](#) and [Drone.io](#), which actually makes use of Docker.

Build a Jenkins and Docker server

To provide our Jenkins server, we're going to build an Ubuntu 14.04 image from a Dockerfile that both installs Jenkins and Docker. Let's take a look at that Dockerfile now.

Listing 5.53: Jenkins and Docker Dockerfile

```
FROM ubuntu:14.04
MAINTAINER james@example.com
ENV REFRESHED_AT 2014-06-01

RUN apt-get update -qq && apt-get install -qq curl
RUN curl https://get.docker.io/gpg | apt-key add -
RUN echo deb http://get.docker.io/ubuntu docker main > /etc/apt/←
  sources.list.d/docker.list
RUN apt-get update -qq && apt-get install -qq iptables ca-←
  certificates lxc openjdk-6-jdk git-core lxc-docker

ENV JENKINS_HOME /opt/jenkins/data
ENV JENKINS_MIRROR http://mirrors.jenkins-ci.org

RUN mkdir -p $JENKINS_HOME/plugins
RUN curl -sf -o /opt/jenkins/jenkins.war -L $JENKINS_MIRROR/war-←
  stable/latest/jenkins.war

RUN for plugin in chucknorris greenballs scm-api git-client git ←
  ws-cleanup ;\
  do curl -sf -o $JENKINS_HOME/plugins/${plugin}.hpi \
  -L $JENKINS_MIRROR/plugins/${plugin}/latest/${plugin}.hpi ←
  ; done

ADD ./dockerjenkins.sh /usr/local/bin/dockerjenkins.sh
RUN chmod +x /usr/local/bin/dockerjenkins.sh

VOLUME /var/lib/docker

EXPOSE 8080

ENTRYPOINT [ "/usr/local/bin/dockerjenkins.sh" ]
```

We can see that our Dockerfile inherits from the `ubuntu:14.04` image and then does a lot of other stuff. Indeed, it is probably the most complex Dockerfile we've seen so far. Let's walk through what it does.

Firstly, it sets up the Ubuntu and Docker APT repositories we need and adds the Docker repository GPG key. We then update our package list and install the packages required to run both Docker and Jenkins. We've followed the same instructions that we used in Chapter 2 and added some additional packages we need for Jenkins.

Next, we've created a directory, `/opt/jenkins`, and downloaded the latest version of Jenkins into it. We also need some Jenkins plugins. Plugins provide support for additional capabilities for Jenkins (e.g., for Git version control).

We've also set the `JENKINS_HOME` and `JENKINS_MIRROR` environment variables to the location of our Jenkins data directory and mirror site using the `ENV` instruction.

We've then specified a `VOLUME` instruction. Remember, the `VOLUME` instruction adds a volume from the host launching the container. In this case, we're 'faking out' Docker and specifying `/var/lib/docker` as a volume. This is because the `/var/←lib/docker` directory is where Docker stores its containers. This location must be a real filesystem rather than a mount point like the layers in a Docker image.

So, using the `VOLUME` instruction, we tell the Docker daemon that we're going to be running inside our container to use the host's filesystem for its container storage. Hence, the `/var/lib/docker` directory of the nested Docker will live somewhere in the `/var/lib/docker/volumes` directory on the host system.

We've exposed the Jenkins's default port of 8080.

Lastly we've specified a [shell script](#) that will run when our container is launched. This shell script (specified as an `ENTRYPOINT` instruction) helps configure Docker on our host, enables Docker in Docker, starts the Docker daemon, and then launches Jenkins. There is a bit more information about why the shell script does what it does to allow Docker-in-Docker [here](#).

NOTE The Dockerfile and the shell script are available as part of this book's code [here](#) or in the [GitHub repository](#).

Now that we have our Dockerfile, let's build a new image using the `docker build` command.

Listing 5.54: Building our Docker-Jenkins image

```
$ sudo docker build -t jamtur01/dockerjenkins .
```

We've called our new image, somewhat unoriginally, `jamtur01/dockerjenkins`. We can now create a container from this image using the `docker run` command.

Listing 5.55: Running our Docker-Jenkins image

```
$ sudo docker run -p 8080:8080 --name jenkins --privileged \
-d jamtur01/dockerjenkins
190f5c6333576f017257b3348cf64dfcd370ac10721c1150986ab1db3e3221ff8
```

We can see that we've used one new flag, `--privileged`, to run this container. The `--privileged` flag is special and enables Docker's privileged mode. Privileged mode allows us to run containers with (almost) all the capabilities of their host machine, including kernel features and device access. This enables the special magic that allows us to run Docker inside Docker.

WARNING Running Docker in `--privileged` mode is a security risk. Containers with this enabled have root-level access to the Docker host. Ensure you appropriately secure your Docker host and only use a Docker host that is an appropriate trust domain or only runs containers with similar trust profiles.

We can also see that we've used the `-p` flag to expose port 8080 on port 8080 on the local host, which would normally be poor practice, but we're only going to run one Jenkins server.

We can see that our new container, `jenkins`, has been started. Let's check out its logs.

Listing 5.56: Checking the Docker Jenkins container logs

```
$ sudo docker logs jenkins
Running from: /opt/jenkins/jenkins.war
webroot: EnvVars.masterEnvVars.get("JENKINS_HOME")
Sep 8, 2013 12:53:01 AM winstome.Logger logInternal
INFO: Beginning extraction from war file
.
.
.
INFO: HTTP Listener started: port=8080
.
.
```

You can keep checking the logs, or run `docker logs` with the `-f` flag, until you see a message similar to:

Listing 5.57: Checking that Jenkins is up and running

```
INFO: Jenkins is fully up and running
```

Excellent. Our Jenkins server should now be available in your browser on port 8080, as we can see here:

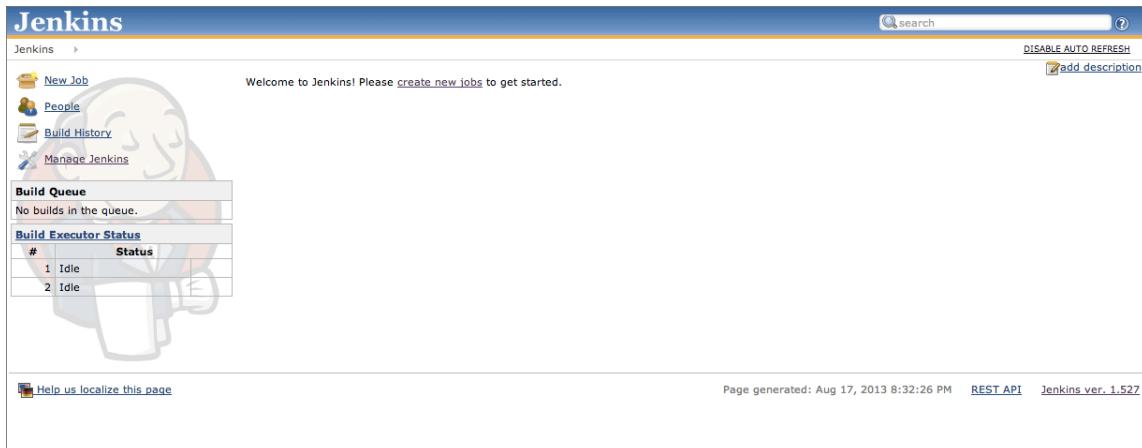


Figure 5.3: Browsing the Jenkins server.

Create a new Jenkins job

Now that we have a running Jenkins server, let's continue by creating a Jenkins job to run. To do this, we'll click the `create new jobs` link, which will open up the New Job wizard.

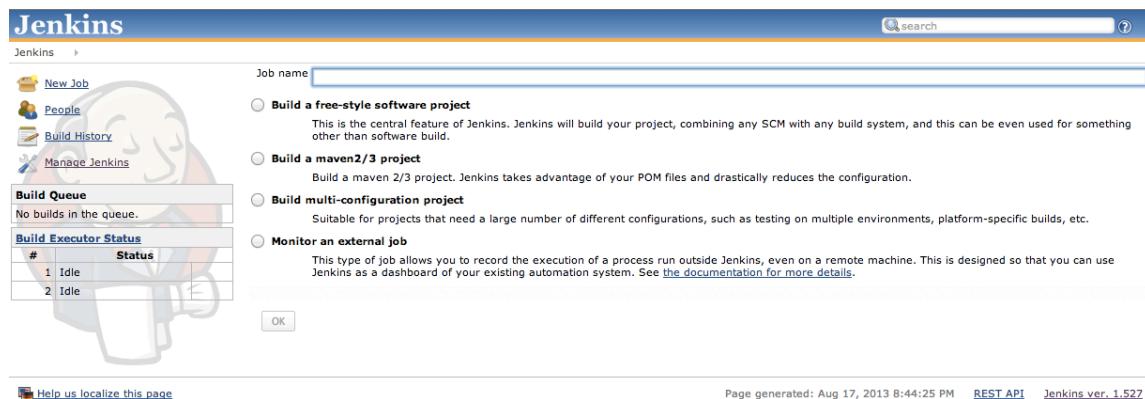


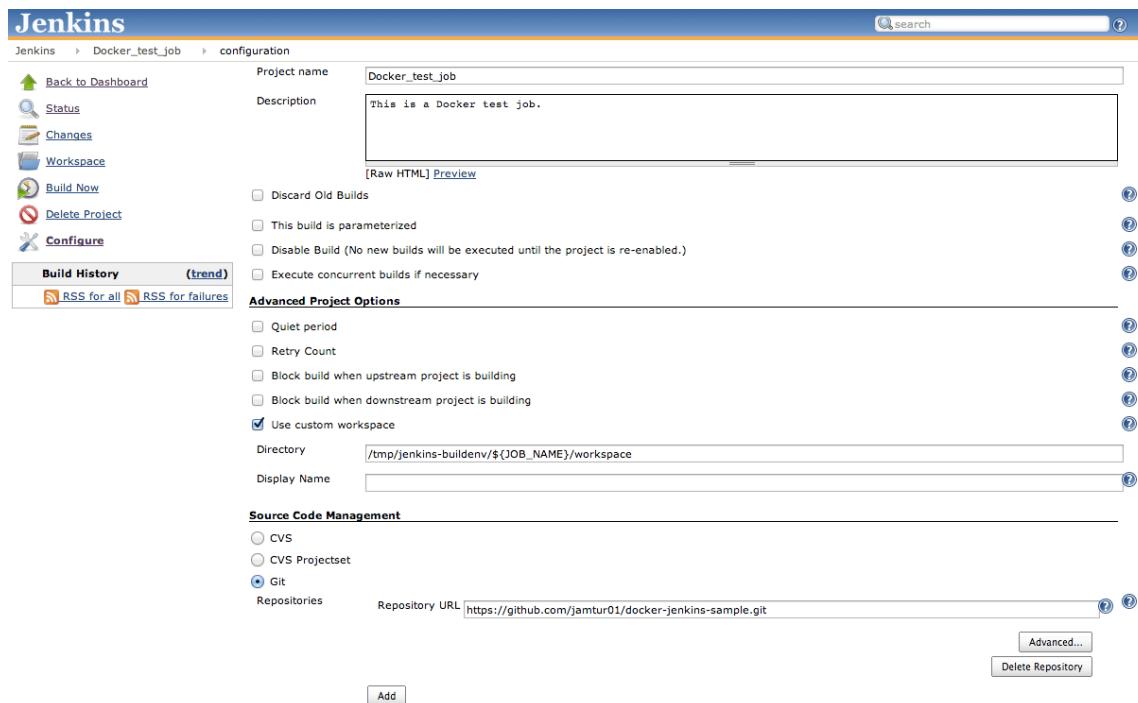
Figure 5.4: Creating a new Jenkins job.

Let's name our new job `Docker_test_job`, select a job type of `Build a free-style software project`, and click `OK` to continue to the next screen.

Now let's fill in a few sections. We'll start with a description of the job. Then click the `Advanced . . .` button under the `Advanced Project Options`, tick the `Use Custom workspace` radio button, and specify `/tmp/jenkins-buildenv/${JOB_NAME}/workspace` as the `Directory`. This is the workspace in which our Jenkins job is going to run.

Under `Source Code Management`, select `Git` and specify the following test repository: `https://github.com/jamtur01/docker-jenkins-sample.git`. This is a simple repository containing some Ruby-based RSpec tests.

Chapter 5: Testing with Docker



The screenshot shows the Jenkins configuration page for a job named 'Docker_test_job'. The job description is 'This is a Docker test job.' Under 'Advanced Project Options', the 'Use custom workspace' checkbox is checked. The workspace directory is set to '/tmp/jenkins-buildenv/\${JOB_NAME}/workspace'. Under 'Source Code Management', the 'Git' radio button is selected, and the repository URL is 'https://github.com/jamtur01/docker-jenkins-sample.git'. There are buttons for 'Advanced...', 'Delete Repository', and 'Add'.

Figure 5.5: Jenkins job details part 1.

Now we'll scroll down and update a few more fields. First, we'll add a build step by clicking the Add Build Step button and selecting Execute shell. Let's specify this shell script that will launch our tests and Docker.

Listing 5.58: The Docker shell script for Jenkins jobs

```
# Build the image to be used for this job.
IMAGE=$(docker build . | tail -1 | awk '{ print $NF }')

# Build the directory to be mounted into Docker.
MNT="$WORKSPACE/.."

# Execute the build inside Docker.
CONTAINER=$(docker run -d -v "$MNT:/opt/project" $IMAGE /bin/bash<-
-c 'cd /opt/project/workspace && rake spec')

# Attach to the container so that we can see the output.
docker attach $CONTAINER

# Get its exit code as soon as the container stops.
RC=$(docker wait $CONTAINER)

# Delete the container we've just used.
docker rm $CONTAINER

# Exit with the same value as that with which the process exited.
exit $RC
```

So what does this script do? Firstly, it will create a new Docker image using a Dockerfile contained in the Git repository we've just specified. This Dockerfile provides the test environment in which we wish to execute. Let's take a quick look at it now.

Listing 5.59: The Docker test job Dockerfile

```
FROM ubuntu:14.04
MAINTAINER James Turnbull "james@example.com"
ENV REFRESHED_AT 2014-06-01
RUN apt-get update
RUN apt-get -y install ruby rake
RUN gem install --no-rdoc --no-ri rspec ci_reporter_rspec
```

TIP If we add a new dependency or require another package to run our tests, all we'll need to do is update this Dockerfile with the new requirements, and the image will be automatically rebuilt when the tests are run.

As we can see, we're building an Ubuntu host, installing Ruby and RubyGems, and then installing two gems: `rspec` and `ci_reporter_rspec`. This will build an image that we can test using a typical Ruby-based application that relies on the RSpec test framework. The `ci_reporter_rspec` gem allows RSpec output to be converted to JUnit-formatted XML that Jenkins can consume. We'll see the results of this conversion shortly.

Back to our script. We're building an image from this Dockerfile. Next, we're creating a directory containing our Jenkins workspace (this is where the Git repository is checked out to), and it is this directory we're going to mount into our Docker container and from which we're going to run our tests.

Next we create a container from our image and run the tests. Inside this container, we've mounted our workspace to the `/opt/project` directory. We're also executing a command that changes into this directory and executes the `rake spec` command which actually runs our RSpec tests.

Now we've got a container started and we've grabbed the container ID.

TIP Docker also comes with a command line option called `--cidfile` that captures the container's ID and stores it in a file specified in the `--cidfile` options,

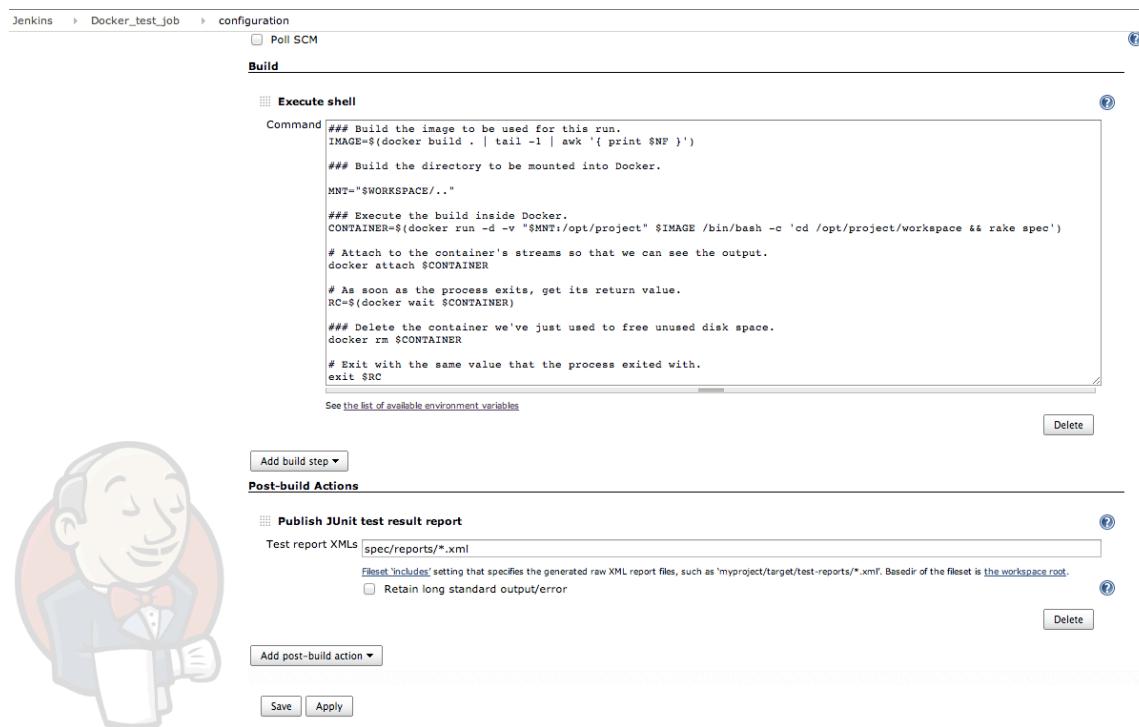
like so: `--cidfile=/tmp/containerid.txt`

Now we want to attach to that container to get the output from it using the `docker attach` command. and then use the `docker wait` command. The `docker wait` command blocks until the command the container is executing finishes and then returns the exit code of the container. The `RC` variable captures the exit code from the container when it completes.

Finally, we clean up and delete the container we've just created and exit with the container's exit code. This should be the exit code of our test run. Jenkins relies on this exit code to tell it if a job's tests have run successfully or failed.

Next we click the `Add post-build action` and add `Publish JUnit test result report`. In the `Test report XMLs`, we need to specify `spec/reports/*.xml`; this is the location of the `ci_reporter` gem's XML output, and locating it will allow Jenkins to consume our test history and output.

Finally, we must click the `Save` button to save our new job.



The screenshot shows the Jenkins job configuration for 'Docker_test_job'. The 'Build' section contains an 'Execute shell' step with the following command:

```
#!/bin/sh
### Build the image to be used for this run.
IMAGE=$(docker build . | tail -1 | awk '{ print $NF }')

### Build the directory to be mounted into Docker.
MNT="$WORKSPACE/.."

### Execute the build inside Docker.
CONTAINER=$(docker run -d -v "$MNT:/opt/project" $IMAGE /bin/bash -c 'cd /opt/project/workspace && rake spec')

# Attach to the container's streams so that we can see the output.
docker attach $CONTAINER

# As soon as the process exits, get its return value.
RC=$(docker wait $CONTAINER)

### Delete the container we've just used to free unused disk space.
docker rm $CONTAINER

# Exit with the same value that the process exited with.
exit $RC
```

The 'Post-build Actions' section contains a 'Publish JUnit test result report' step with the 'Test report XMLs' field set to 'spec/reports/*.xml'. The 'Save' and 'Apply' buttons are at the bottom.

Figure 5.6: Jenkins job details part 2.

Running our Jenkins job

We now have our Jenkins job, so let's run it. We'll do this by clicking the Build Now button; a job will appear in the Build History box.

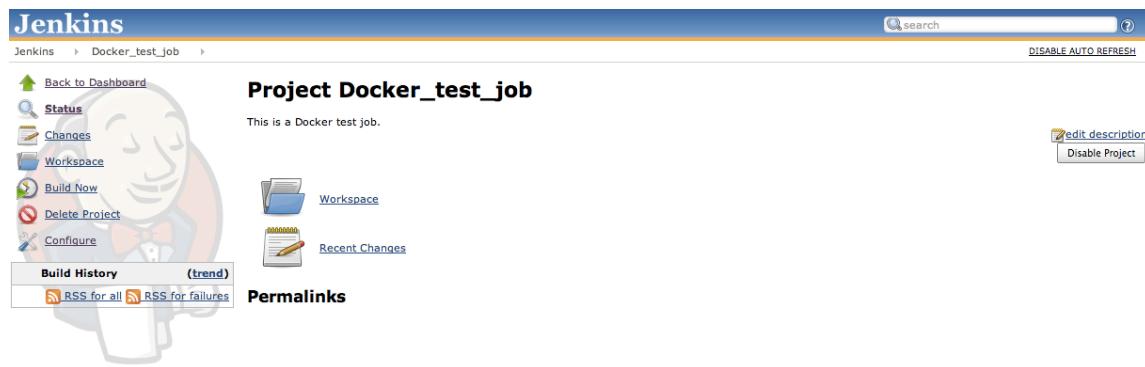


Figure 5.7: Running the Jenkins job.

NOTE The first time the tests run, it'll take a little longer because Docker is building our new image. The next time you run the tests, however, it'll be much faster, as Docker will already have the required image prepared.

We'll click on this job to get details of the test run we're executing.

Chapter 5: Testing with Docker



Jenkins

Jenkins > Docker_test_job > #30

search [DISABLE AUTO REFRESH](#)

[Back to Project](#) [Status](#) [Changes](#) [Console Output](#) [Edit Build Information](#) [Delete Build](#) [Git Build Data](#) [No Tags](#) [Test Result](#)

[Help us localize this page](#)

Build #30 (Aug 18, 2013 3:07:05 AM)

Started 1 min 44 sec ago
Took 4.2 sec

[add description](#)

No changes.

Started by anonymous user

Revision: 5491972404ee395ea12b4696c464cabdb0074575

git

- origin/HEAD
- origin/master

Test Result (no failures)

Figure 5.8: The Jenkins job details.

We can click on **Console Output** to see the commands that have been executed as part of the job.

The screenshot shows a Jenkins interface with the following details:

- Project Path:** Jenkins > Docker_test_job > #30
- Console Output:** The main content area displays the build logs. The logs show a successful build process using Docker, including fetching changes from a remote Git repository, building the Docker image, and running tests. The output ends with a "Finished: SUCCESS" message.
- Left Sidebar:** A sidebar on the left contains links for "Back to Project", "Status", "Changes", "Console Output" (which is currently selected), "View as plain text", "Edit Build Information", "Delete Build", "Git Build Data", "No Tags", and "Test Result".
- Search Bar:** A search bar is located at the top right of the page.

Figure 5.9: The Jenkins job console output.

We can see that Jenkins has downloaded our Git repository to the workspace. We can then execute our Shell script and build a Docker image using the `docker build` command. Then, we'll capture the image ID and use it to build a new

container using the `docker run` command. Running this new container executes the RSpec tests and captures the results of the tests and the exit code. If the job exits with an exit code of 0, then the job will be marked as successful.

You can also view the precise test results by clicking the `Test Result` link. This will have captured the RSpec output of our tests in JUnit form. This is the output that the `ci_reporter` gem produces and our `After Build` step captures.

Next steps with our Jenkins job

We can also automate our Jenkins job further by [enabling SCM polling](#), which triggers automatic builds when new commits are made to the repository. Similar automation can be achieved with a post-commit hook or via a GitHub or Bitbucket repository hook.

Summary of our Jenkins setup

We've achieved a lot so far: we've installed Jenkins, run it, and created our first job. This Jenkins job uses Docker to create an image that we can manage and keep updated using the Dockerfile contained in our repository. In this scenario, not only does our infrastructure configuration live with our code, but managing that configuration becomes a simple process. Containers are then created (from that image) in which we then run our tests. When we're done with the tests, we can dispose of the containers, which makes our testing fast and lightweight. It is also easy to adapt this example to test on different platforms or using different test frameworks for numerous languages.

TIP You could also use [parameterized builds](#) to make this job and the shell script step more generic to suit multiple frameworks and languages.

Multi-configuration Jenkins

We've now seen a simple, single container build using Jenkins. What if we wanted to test our application on multiple platforms? Let's say we'd like to test it on Ubuntu, Debian, and CentOS. To do that, we can take advantage of a Jenkins job type called a "multi-configuration job" that allows a matrix of test jobs to be run. When the Jenkins multi-configuration job is run, it will spawn multiple sub-jobs that will test varying configurations.

Create a multi-configuration job

Let's look at creating our new multi-configuration job. Click on the `New Job` from the Jenkins console. We're going to name our new job `Docker_matrix_job`, select `Build multi-configuration project`, and click `OK`.



Figure 5.10: Creating a multi-configuration job.

We'll see a screen that is very similar to the job creation screen we saw earlier. Let's add a description for our job, select Git as our repository type, and specify our sample application repository: <https://github.com/jamtur01/docker-←jenkins-sample.git>.

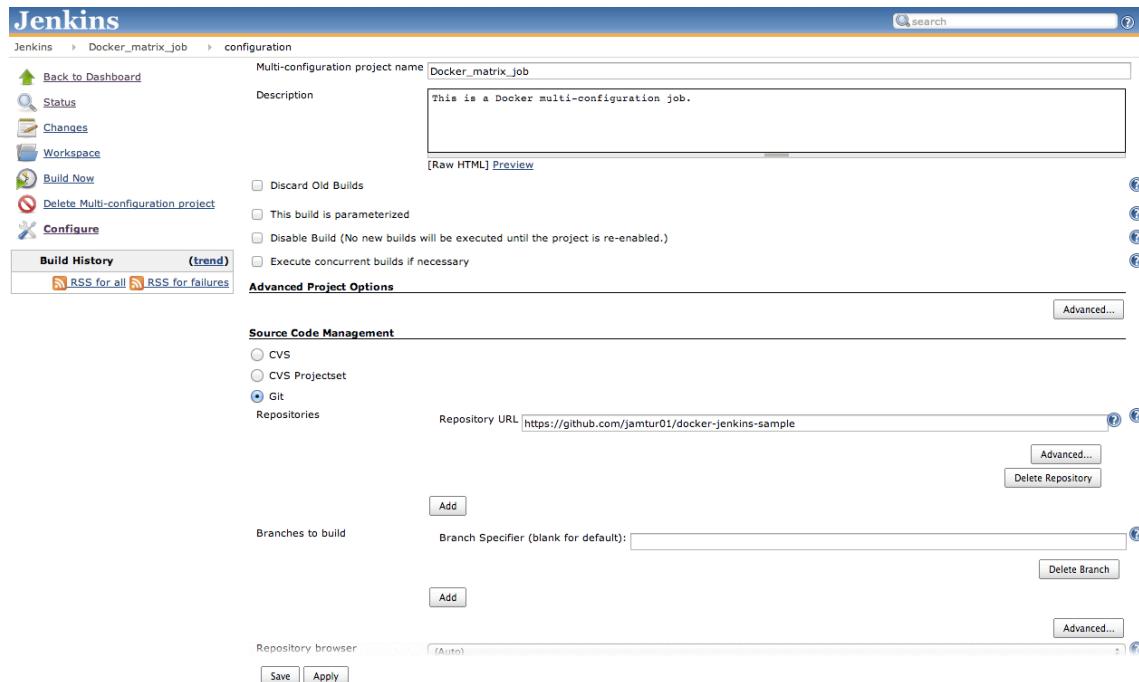
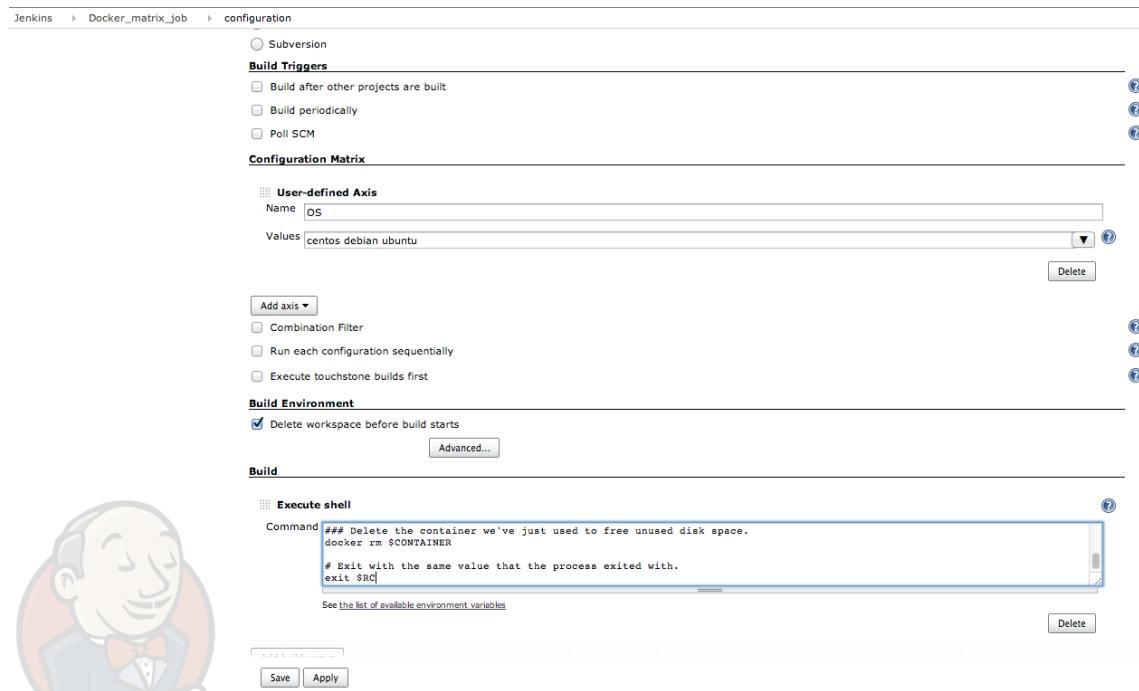


Figure 5.11: Configuring a multi-configuration job Part 1.

Next, let's scroll down and configure our multi-configuration axis. The axis is the list of matrix elements that we're going to execute as part of the job. We'll click the **Add Axis** button and select **User-defined Axis**. We're going to specify an axis named **OS** (which will be short for operating system) and specify three values: **centos**, **debian**, and **ubuntu**. When we execute our multi-configuration job, Jenkins will look at this axis and spawn three jobs: one for each point on the axis.

You'll also note that in the **Build Environment** section we've ticked **Delete workspace before build starts**. This option cleans up our build environment by deleting the checked-out repository prior to initiating a new set of jobs.

Chapter 5: Testing with Docker



The screenshot shows the Jenkins configuration interface for a job named 'Docker_matrix_job'. The 'configuration' tab is selected. The interface is divided into several sections:

- Build Triggers:** Subversion is selected. Other options include 'Build after other projects are built', 'Build periodically', and 'Poll SCM'.
- Configuration Matrix:** A 'User-defined Axis' named 'OS' is defined with values 'centos', 'debian', and 'ubuntu'. Buttons for 'Add axis', 'Combination Filter', 'Run each configuration sequentially', and 'Execute touchstone builds first' are available.
- Build Environment:** A checkbox for 'Delete workspace before build starts' is checked. An 'Advanced...' button is present.
- Build:** An 'Execute shell' step is defined with the following command:

```
### Delete the container we've just used to free unused disk space.
docker rm $CONTAINER
# Exit with the same value that the process exited with.
exit $RC
```

A link 'See the list of available environment variables' is provided. Buttons for 'Save' and 'Apply' are at the bottom.

Figure 5.12: Configuring a multi-configuration job Part 2.

Lastly, we've specified another shell build step with a simple shell script. It's a modification of the shell script we used earlier.

Listing 5.60: Jenkins multi-configuration shell step

```

# Build the image to be used for this run.
cd $OS && IMAGE=$(docker build . | tail -1 | awk '{ print $NF }')

# Build the directory to be mounted into Docker.
MNT="$WORKSPACE/.."

# Execute the build inside Docker.
CONTAINER=$(docker run -d -v "$MNT:/opt/project" $IMAGE /bin/bash<-
-c "cd /opt/project/$OS && rake spec")

# Attach to the container's streams so that we can see the output
.
docker attach $CONTAINER

# As soon as the process exits, get its return value.
RC=$(docker wait $CONTAINER)

# Delete the container we've just used.
docker rm $CONTAINER

# Exit with the same value as that with which the process exited.
exit $RC

```

We can see that this script has a modification: we're changing into directories named for each operating system for which we're executing a job. We can see inside our test repository that we have three directories: centos, debian, and ubuntu. Inside each directory is a different Dockerfile containing the build instructions for a CentOS, Debian, or Ubuntu image, respectively. This means that each job that is started will change into the appropriate directory for the required operating system, build an image based on that operating system, install any required prerequisites, and launch a container based on that image in which to run our tests.

Let's look at one of these new Dockerfile examples.

Listing 5.61: Our CentOS-based Dockerfile

```
FROM centos:latest
MAINTAINER James Turnbull "james@example.com"
ENV REFRESHED_AT 2014-06-01
RUN yum -y install ruby rubygems rubygem-rake
RUN gem install --no-rdoc --no-ri rspec ci_reporter_rspec
```

We can see that this is a CentOS-based variant of the Dockerfile we were using as a basis of our previous job. It basically performs the same tasks as that previous Dockerfile did, but uses the CentOS-appropriate commands like `yum` to install packages.

We're also going to add a post-build action of `Publish JUnit test result` ← `report` and specify the location of our XML output: `spec/reports/*.xml`. This will allow us to check the test result output.

Finally, we'll click `Save` to create our new job and save our proposed configuration. We can now see our freshly created job and note that it includes a section called `Configurations` that contains sub-jobs for each element of our axis.



Figure 5.13: Our Jenkins multi-configuration job

Testing our multi-configuration job

Now let's test this new job. We can launch our new multi-configuration job by clicking the Build Now button. When Jenkins runs, it will create a master job. This master job will, in turn, generate three sub-jobs that execute our tests on each of the three platforms we've chosen.

NOTE Like our previous job, it may take a little time to run the first time, as it builds the required images in which we'll test. Once they are built, though, the next runs should be much faster. Docker will only change the image if you update the Dockerfile.

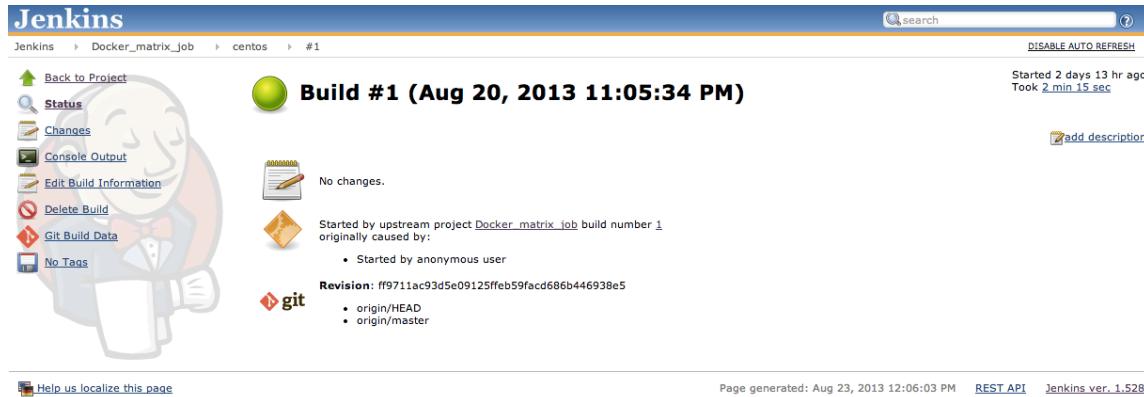
We can see that the master job executes first, and then each sub-job executes. Let's look at the output of one of these sub-jobs, our new centos job.



Figure 5.14: The centos sub-job.

We can see that it has executed: the green ball tells us it executed successfully. We can drill down into its execution to see more. To do so, click on the #1 entry in the Build History.

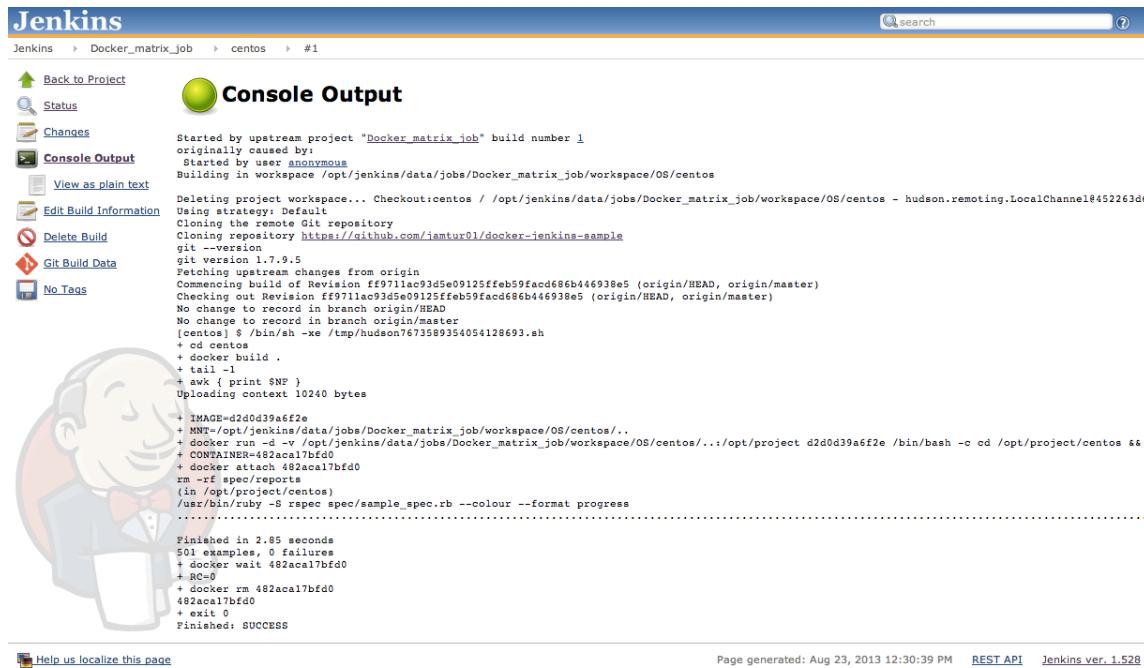
Chapter 5: Testing with Docker



The screenshot shows the Jenkins interface for a job named 'centos' under the 'Docker_matrix_job' project. The build number is #1, and it was started on Aug 20, 2013, at 11:05:34 PM. The status is 'No changes.' and it was started by an anonymous user. The revision is ff9711ac93d5e09125ffeb59facd686b446938e5, with branches 'origin/HEAD' and 'origin/master'. The page includes a search bar, a 'Disable Auto Refresh' button, and links for 'add description', 'REST API', and 'Jenkins ver. 1.528'.

Figure 5.15: The centos sub-job details.

Here we can see some more details of the executed centos job. We can see that the job has been Started by upstream project Docker_matrix_job and is build number 1. To see the exact details of what happened during the run, we can check the console output by clicking the Console Output link.



The screenshot shows the Jenkins console output for the centos job. It starts with the message 'Started by upstream project "Docker_matrix_job" build number 1' and 'originally caused by'. It then shows the build process, including cloning the repository from https://github.com/jamtur01/docker-jenkins-sample, fetching upstream changes, and performing a docker build. The output ends with 'Finished: SUCCESS'. The page includes a search bar, a 'Disable Auto Refresh' button, and links for 'add description', 'REST API', and 'Jenkins ver. 1.528'.

Figure 5.16: The centos sub-job console output.

We can see that the job cloned the repository, built the required Docker image, spawned a container from that image, and then ran the required tests. All of the tests passed successfully (we can also check the `Test Result` link for the uploaded JUnit test results if required).

We've now successfully completed a very simple, but powerful example of a multi-platform testing job for an application.

Summary of our multi-configuration Jenkins

These examples show very simplistic implementations of Jenkins CI working with Docker. You can enhance both of the examples shown with a lot of additional capabilities ranging from automated, triggered builds to multi-level job matrices using combinations of platform, architecture, and versions. Our simple Shell build step could also be rewritten in a number of ways to make it more sophisticated or to further support multi-container execution (e.g., to provide separate containers for web, database, or application layers to better simulate an actual multi-tier production application).

Other alternatives

One of the more interesting parts of the Docker ecosystem is continuous integration and continuous deployment (CI/CD). Beyond integration with existing tools like Jenkins, we're also seeing people build their own tools and integrations on top of Docker.

Drone

One of the more promising CI/CD tools being developed on top of Docker is [Drone](#). Drone is a SAAS continuous integration platform that connects to GitHub, Bitbucket, and Google Code repositories written in a wide variety of languages, including Python, Node.js, Ruby, Go, and numerous others. It runs the test suites of repositories added to it inside a Docker container.

Shippable

[Shippable](#) is a free, hosted continuous integration and deployment service for GitHub and Bitbucket. It is blazing fast and lightweight, and it supports Docker natively.

Summary

In this chapter, we've seen how to use Docker as a core part of our development and testing workflow. We've looked at developer-centric testing with Docker on a local workstation or virtual machine. We've also explored scaling that testing up to a continuous integration model using Jenkins CI as our tool. We've seen how to use Docker for both point testing and how to build distributed matrix jobs.

In the next chapter, we'll start to see how we can use Docker in production to provide containerized, stackable, scalable, and resilient services.

Chapter 6

Building services with Docker

In Chapter 5, we saw how to use Docker to facilitate better testing by using containers in our local development workflow and in a continuous integration environment. In this chapter, we're going to explore using Docker to run production services.

We're going to build a simple application first and then build some more complex multi-container applications. We'll explore how to make use of Docker features like links and volumes to combine and manage applications running in Docker.

Building our first application

The first application we're going to build is an on-demand website using the [Jekyll framework](#). We're going to build two images:

- An image that both installs Jekyll and the prerequisites we'll need and builds our Jekyll site.
- An image that serves our Jekyll site via Apache.

We're going to make it on demand by creating a new Jekyll site when a new container is launched. Our workflow is going to be:

- Create the Jekyll base image and the Apache image (once-off).
- Create a container from our Jekyll image that holds our website source mounted via a volume.
- Create a Docker container from our Apache image that uses the volume containing the compiled site and serve that out.
- Rinse and repeat as the site needs to be updated.

You could consider this a simple way to create multiple hosted website instances. Our implementation is very simple, but you will see how we can extend it beyond this simple premise later in the chapter.

The Jekyll base image

Let's start creating a new Dockerfile for our first image: the Jekyll base image. Let's create a new directory first and an empty Dockerfile.

Listing 6.1: Creating our Jekyll Dockerfile

```
$ mkdir jekyll
$ cd jekyll
$ vi Dockerfile
```

Now let's populate our Dockerfile.

Listing 6.2: Jekyll Dockerfile

```
FROM ubuntu:14.04
MAINTAINER James Turnbull <james@example.com>
ENV REFRESHED_AT 2014-06-01

RUN apt-get -yqq update
RUN apt-get -yqq install ruby ruby-dev make nodejs
RUN gem install --no-rdoc --no-ri jekyll

VOLUME [ "/data/", "/var/www/html" ]
WORKDIR /data

ENTRYPOINT [ "jekyll", "build", "--destination=/var/www/html" ]
```

Our Dockerfile uses the template we saw in Chapter 3 as its basis. Our image is based on Ubuntu 14.04 and installs Ruby and the prerequisites necessary to support Jekyll. It creates two volumes using the VOLUME instruction:

- /data/, which is going to hold our new website source code.
- /var/www/html/, which is going to hold our compiled Jekyll site.

We also need to set the working directory to /data/ and specify an ENTRYPOINT instruction that will automatically build any Jekyll site it finds in the /data/ working directory into the /var/www/html/ directory.

Building the Jekyll base image

With this Dockerfile, we can now build an image from which we can launch containers. We'll do this using the docker build command.

Listing 6.3: Building our Jekyll image

```
$ sudo docker build -t jamtur01/jekyll .
Sending build context to Docker daemon 2.56 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu:14.04
--> 99ec81b80c55
Step 1 : MAINTAINER James Turnbull <james@example.com>
...
Step 7 : ENTRYPOINT [ "jekyll", "build" "--destination=/var/www/←
  html" ]
--> Running in 542e2de2029d
--> 79009691f408
Removing intermediate container 542e2de2029d
Successfully built 79009691f408
```

We can see that we've built a new image with an ID of 79009691f408 named `jamtur01/jekyll` that is our new Jekyll image. We can view our new image using the `docker images` command.

Listing 6.4: Viewing our new Jekyll Base image

```
$ sudo docker images
REPOSITORY      TAG      ID          CREATED      SIZE
jamtur01/jekyll latest  79009691f408  6 seconds ago  12.29 kB (←
  virtual 671 MB)
...
```

The Apache image

Finally, let's build our second image, an Apache server to serve out our new site. Let's create a new directory first and an empty Dockerfile.

Listing 6.5: Creating our Apache Dockerfile

```
$ mkdir apache
$ cd apache
$ vi Dockerfile
```

Now let's populate our Dockerfile.

Listing 6.6: Jekyll Apache Dockerfile

```
FROM ubuntu:14.04
MAINTAINER James Turnbull <james@example.com>
ENV REFRESHED_AT 2014-06-01

RUN apt-get -yqq update
RUN apt-get -yqq install apache2

VOLUME [ "/var/www/html" ]
WORKDIR /var/www/html

ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2
ENV APACHE_PID_FILE /var/run/apache2.pid
ENV APACHE_RUN_DIR /var/run/apache2
ENV APACHE_LOCK_DIR /var/lock/apache2

RUN mkdir -p $APACHE_RUN_DIR $APACHE_LOCK_DIR $APACHE_LOG_DIR

EXPOSE 80

ENTRYPOINT [ "/usr/sbin/apache2" ]
CMD [ "-D", "FOREGROUND" ]
```

This final image is again based on Ubuntu 14.04 and installs Apache. It creates a volume using the `VOLUME` instruction, `/var/www/html/`, which is going to hold our

compiled Jekyll website. We also set `/var/www/html` to be our working directory. We'll then use some `ENV` instructions to set some required environment variables, create some required directories, and `EXPOSE` port 80. We've also specified an `ENTRYPOINT` and `CMD` combination to run Apache by default when the container starts.

Building the Jekyll Apache image

With this Dockerfile, we can now build an image from which we can launch containers. We do this using the `docker build` command.

Listing 6.7: Building our Jekyll Apache image

```
$ sudo docker build -t jamtur01/apache .
Sending build context to Docker daemon 2.56 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu:14.04
--> 99ec81b80c55
Step 1 : MAINTAINER James Turnbull <james@example.com>
--> Using cache
--> c444e8ee0058
...
Step 11 : CMD ["-D", "FOREGROUND"]
--> Running in 7aa5c127b41e
--> fc8e9135212d
Removing intermediate container 7aa5c127b41e
Successfully built fc8e9135212d
```

We can see that we've built a new image with an ID of `fc8e9135212d` named `jamtur01/apache` that is our new Apache image. We can view our new image using the `docker images` command.

Listing 6.8: Viewing our new Jekyll Apache image

```
$ sudo docker images
REPOSITORY      TAG      ID          CREATED      SIZE
jamtur01/apache latest  fc8e9135212d  6 seconds ago  12.29 kB (←
  virtual 671 MB)
...
```

Launching our Jekyll site

Now we've got two images:

- Jekyll - Our Jekyll image with Ruby and the prerequisites installed.
- Apache - The image that will serve our compiled website via the Apache web server.

Let's get started on our new site by creating a new Jekyll container using the `docker run` command. We're going to launch a container and build our site.

We're going to need some source code for our blog. Let's clone a sample Jekyll blog into our `$HOME` directory (in my case `/home/james`).

Listing 6.9: Getting a sample Jekyll blog

```
$ cd $HOME
$ git clone https://github.com/jamtur01/james_blog.git
```

You can see a basic [Twitter Bootstrap](#)-enabled Jekyll blog inside this directory. If you want to use it, you can easily update the `_config.yml` file and the theme to suit your purposes.

Now let's use this sample data inside our Jekyll container.

Listing 6.10: Creating a Jekyll container

```
$ sudo docker run -v /home/james/james_blog:/data/ \
--name james_blog \jamtur01/jekyll
Configuration file: /data/_config.yml
  Source: /data
  Destination: /var/www/html
  Generating...
  done.
```

We've started a new container called `james_blog` and mounted our `james_blog` directory inside the container as the `/data/` volume. The container has taken this source code and built it into a compiled site stored in the `/var/www/html/` directory.

So we've got a completed site, now how do we use it? This is where volumes become a lot more interesting. When we briefly introduced volumes in Chapter 4, we discovered a bit about them. Let's revisit that.

A volume is a specially designated directory within one or more containers that bypasses the Union File System to provide several useful features for persistent or shared data:

- Volumes can be shared and reused between containers.
- A container doesn't have to be running to share its volumes.
- Changes to a volume are made directly.
- Changes to a volume will not be included when you update an image.
- Volumes persist until no containers use them.

This allows you to add data (e.g., source code, a database, or other content) into an image without committing it to the image and allows you to share that data between containers.

Volumes live on your Docker host, in the `/var/lib/docker/volumes` directory. You can identify the location of specific volumes using the `docker inspect` command; for example, `docker inspect -f "{{ .Volumes }}"`.

So if we want to use our compiled site in the `/var/www/html/` volume from another container, we can do so. To do this, we'll create a new container that links to this volume.

Listing 6.11: Creating an Apache container

```
$ sudo docker run -d -P --volumes-from james_blog jamtur01/apache  
09a570cc2267019352525079fbba9927806f782acb88213bd38dde7e2795407d
```

This looks like a typical `docker run`, except that we've used a new flag: `--volumes-from`. The `--volumes-from` flag adds any volumes in the named container to the newly created container. This means our Apache container has access to the compiled Jekyll site in the `/var/www/html` volume within the `james_blog` container we created earlier. It has that access even though the `james_blog` container is not running. As you'll recall, that is one of the special properties of volumes. The container does have to exist, though. If you had deleted the `james_blog` container using the `docker rm` command, then the volume would be gone.

WARNING If you delete the last container that uses a volume, the volume will disappear. Be careful about deleting containers that might hold volumes with data you need. We'll see how to backup a volume shortly to help you avoid this issue.

What is the end result of building our Jekyll website? Let's see onto what port our container has mapped our exposed port 80:

Listing 6.12: Resolving the Apache container's port

```
$ sudo docker port 09a570cc2267 80  
0.0.0.0:49160
```

Now let's browse to that site on our Docker host.

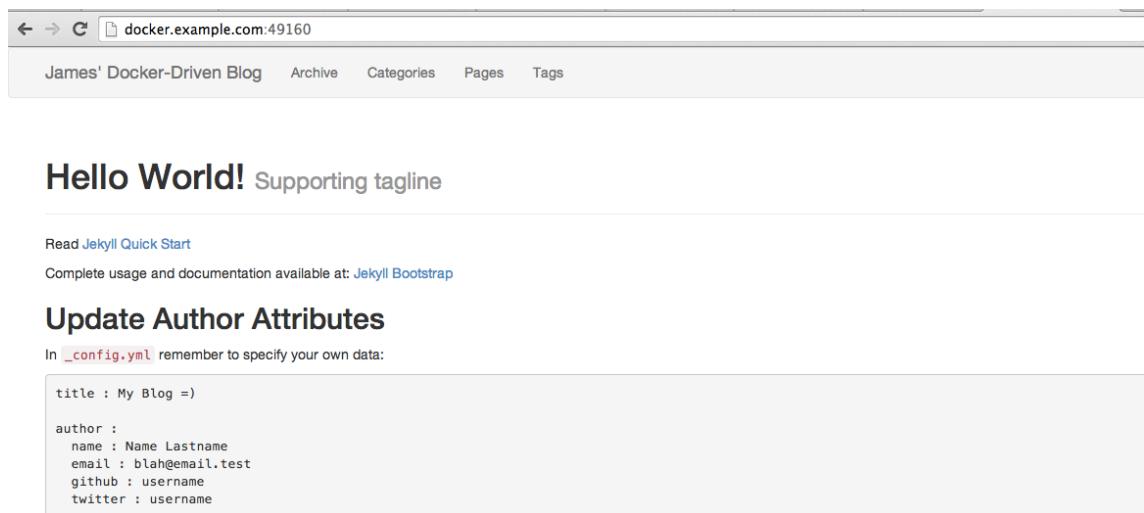


Figure 6.1: Our Jekyll website.

We have a running Jekyll website!

Updating our Jekyll site

Things get even more interesting when we want to update our site. Let's say we'd like to make some changes to our Jekyll website. We're going to rename our blog by editing the `james_blog/_config.yml` file.

Listing 6.13: Editing our Jekyll blog

```
$ vi james_blog/_config.yml
```

And update the `title` field to `James' Dynamic Docker-driven Blog`.

So how do we update our blog? All we need to do is start our Docker container again with the `docker start` command..

Listing 6.14: Restarting our `james_blog` container

```
$ sudo docker start james_blog
james_blog
```

It looks like nothing happened. Let's check the container's logs.

Listing 6.15: Checking the `james_blog` container logs

```
$ sudo docker logs james_blog
Configuration file: /data/_config.yml
  Source: /data
  Destination: /var/www/html
  Generating...
    done.
Configuration file: /data/_config.yml
  Source: /data
  Destination: /var/www/html
  Generating...
    done.
```

We can see that the Jekyll build process has been run a second time and our site has been updated. The update has been written to our volume. Now if we browse to the Jekyll website, we should see our update.

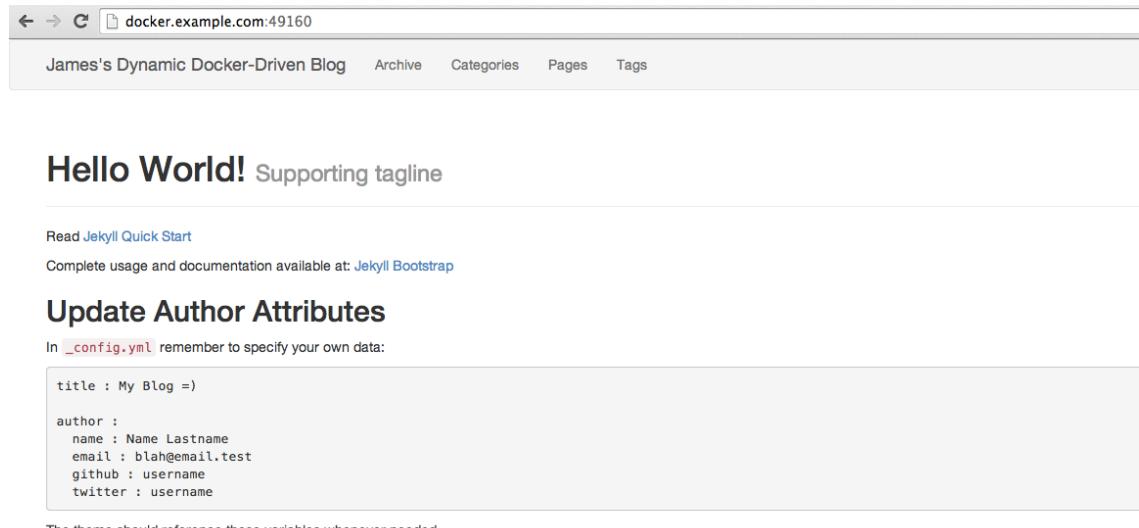


Figure 6.2: Our updated Jekyll website.

This all happened without having to update or restart our Apache container, be-

cause the volume it was sharing was updated automatically. You can see how easy this workflow is and how you could expand it for more complicated deployments.

Backing up our Jekyll volume

You're probably a little worried about accidentally deleting your volume (although we can pretty easily rebuild our site using the existing process). One of the advantages of volumes is that because they can be mounted into any container, we can very easily create backups of them. Let's create a new container now that backs up the `/var/www/html` volume.

Listing 6.16: Backing up the `/var/www/html` volume

```
$ sudo docker run --rm --volumes-from james_blog \
-v $(pwd):/backup ubuntu \
tar cvf /backup/james_blog_backup.tar /var/www/html
tar: Removing leading '//' from member names
/var/www/html/
/var/www/html/assets/
/var/www/html/assets/themes/
...
$ ls james_blog_backup.tar
james_blog_backup.tar
```

Here we've run a stock Ubuntu container and mounted the volume from `james_blog` into that container. That will create the directory `/var/www/html` inside the container. We've then used the `-v` flag to mount our current directory, using the `$(pwd)` command, inside the container at `/backup`. Our container then runs the command.

TIP We've also specified the `--rm` flag, which is useful for single-use or throw-away containers. It automatically deletes the container after the process running in it is ended. This is a neat way of tidying up after ourselves for containers we only need once.

Listing 6.17: Backup command

```
tar cvf /backup/james_blog_backup.tar /var/www/html
```

This will create a tarfile called `james_blog_backup.tar` containing the contents of the `/var/www/html` directory and then exit. This process creates a backup of our volume.

This is obviously an incredibly simple example of a backup process. You could easily extend this to back up to storage locally or in the cloud (e.g., to [Amazon S3](#) or to more traditional backup software like [Amanda](#)).

TIP This example could also work for a database stored in a volume or similar data. Simply mount the volume in a fresh container, perform your backup, and discard the container you created for the backup.

Extending our Jekyll website example

Here are some ways we could expand on our simple Jekyll website service:

- Run multiple Apache containers, all which use the same volume from the `james_blog` container. Put a load balancer in front of it, and we have a web cluster.
- Build a further image that cloned or copied a user-provided source (e.g., a `git clone`) into a volume. Mount this volume into a container created from our `jamtur01/jekyll` image. This would make the solution portable and generic and would not require any local source on a host.
- With the previous expansion, you could easily build a web front end for our service that built and deployed sites automatically from a specified source. Then you would have your very own GitHub Pages.

Building a Java application server with Docker

Now let's take a slightly different tack and think about Docker as an application server and build pipeline. This time we're serving a more "enterprisey" and traditional workload: fetching and running a Java application from a WAR file in a Tomcat server. To do this, we're going to build a two-stage Docker pipeline:

- An image that pulls down specified WAR files from a URL and stores them in a volume.
- An image with a Tomcat server installed that runs those downloaded WAR files.

A WAR file fetcher

Let's start by building an image to download a WAR file for us and mount it in a volume.

Listing 6.18: Creating our fetcher Dockerfile

```
$ mkdir fetcher
$ cd fetcher
$ touch Dockerfile
```

Now let's populate our Dockerfile.

Listing 6.19: Our war file fetcher

```
FROM ubuntu:14.04
MAINTAINER James Turnbull <james@example.com>
ENV REFRESHED_AT 2014-06-01

RUN apt-get -yqq update
RUN apt-get -yqq install wget

VOLUME [ "/var/lib/tomcat7/webapps/" ]
WORKDIR /var/lib/tomcat7/webapps/

ENTRYPOINT [ "wget" ]
CMD [ "-?" ]
```

This incredibly simple image does one thing: it `wget`s whatever file from a URL that is specified when a container is run from it and stores the file in the `/var/lib/tomcat7/webapps/` directory. This directory is also a volume and the working directory for any containers. We're going to share this volume with our Tomcat server and run its contents.

Finally, the `ENTRYPOINT` and `CMD` instructions allow our container to run when no URL is specified; they do so by returning the `wget` help output when the container is run without a URL.

Let's build this image now.

Listing 6.20: Building our fetcher image

```
$ sudo docker build -t jamtur01/fetcher .
```

Fetching a WAR file

Let's fetch an example file as a way to get started with our new image. We're going to download the sample Apache Tomcat application from [here](#).

Listing 6.21: Fetching a war file

```
$ sudo docker run -t -i --name sample jamtur01/fetcher \
https://tomcat.apache.org/tomcat-7.0-doc/appdev/sample/sample.war
--2014-06-21 06:05:19-- https://tomcat.apache.org/tomcat-7.0-doc<-
/appdev/sample/sample.war
Resolving tomcat.apache.org (tomcat.apache.org)... <-
140.211.11.131, 192.87.106.229, 2001:610:1:80bc:192:87:106:229
Connecting to tomcat.apache.org (tomcat.apache.org)<-
|140.211.11.131|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 4606 (4.5K)
Saving to: 'sample.war'

100%[=====] 4,606          --.-K/s   in<-
0s

2014-06-21 06:05:19 (14.4 MB/s) - 'sample.war' saved [4606/4606]
```

We can see that our container has taken the provided URL and downloaded the `sample.war` file. We can't see it here, but because we set the working directory in the container, that `sample.war` file will have ended up in our `/var/lib/tomcat7/webapps/` directory.

We can find our WAR file in the `/var/lib/docker` directory. Let's first establish where the volume is located using the `docker inspect` command.

Listing 6.22: Inspecting our Sample volume

```
$ sudo docker inspect -f "{{ .Volumes }}" sample
map[/var/lib/tomcat7/webapps:/var/lib/docker/vfs/dir/<-
e59cd92502663adf6e76ae57a49c0d858950cd01f32415ab6a09b44eaf8727e<-
]
```

We can then list this directory.

Listing 6.23: Listing the volume directory

```
$ ls -l /var/lib/docker/vfs/dir/←  
e59cd92502663adf6e76ae57a49c0d858950cd01f32415ab6a09b44eaf8727e  
total 8  
-rw-r--r-- 1 root root 4606 Mar 31 2012 sample.war
```

Our Tomcat 7 application server

We have an image that will get us WAR files, and we have a sample WAR file downloaded into a container. Let's build an image that will be the Tomcat application server that will run our WAR file.

Listing 6.24: Creating our Tomcat 7 Dockerfile

```
$ mkdir tomcat7  
$ cd tomcat7  
$ touch Dockerfile
```

Now let's populate our Dockerfile.

Listing 6.25: Our Tomcat 7 Application server

```
FROM ubuntu:14.04
MAINTAINER James Turnbull <james@example.com>
ENV REFRESHED_AT 2014-06-01

RUN apt-get -yqq update
RUN apt-get -yqq install tomcat7 default-jdk

ENV CATALINA_HOME /usr/share/tomcat7
ENV CATALINA_BASE /var/lib/tomcat7
ENV CATALINA_PID /var/run/tomcat7.pid
ENV CATALINA_SH /usr/share/tomcat7/bin/catalina.sh
ENV CATALINA_TMPDIR /tmp/tomcat7-tomcat7-tmp

RUN mkdir -p $CATALINA_TMPDIR

VOLUME [ "/var/lib/tomcat7/webapps/" ]

EXPOSE 8080

ENTRYPOINT [ "/usr/share/tomcat7/bin/catalina.sh", "run" ]
```

Our image is pretty simple. We need to install a Java JDK and the Tomcat server. We'll specify some environment variables Tomcat needs in order to get started, then create a temporary directory. We'll also create a volume called `/var/lib/tomcat7/webapps/`, expose port 8080 (the Tomcat default), and finally use an `ENTRYPOINT` instruction to launch Tomcat.

Now let's build our Tomcat 7 image.

Listing 6.26: Building our Tomcat 7 image

```
$ sudo docker build -t jamtur01/tomcat7 .
```

Running our WAR file

Now let's see our Tomcat server in action by creating a new Tomcat instance running our sample application.

Listing 6.27: Creating our first Tomcat instance

```
$ sudo docker run --name sample_app --volumes-from sample \
-d -P jamtur01/tomcat7
```

This will create a new container named `sample_app` that reuses the volumes from the `sample` container. This means our WAR file, stored in the `/var/lib/tomcat7/webapps/` volume, will be mounted from the `sample` container into the `sample_app` container and then loaded by Tomcat and executed.

Let's look at our sample application in the web browser. First, we must identify the port being exposed using the `docker port` command.

Listing 6.28: Identifying the Tomcat application port

```
sudo docker port sample_app 8080
0.0.0.0:49154
```

Now let's browse to our application (using the URL and port and adding the `/sample` suffix) and see what's there.

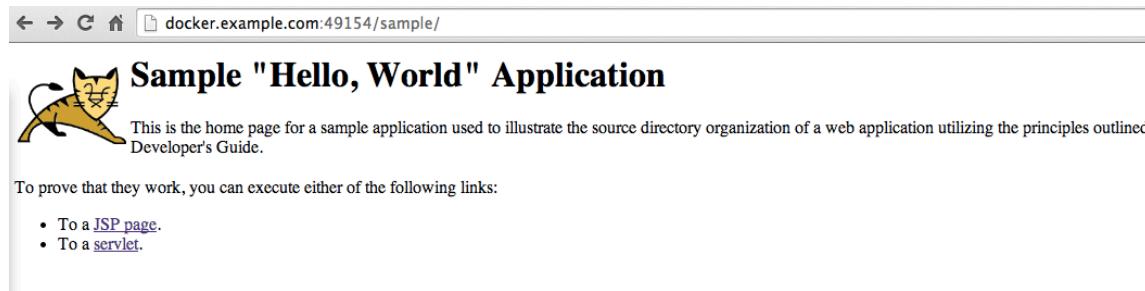


Figure 6.3: Our Tomcat sample application.

We should see our running Tomcat application.

Building on top of our Tomcat application server

Now we have the building blocks of a simple on-demand web service. Let's look at how we might expand on this. To do so, we've built a simple Sinatra-based web application to automatically provision Tomcat applications via a web page. We've called this application TProv. You can see its source code [here](#) or on [GitHub](#).

Let's install it as a demo of how you might extend this or similar examples. First, we'll need to ensure Ruby is installed. We're going to install our TProv application on our Docker host because our application is going to be directly interacting with our Docker daemon, so that's where we need to install Ruby.

NOTE We could also install the TProv application inside a Docker container using the Docker-in-Docker trick we learned in Chapter 5.

Listing 6.29: Installing Ruby

```
$ sudo apt-get -qqy install ruby make ruby-dev
```

We can then install our application from a Ruby gem.

Listing 6.30: Installing the TProv application

```
$ sudo gem install --no-rdoc --no-ri tprov
...
Successfully installed tprov-0.0.4
```

This will install the TProv application and some supporting gems.

We can then launch the application using the `tprov` binary.

Listing 6.31: Launching the TProv application

```
$ sudo tprov
[2014-06-21 16:17:24] INFO  WEBrick 1.3.1
[2014-06-21 16:17:24] INFO  ruby 1.8.7 (2011-06-30) [x86_64-linux]
[2014-06-21 16:17:24] == Sinatra/1.4.5 has taken the stage on 4567 for development with<-
[2014-06-21 16:17:24] == backup from WEBrick
[2014-06-21 16:17:24] INFO  WEBrick::HTTPServer#start: pid=14209 <-
[2014-06-21 16:17:24] == port=4567
```

This command has launched our application; now we can browse to the TProv website on port 4567 of the Docker host.

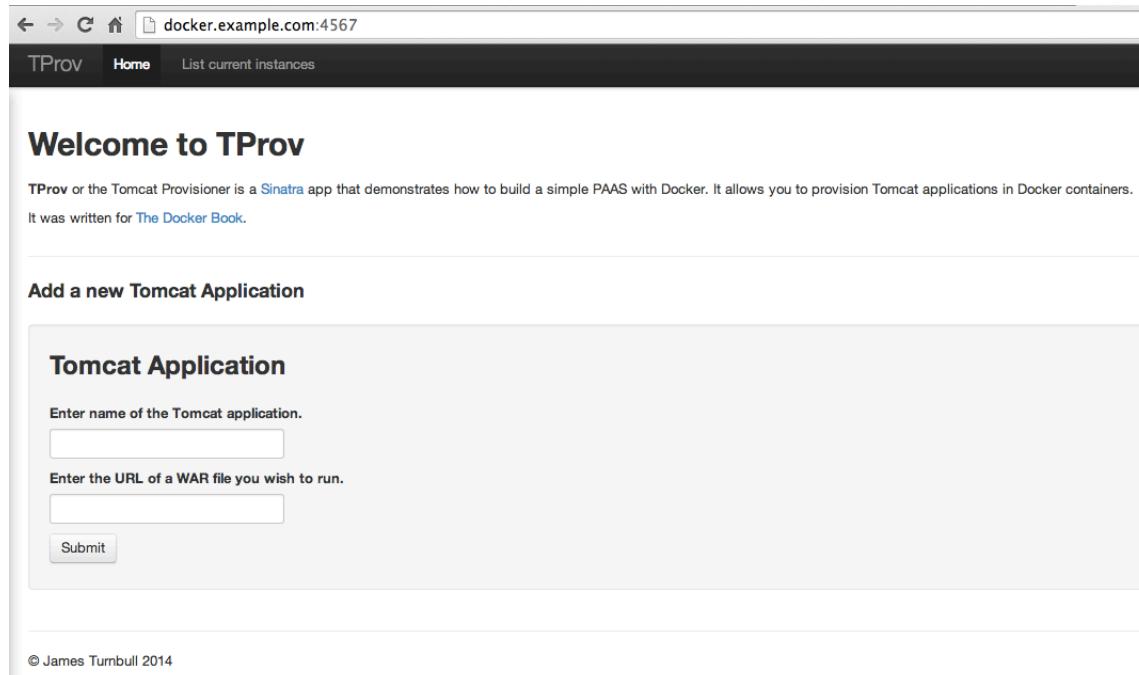
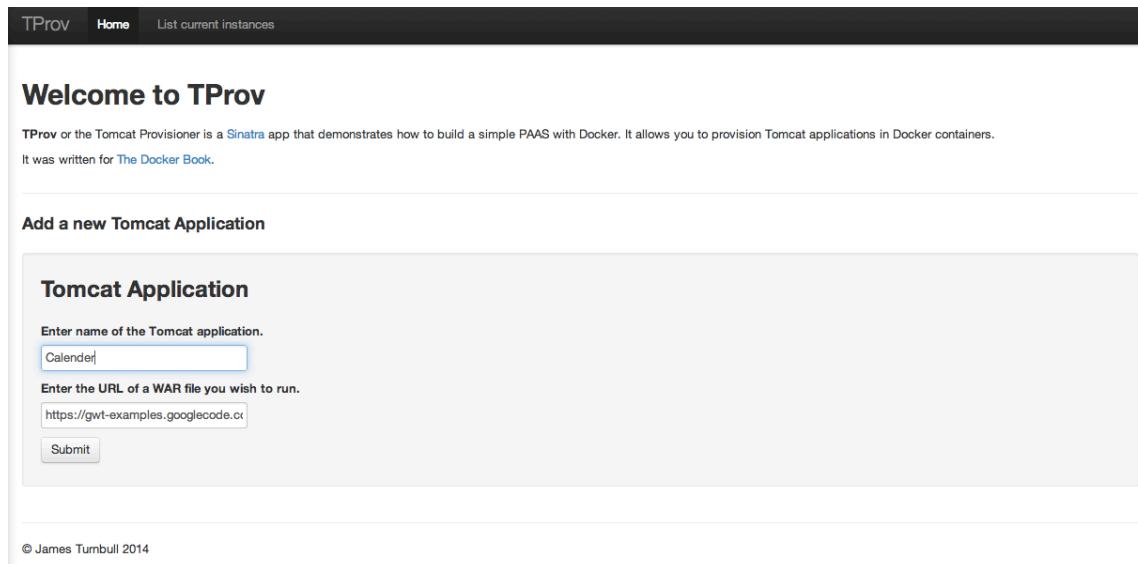


Figure 6.4: Our TProv web application.

As we can see, we can specify a Tomcat application name and the URL to a Tomcat WAR file. Let's download a sample calendar application from [here](#) and call it Calendar.



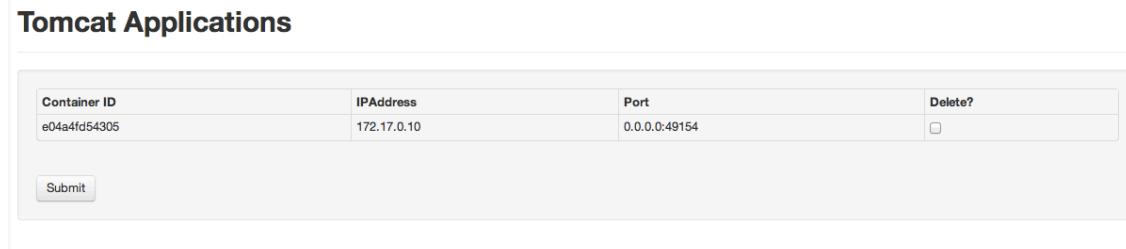
The screenshot shows the TProv application interface. At the top, there is a navigation bar with 'TProv' and 'Home' (which is the active tab) and a link to 'List current instances'. Below the navigation bar, the main content area has a title 'Welcome to TProv'. A sub-section titled 'Tomcat Application' contains fields for 'Enter name of the Tomcat application' (with 'Calender' typed in) and 'Enter the URL of a WAR file you wish to run' (with 'https://gwt-examples.googlecode.com' typed in). A 'Submit' button is located below these fields. At the bottom of the page, there is a copyright notice: '© James Turnbull 2014'.

Figure 6.5: Downloading a sample application.

We click Submit to download the WAR file, place it into a volume, run a Tomcat server, and serve the WAR file in that volume. We can see our instance by clicking on the List instances link.

This shows us:

- The container ID.
- The container's internal IP address.
- The interface and port it is mapped to.



The screenshot shows the 'Tomcat Applications' list. The table has four columns: 'Container ID', 'IPAddress', 'Port', and 'Delete?'. There is one entry: 'e04a4fd54305' in the Container ID column, '172.17.0.10' in the IPAddress column, '0.0.0.0:49154' in the Port column, and an empty checkbox in the Delete? column. A 'Submit' button is located at the bottom of the table.

Container ID	IPAddress	Port	Delete?
e04a4fd54305	172.17.0.10	0.0.0.0:49154	<input type="checkbox"/>

Figure 6.6: Listing the Tomcat instances.

Using this information, we can check the status of our application by browsing to the mapped port. We can also use the Delete? checkbox to remove an instance.

You can see how we achieved this by looking at [the TProv application code](#). It's a pretty simple application that shells out to the docker binary and captures output to run and remove containers.

You're welcome to use the TProv code or adapt or write your own ¹, but its primary purpose is to show you how easy it is to extend a simple application deployment pipeline built with Docker.

WARNING The TProv application is pretty simple and lacks some error handling and tests. It is also built on a very simple code: it was built in an hour to demonstrate how powerful Docker can be as a tool for building applications and services. If you find a bug with the application (or want to make it better), please let me know [with an issue or PR here](#).

A multi-container application stack

In our last service example, we're going full hipster by Dockerizing a Node.js application that makes use of the Express framework with a Redis back end. We're going to demonstrate a combination of all the Docker features we've learned over the last two chapters, including links and volumes.

In our sample application, we're going to build a series of images that will allow us to deploy a multi-container application:

- A Node container to serve our Node application, linked to:
- A Redis primary container to hold and cluster our state, linked to:
- Two Redis replica containers to cluster our state.
- A logging container to capture our application logs.

¹Really write your own - no one but me loves my code.

We're then going to run our Node application in a container with Redis in primary-replica configuration in multiple containers behind it.

The Node.js image

Let's start with an image that installs Node.js, our Express application, and the associated prerequisites.

Listing 6.32: Creating our Node.js Dockerfile

```
$ mkdir nodejs
$ cd nodejs
$ mkdir -p nodeapp
$ cd nodeapp
$ wget https://raw.githubusercontent.com/jamtur01/dockerbook-code<-
  /master/code/6/node/nodejs/nodeapp/package.json
$ wget https://raw.githubusercontent.com/jamtur01/dockerbook-code<-
  /master/code/6/node/nodejs/nodeapp/server.js
$ cd ..
$ vi Dockerfile
```

We've created a new directory called `nodejs` and then a sub-directory, `nodeapp`, to hold our application code. We've then changed into this directory and downloaded the source code for our Node.JS application.

NOTE You can get our Node application's source code [here](#) or on [GitHub](#).

Finally we've changed back to the `nodejs` directory and now we can populate our Dockerfile.

Listing 6.33: Our Node.js image

```
FROM ubuntu:14.04
MAINTAINER James Turnbull <james@example.com>
ENV REFRESHED_AT 2014-06-01

RUN apt-get -yqq update
RUN apt-get -yqq install nodejs npm
RUN ln -s /usr/bin/nodejs /usr/bin/node
RUN mkdir -p /var/log/nodeapp

ADD nodeapp /opt/nodeapp/

WORKDIR /opt/nodeapp
RUN npm install

VOLUME [ "/var/log/nodeapp" ]

EXPOSE 3000

ENTRYPOINT [ "nodejs", "server.js" ]
```

Our Node.js image installs Node and makes a simple workaround of linking the binary `nodejs` to `node` to address some backwards compatibility issues on Ubuntu.

We then add our `nodeapp` code into the `/opt/nodeapp` directory using an `ADD` instruction. Our Node.js application is a very simple Express server and contains both a `package.json` file holding the application's dependency information and the `server.js` file that contains our actual application. Let's look at a subset of that application.

Listing 6.34: Our Node.js server.js application

```
    . . .

var logFile = fs.createWriteStream('/var/log/nodeapp/nodeapp.log',
  {flags: 'a'});

app.configure(function() {

    . . .

    app.use(express.session({
        store: new RedisStore({
            host: process.env.REDIS_HOST || 'redis_primary',
            port: process.env.REDIS_PORT || 6379,
            db: process.env.REDIS_DB || 0
        }),
        cookie: {

            . . .

        }
    }));

    app.get('/', function(req, res) {
        res.json({
            status: "ok"
        });
    });
}

. . .

var port = process.env.HTTP_PORT || 3000;
server.listen(port);
console.log('Listening on port ' + port);
```

The `server.js` file pulls in all the dependencies and starts an Express application. The Express app is configured to store its session information in Redis and exposes

a single endpoint that returns a status message as JSON. We've configured its connection to Redis to use a host called `redis_primary` with an option to override this with an environment variable if needed.

The application will also log to the `/var/log/nodeapp/nodeapp.log` file and will listen on port 3000.

NOTE You can get our Node application's source code [here](#) or on [GitHub](#).

We've then set the working directory to `/opt/nodeapp` and installed the prerequisites for our Node application. We've also created a volume that will hold our Node application's logs, `/var/log/nodeapp`.

We expose port 3000 and finally specify an `ENTRYPOINT` of `nodejs server.js` that will run our Node application.

Let's build our image now.

Listing 6.35: Building our Node.js image

```
$ sudo docker build -t jamtur01/nodejs .
```

The Redis base image

Let's continue with our first Redis image: a base image that will install Redis. It is on top of this base image that we'll build our Redis primary and replica images.

Listing 6.36: Creating our Redis base Dockerfile

```
$ mkdir redis_base
$ cd redis_base
$ vi Dockerfile
```

Now let's populate our Dockerfile.

Listing 6.37: Our Redis base image

```
FROM ubuntu:14.04
MAINTAINER James Turnbull <james@example.com>
ENV REFRESHED_AT 2014-06-01

RUN apt-get -yqq update
RUN apt-get install -yqq software-properties-common python-←
    software-properties
RUN add-apt-repository ppa:chris-lea/redis-server
RUN apt-get -yqq update
RUN apt-get -yqq install redis-server redis-tools

VOLUME [ "/var/lib/redis", "/var/log/redis/" ]

EXPOSE 6379
CMD []
```

Our Redis base image installs the latest version of Redis (from a PPA rather than using the older packages shipped with Ubuntu), specifies two VOLUMEs (/var/lib/redis and /var/log/redis), and exposes the Redis default port 6379. It doesn't have an ENTRYPOINT or CMD because we're not actually going to run this image. We're just going to build on top of it.

Let's build our Redis primary image now.

Listing 6.38: Building our Redis base image

```
$ sudo docker build -t jamtur01/redis .
```

The Redis primary image

Let's continue with our first Redis image: a Redis primary server.

Listing 6.39: Creating our Redis primary Dockerfile

```
$ mkdir redis_primary
$ cd redis_primary
$ vi Dockerfile
```

Now let's populate our Dockerfile.

Listing 6.40: Our Redis primary image

```
FROM jamtur01/redis
MAINTAINER James Turnbull <james@example.com>
ENV REFRESHED_AT 2014-06-01

ENTRYPOINT [ "redis-server", "--logfile /var/log/redis/redis-←
server.log" ]
```

Our Redis primary image is based on our jamtur01/redis image and has an ENTRYPOINT that runs the default Redis server with logging directed to /var/←log/redis/redis-server.log.

Let's build our Redis primary image now.

Listing 6.41: Building our Redis primary image

```
$ sudo docker build -t jamtur01/redis_primary .
```

The Redis replica image

As a complement to our Redis primary image, we're going to create an image that runs a Redis replica to allow us to provide some redundancy to our Node.js application.

Listing 6.42: Creating our Redis replica Dockerfile

```
$ mkdir redis_replica
$ cd redis_replica
$ touch Dockerfile
```

Now let's populate our Dockerfile.

Listing 6.43: Our Redis replica image

```
FROM jamtur01/redis
MAINTAINER James Turnbull <james@example.com>
ENV REFRESHED_AT 2014-06-01

ENTRYPOINT [ "redis-server", "--logfile /var/log/redis/redis-←
replica.log", "--slaveof redis_primary 6379" ]
```

Again, we base our image on `jamtur01/redis` and specify an `ENTRYPOINT` that runs the default Redis server with our logfile and the `slaveof` option. This configures our primary-replica relationship and tells any containers built from this image that they are a replica of the `redis_primary` host and should attempt replication on port 6379.

Let's build our Redis replica image now.

Listing 6.44: Building our Redis replica image

```
$ sudo docker build -t jamtur01/redis_replica .
```

Creating our Redis back-end cluster

Now that we have both a Redis primary and replica image, we can build our own Redis replication environment. Let's start by building the Redis primary container.

Listing 6.45: Running the Redis primary container

```
$ sudo docker run -d -h redis_primary \
--name redis_primary jamtur01/redis_primary
d21659697baf56346cc5bbe8d4631f670364ffddf4863ec32ab0576e85a73d27
```

Here we've created a container with the `docker run` command from the `jamtur01/redis_primary` image. We've used a new flag that we've not seen before, `-h`, which sets the hostname of the container. This overrides the default behavior (setting the hostname of the container to the short container ID) and allows us to specify our own hostname. We'll use this to ensure that our container is given a hostname of `redis_primary` and will thus be resolved that way with local DNS. We've also specified the `--name` flag to ensure that our container's name is `redis_primary`, too. We're going to use this for our container linking, as we'll see shortly.

Let's see what the `docker logs` command can tell us about our Redis primary container.

Listing 6.46: Our Redis primary logs

```
$ sudo docker logs redis_primary
```

Nothing? Why is that? Our Redis server is logging to a file rather than to standard out, so we see nothing in the Docker logs. So how can we tell what's happening to our Redis server? To do that, we can use the `/var/log/redis` volume we created earlier. Let's use this volume and read some log files now.

Listing 6.47: Reading our Redis primary logs

```
$ sudo docker run -ti --rm --volumes-from redis_primary \
ubuntu cat /var/log/redis/redis-server.log
...
[1] 25 Jun 21:45:03.074 # Server started, Redis version 2.8.12
[1] 25 Jun 21:45:03.074 # WARNING overcommit_memory is set to 0! ←
Background save may fail under low memory condition. To fix this←
issue add 'vm.overcommit_memory = 1' to /etc/sysctl.conf and ←
then reboot or run the command 'sysctl vm.overcommit_memory=1' ←
for this to take effect.
[1] 25 Jun 21:45:03.074 * The server is now ready to accept ←
connections on port 6379
```

Here we've run another container interactively. We've specified the `--rm` flag, which automatically deletes a container after the process it runs stops. We've also specified the `--volumes-from` flag and told it to mount the volumes from our `redis_primary` container. Then we've specified a base `ubuntu` image and told it to cat the `/var/log/redis/redis-server.log` log file. This takes advantage of volumes to allow us to mount the `/var/log/redis` directory from the `redis_primary` container and read the log file inside it. We're going to see more about how we can use this shortly.

Looking at our Redis logs, we see some general warnings, but everything is looking pretty good. Our Redis server is ready to receive data on port 6379.

So next, let's create our first Redis replica.

Listing 6.48: Running our first Redis replica container

```
$ sudo docker run -d -h redis_replica1 \
--name redis_replica1 \
--link redis_primary:redis_primary \
jamtur01/redis_replica
0ae440b5c56f48f3190332b4151c40f775615016bf781fc817f631db5af34ef8
```

We've run another container: this one from the `jamtur01/redis_replica` image. We've again specified a hostname (with the `-h` flag) and a name (with `--name`←

) of `redis_replica1`. We've also used the `--link` flag to link our Redis replica container with the `redis_primary` container with the alias `redis_primary`.

Let's check this new container's logs.

Listing 6.49: Reading our Redis replica logs

```
$ sudo docker run -ti --rm --volumes-from redis_replica1 \
ubuntu cat /var/log/redis/redis-replica.log
...
[1] 25 Jun 22:10:04.240 # Server started, Redis version 2.8.12
[1] 25 Jun 22:10:04.240 # WARNING overcommit_memory is set to 0! ←
  Background save may fail under low memory condition. To fix this←
  issue add 'vm.overcommit_memory = 1' to /etc/sysctl.conf and ←
  then reboot or run the command 'sysctl vm.overcommit_memory=1' ←
  for this to take effect.
[1] 25 Jun 22:10:04.240 * The server is now ready to accept ←
  connections on port 6379
[1] 25 Jun 22:10:04.242 * Connecting to MASTER redis_primary:6379
[1] 25 Jun 22:10:04.244 * MASTER <-> SLAVE sync started
[1] 25 Jun 22:10:04.244 * Non blocking connect for SYNC fired the←
  event.
[1] 25 Jun 22:10:04.244 * Master replied to PING, replication can←
  continue...
[1] 25 Jun 22:10:04.245 * Partial resynchronization not possible ←
  (no cached master)
[1] 25 Jun 22:10:04.246 * Full resync from master: 24←
  a790df6bf4786a0e886be4b34868743f6145cc:1485
[1] 25 Jun 22:10:04.274 * MASTER <-> SLAVE sync: receiving 18 ←
  bytes from master
[1] 25 Jun 22:10:04.274 * MASTER <-> SLAVE sync: Flushing old ←
  data
[1] 25 Jun 22:10:04.274 * MASTER <-> SLAVE sync: Loading DB in ←
  memory
[1] 25 Jun 22:10:04.275 * MASTER <-> SLAVE sync: Finished with ←
  success
```

We've run another container to query our logs interactively. We've again specified the `--rm` flag, which automatically deletes a container after the process it runs stops. We've specified the `--volumes-from` flag and told it to mount the volumes from our `redis_replica1` container this time. Then we've specified a base `ubuntu` image and told it to cat the `cat /var/log/redis/redis-replica.log` log file.

Woot! We're off and replicating between our `redis_primary` container and our `redis_replica1` container.

Let's add another replica, `redis_replica2`, just to be sure.

Listing 6.50: Running our second Redis replica container

```
$ sudo docker run -d -h redis_replica2 \
--name redis_replica2 \
--link redis_primary:redis_primary \
jamtur01/redis_replica
72267cd74c412c7b168d87bba70f3aaa3b96d17d6e9682663095a492bc260357
```

Let's see a sampling of the logs from our new container.

Listing 6.51: Our Redis replica2 logs

```
$ sudo docker run -ti --rm --volumes-from redis_replica2 ubuntu \
cat /var/log/redis/redis-replica.log
.
.
.
[1] 25 Jun 22:11:39.417 # Server started, Redis version 2.8.12
[1] 25 Jun 22:11:39.417 # WARNING overcommit_memory is set to 0! ←
Background save may fail under low memory condition. To fix this←
issue add 'vm.overcommit_memory = 1' to /etc/sysctl.conf and ←
then reboot or run the command 'sysctl vm.overcommit_memory=1' ←
for this to take effect.
[1] 25 Jun 22:11:39.417 * The server is now ready to accept ←
connections on port 6379
[1] 25 Jun 22:11:39.417 * Connecting to MASTER redis_primary:6379
[1] 25 Jun 22:11:39.422 * MASTER <-> SLAVE sync started
[1] 25 Jun 22:11:39.422 * Non blocking connect for SYNC fired the←
event.
[1] 25 Jun 22:11:39.422 * Master replied to PING, replication can←
continue...
[1] 25 Jun 22:11:39.423 * Partial resynchronization not possible ←
(no cached master)
[1] 25 Jun 22:11:39.424 * Full resync from master: 24←
a790df6bf4786a0e886be4b34868743f6145cc:1625
[1] 25 Jun 22:11:39.476 * MASTER <-> SLAVE sync: receiving 18 ←
bytes from master
[1] 25 Jun 22:11:39.476 * MASTER <-> SLAVE sync: Flushing old ←
data
[1] 25 Jun 22:11:39.476 * MASTER <-> SLAVE sync: Loading DB in ←
memory
```

And again, we're off and away replicating!

Creating our Node container

Now that we've got our Redis cluster running, we can launch a container for our Node.js application.

Listing 6.52: Running our Node.js container

```
$ sudo docker run -d \
--name nodeapp -p 3000:3000 \
--link redis_primary:redis_primary \
jamtur01/nodejs
9a9dd33957c136e98295de7405386ed2c452e8ad263a6ec1a2a08b24f80fd175
```

We've created a new container from our `jamtur01/nodejs` image, specified a name of `nodeapp`, and mapped port `3000` inside the container to port `3000` outside. We've also linked our new `nodeapp` container to the `redis_primary` container with an alias of `redis_primary`.

We can use `docker logs` to see what's going on in our `nodeapp` container.

Listing 6.53: The nodeapp console log

```
$ sudo docker logs nodeapp
Listening on port 3000
```

Here we can see that our Node application is bound and listening at port `3000`.

Let's browse to our Docker host and see the application at work.



Figure 6.7: Our Node application.

We can see that our simple Node application returns an OK status.

Listing 6.54: Node application output

```
{  
  "status": "ok"  
}
```

That tells us it's working. Our session state will also be recorded and stored in our primary Redis container, `redis_primary`, then replicated to our Redis replicas: `redis_replica1` and `redis_replica2`.

Capturing our application logs

Now that our application is up and running, we'll want to put it into production, which involves ensuring that we capture its log output and put it into our logging servers. We are going to use [Logstash](#) to do so. We're going to start by creating an image that installs Logstash.

Listing 6.55: Creating our Logstash Dockerfile

```
$ mkdir logstash  
$ cd logstash  
$ touch Dockerfile
```

Now let's populate our Dockerfile.

Listing 6.56: Our Logstash image

```
FROM ubuntu:14.04
MAINTAINER James Turnbull <james@example.com>
ENV REFRESHED_AT 2014-06-01

RUN apt-get -yqq update
RUN apt-get -yqq install wget
RUN wget -O - http://packages.elasticsearch.org/GPG-KEY-←
  elasticsearch | apt-key add -
RUN echo 'deb http://packages.elasticsearch.org/logstash/1.4/←
  debian stable main' > /etc/apt/sources.list.d/logstash.list
RUN apt-get -yqq update
RUN apt-get -yqq install logstash

ADD logstash.conf /etc/

WORKDIR /opt/logstash

ENTRYPOINT [ "bin/logstash" ]
CMD [ "--config=/etc/logstash.conf" ]
```

We've created an image that installs Logstash and adds a `logstash.conf` file to the `/etc/` directory using the `ADD` instruction. Let's quickly look at this file.

Listing 6.57: Our Logstash configuration

```

input {
  file {
    type => "syslog"
    path => ["/var/log/nodeapp/nodeapp.log", "/var/log/redis/←
              redis-server.log"]
  }
}
output {
  stdout {
    codec => rubydebug
  }
}

```

This is a very simple Logstash configuration that monitors two files: `/var/log/nodeapp/nodeapp.log` and `/var/log/redis/redis-server.log`. Logstash will watch these files and send any new data inside of them into Logstash. The second part of our configuration, the `output` stanza, takes any events Logstash receives and outputs them to standard out. In a real world Logstash configuration we would output to an Elasticsearch cluster or other destination, but we're just using this as a demo, so we're going to skip that.

NOTE If you don't know much about Logstash, you can learn more [from my book](#) or the [Logstash documentation](#).

We've specified a working directory of `/opt/logstash`. Finally, we have specified an `ENTRYPOINT` of `bin/logstash` and a `CMD` of `--config=/etc/logstash.conf` to pass in our command flags. This will launch Logstash and load our `/etc/logstash.conf` configuration file.

Let's build our Logstash image now.

Listing 6.58: Building our Logstash image

```
$ sudo docker build -t jamtur01/logstash .
```

Now that we've built our Logstash image, we can launch a container from it.

Listing 6.59: Launching a Logstash container

```
$ sudo docker run -d --name logstash \
--volumes-from redis_primary \
--volumes-from nodeapp \
jamtur01/logstash
```

We've launched a new container called `logstash` and specified the `--volumes-from` flag twice to get the volumes from the `redis_primary` and `nodeapp`. This gives us access to the Node and Redis log files. Any events added to those files will be reflected in the volumes in the `logstash` container and passed to Logstash for processing.

Let's look at that now by examining the logs of the `logstash` container with the `-f` flag to follow the log.

Listing 6.60: The logstash container's logs

```
$ sudo docker logs -f logstash
{:timestamp=>"2014-06-26T00:41:53.273000+0000", :message=>"Using ←
  milestone 2 input plugin 'file'. This plugin should be stable, ←
  but if you see strange behavior, please let us know! For more ←
  information on plugin milestones, see http://logstash.net/docs←
  /1.4.2-modified/plugin-milestones", :level=>:warn}
```

Let's browse to our web application again and refresh it to generate an event. We should see that event reflected in our `logstash` container output.

Listing 6.61: A Node event in Logstash

```
{
  "message" => "63.239.94.10 - - [Thu, 26 Jun 2014 01:28:42 +0000] \"GET /hello/frank HTTP/1.1\" 200 22 \"-\" \"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/35.0.1916.153 Safari/537.36\"",
  "@version" => "1",
  "@timestamp" => "2014-06-26T01:28:42.593Z",
  "type" => "syslog",
  "host" => "cfa96519ba54",
  "path" => "/var/log/nodeapp/nodeapp.log"
}
```

And now we have our Node and Redis containers logging to Logstash. In a production environment, we'd be sending these events to a Logstash server and storing them in Elasticsearch. We could also easily add our Redis replica containers or other components of the solution to our logging environment.

NOTE We could also do Redis backups via volumes if we wanted to.

Summary of our Node stack

We've now seen a multi-container application stack. We've used Docker links to connect our application together and Docker volumes to help manage a variety of aspects of our application. We can easily build on this foundation to produce more complex applications and architectures.

Managing Docker containers without SSH

Lastly, before we wrap up our chapter on running services with Docker, it's important to understand some of the ways we can manage Docker containers and how those differ from some more traditional management techniques.

Traditionally, when managing services, we're used to SSHing into our environment or virtual machines to manage them. In the Docker world, where most containers run a single process, this access isn't available. As we've seen much of the time, this access isn't needed: we can use volumes or links to perform a lot of the same actions. For example, if our service is managed via a network interface, we can expose that on a container; if our service is managed through a Unix socket, we can expose that with a volume. If we need to send a signal to a Docker container, we can also use the `docker kill` command, like so:

Listing 6.62: Using docker kill to send signals

```
$ sudo docker kill -s <signal> <container>
```

This will send the specific signal you want (e.g., a HUP) to the container in question rather than killing the container.

Sometimes, however, we do need to sign into a container. To do that, though, we don't need to run an SSH service or open up any access. We can use a small tool called `nsenter`.

NOTE The use of `nsenter` generally applies to Docker 1.2 and earlier releases. The `docker exec` command introduced in Docker 1.3 replaces much of this functionality.

The `nsenter` tool allows us to enter into the kernel namespaces that Docker uses to construct its containers. Technically, it can enter existing namespaces, or spawn a process into a new set of namespaces. The short version is this: with `nsenter`, we can get a shell into an existing container, even if that container doesn't run SSH or any kind of special-purpose daemon. We can install `nsenter` via a Docker

container.

Listing 6.63: Installing nsenter

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

This will install `nsenter` in `/usr/local/bin`, and you will be able to use it immediately.

TIP The `nsenter` tool might also be available in your Linux distribution (in the `util-linux` package.)

To use `nsenter`, we'll first need to get the process ID, or PID, of the container we wish to enter. To do so, we can use the `docker inspect` command.

Listing 6.64: Finding the process ID of the container

```
PID=$(sudo docker inspect --format '{{.State.Pid}}' <container>)
```

We can then enter the container:

Listing 6.65: Entering a container with nsenter

```
$ sudo nsenter --target $PID --mount --uts --ipc --net --pid
```

This will launch a shell inside the container, without needing SSH or any other special daemon or process.

We can also run commands inside a container by appending the command to the `nsenter` command line.

Listing 6.66: Running a command inside a container with nsenter

```
$ sudo nsenter --target $PID --mount --uts --ipc --net --pid ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  .  .
.
```

This will run the `ls` command inside our target container.

Summary

In this chapter, we've seen how to build some example production services using Docker containers. We've seen a bit more about how we can build multi-container services and manage those stacks. We've combined features like Docker links and volumes and learned how to potentially extend those features to provide us with capabilities like logging and backups.

In the next chapter, we'll look at orchestration with Docker using the Fig and Consul tools.

Chapter 7

Docker Orchestration and Service Discovery

Orchestration is a pretty loosely defined term. It's broadly the process of automated configuration, coordination, and management of services. In the Docker world we use it to describe the set of practices around managing applications running in multiple Docker containers and potentially across multiple Docker hosts. Native orchestration is in its infancy in the Docker community but an exciting ecosystem of tools is being integrated and developed.

In the current ecosystem there are a variety of tools being built and integrated with Docker. Some of these tools are simply designed to elegantly "wire" together multiple containers and build application stacks using simple composition. Other tools provide larger scale coordination between multiple Docker hosts as well as complex service discovery, scheduling and execution capabilities.

Each of these areas really deserves their own book but we've focussed on a couple of useful tools that give you some insight into what you can achieve when orchestrating containers. They hopefully provide some useful building blocks upon which you can grow your Docker-enabled environment.

In this chapter we've focused on two areas:

- Simple container orchestration. Here we've looked at [Fig](#). Fig is an open source Docker orchestration tool developed by the Orchard team and then

acquired by Docker Inc in 2014. It's written in Python and licensed with the Apache 2.0 license.

- Distributed service discovery. Here we've introduced [Consul](#). Consul is also open source, licensed with the Mozilla Public License 2.0, and written in Go. It provides distributed, highly available service discovery. We're going to look at how you might use Consul and Docker to manage application service discovery.

TIP We'll talk about many of the other orchestration tools available to you later in this chapter.

Fig

Now let's get familiar with Fig. With Fig, we define a set of containers to boot up, and their runtime properties, all defined in a YAML file. Fig calls each of these containers "services" which it defines as:

A container that interacts with other containers in some way and that has specific runtime properties.

We're going to take you through installing Fig and then using it to build a simple, multi-container application stack.

Installing Fig

We start by installing Fig. Fig is currently available for Linux and OS X. It can be installed directly as a binary or via a Python Pip package.

NOTE Fig doesn't currently work with Boot2Docker. This is being worked on but it is not quite ready for prime time yet!

To install Fig on Linux we can grab the Fig binary from Git Hub and make it executable. Like Docker, Fig is currently only supported on 64-bit Linux installations. We'll need the `curl` command available to do this.

Listing 7.1: Installing Fig on Linux

```
$ sudo bash -c "curl -L https://github.com/docker/fig/releases/←  
download/0.5.2/linux/ > /usr/local/bin/fig"  
$ sudo chmod +x /usr/local/bin/fig
```

This will download the `fig` binary from Git Hub and install it into the `/usr/←
local/bin` directory. We've also used the `chmod` command to make the `fig` binary executable so we can run it.

If we're on OS X we can do the same like so:

Listing 7.2: Installing Fig on OS X

```
$ curl -L https://github.com/docker/fig/releases/download/0.5.2/←  
darwin > /usr/local/bin/fig  
$ chmod +x /usr/local/bin/fig
```

Fig is also available as a Python package if you're on another platform or if you prefer installing via package. You will need to have the Python-Pip tool installed to use the `pip` command. This is available via the `python-pip` package on most Red Hat, Debian and Ubuntu releases.

Listing 7.3: Installing Fig via Pip

```
$ sudo pip install -U fig
```

Once you have installed the `fig` binary you can test it's working using the `fig` command with the `--version` flag:

Listing 7.4: Testing Fig is working

```
$ fig --version
fig 0.5.2
```

Getting our sample application

To demonstrate how Fig works we're going to use a sample Python Flask application that combines two containers:

- An application container running our sample Python application.
- A Redis container running the Redis database.

Let's start with building our sample application. Firstly, we create a directory and a `Dockerfile`.

Listing 7.5: Creating the figapp directory

```
$ mkdir figapp
$ cd figapp
$ touch Dockerfile
```

Here we've created a directory to hold our sample application, which we're calling `figapp`. We've changed into that directory and created an empty `Dockerfile` to hold our Docker image build instructions.

Next, we need to add our application code. Let's create a file called `app.py` and add the following Python code to it.

Listing 7.6: The app.py file

```
from flask import Flask
from redis import Redis
import os

app = Flask(__name__)
redis = Redis(host="redis_1", port=6379)

@app.route('/')
def hello():
    redis.incr('hits')
    return 'Hello Docker Book reader! I have been seen {0} times'←
        .format(redis.get('hits'))

if __name__ == "__main__":
    app.run(host="0.0.0.0", debug=True)
```

TIP You can find this source code on Git Hub [here](#) or on the Docker Book [site](#).

This simple Flash application tracks a counter stored in Redis. The counter is incremented each time the root URL, /, is hit.

We also need to create a requirements.txt file to store our application's dependencies. Let's create that file now and add the following dependencies.

Listing 7.7: The requirements.txt file

```
flask
redis
```

Now let's populate our Fig Dockerfile.

Listing 7.8: The figapp Dockerfile

```
# Fig Sample application image
FROM python:2.7
MAINTAINER James Turnbull <james@example.com>

ADD . /figapp

WORKDIR /figapp

RUN pip install -r requirements.txt
```

Our Dockerfile is very simple. It is based on the `python:2.7` image. We add our `app.py` and `requirements.txt` files into a directory in the image called `/figapp`. The Dockerfile then sets the working directory to `/figapp` and runs the `pip` installation process to install our application's dependencies: `flask` and `redis`.

Let's build that image now using the `docker build` command.

Listing 7.9: Building the figapp application

```
$ cd figapp
$ sudo docker build -t="jamtur01/figapp" .
Sending build context to Docker daemon 16.9 kB
Sending build context to Docker daemon
Step 0 : FROM python:2.7
--> 1c8df2f0c10b
Step 1 : MAINTAINER James Turnbull <james@example.com>
--> Using cache
--> aa564fe8be5a
Step 2 : ADD . /figapp
--> c33aa147e19f
Removing intermediate container 0097bc79d37b
Step 3 : WORKDIR /figapp
--> Running in 76e5ee8544b3
--> d9da3105746d
Removing intermediate container 76e5ee8544b3
Step 4 : RUN pip install -r requirements.txt
--> Running in e71d4bb33fd2
Downloading/unpacking flask (from -r requirements.txt (line 1))
...
Successfully installed flask redis Werkzeug Jinja2 itsdangerous ←
markupsafe
Cleaning up...
--> bf0fe6a69835
Removing intermediate container e71d4bb33fd2
Successfully built bf0fe6a69835
```

This will build a new image called `jamtur01/figapp` containing our sample application and its required dependencies. We can now use Fig to deploy our application.

NOTE We'll be using a Redis container created from the default Redis image

on the Docker Hub so we don't need to build or customize that.

The `fig.yml` file

Now we've got our application container setup we can configure Fig to create both the services we require. With Fig, we define a set of services (in the form of Docker containers) to boot up. We also define the runtime properties we want these services to start with, much as you would do with the `docker run` command. We define all of this in a YAML file. We then run the `fig up` command. Fig boots the containers, executes the appropriate runtime configuration, and multiplexes the log output together for us.

Let's create a `fig.yml` file for our application.

Listing 7.10: Creating the `fig.yml` file

```
$ cd figapp
$ touch fig.yml
```

Let's populate our `fig.yml` file. The `fig.yml` file is a YAML file that contains instructions for running one or more Docker containers. Let's look the instructions for our example application.

Listing 7.11: The fig.yml file

```

web:
  image: jamtur01/figapp
  command: python app.py
  ports:
    - "5000:5000"
  volumes:
    - .:/figapp
  links:
    - redis
redis:
  image: redis

```

Each service we wish to launch is specified as a YAML hash here: `web` and `redis`.

For our `web` service we've specified some runtime options. Firstly, we've specified the `image` we're using. In our case the `jamtur01/figapp` image. Fig can also build Docker images. You can use the `build` instruction and provide the path to a `Dockerfile` to have Fig build an image and then create services from it.

Listing 7.12: The build instruction

```

web:
  build: /home/james/figapp
  ...

```

This `build` instruction would build a Docker image from a `Dockerfile` found in the `/home/james/figapp` directory.

We've also specified the `command` to run when launching the service. Next we specify the `ports` and `volumes` as a list of the port mappings and volumes we want for our service. We've specified that we're mapping port 5000 inside our service to port 5000 on the host. We're also creating `/figapp` as a volume. Finally, we specify any links for this service. Here we link our `web` service to the `redis` service.

If we were executing the same configuration on the command line using `docker-run` we'd do it like so:

Listing 7.13: The docker run equivalent command

```
$ sudo docker run -d -p 5000:5000 -v .:/figapp --link redis:redis<-
 \
 --name jamtur01/figapp python app.py
```

Next we've specified another service called `redis`. For this service we're not setting any runtime defaults at all. We're just going to use the base `redis` image. By default, containers run from this image launches a Redis database on the standard port. So we don't need to configure or customize it.

TIP You can see a full list of the available instructions you can use in the `fig.yml` file [here](#).

Running Fig

Once we've specified our services in `fig.yml` we use the `fig up` command to execute them both.

Listing 7.14: Running fig up with our sample application

TIP You must be inside the directory with the `fig.yml` file in order to execute most Fig commands.

We can see that Fig has created two new services: `figapp_redis_1` and `figapp_web_1`. So where did these names come from? Well, to ensure our services are unique, Fig has prefixed and suffixed the names specified in the `fig.yml` file with the directory and a number respectively.

Fig then attaches to the logs of each service, each line of log output is prefixed with the abbreviated name of the service it comes from, and outputs them multiplexed:

Listing 7.15: Fig service log output

```
redis_1 | [1] 13 Aug 01:48:32.218 # Server started, Redis version←  
2.8.13
```

The services (and Fig) are being run interactively. That means if you use `Ctrl-C` or the like to cancel Fig then it'll stop the running services. We could also run Fig with `-d` flag to run our services daemonized (similar to the `docker run -d` flag).

Listing 7.16: Running Fig daemonized

```
$ sudo fig up -d
```

Let's look at the sample application that's now running on the host. The application is bound to all interfaces on the Docker host on port 5000. So we can browse to that site on the host's IP address or via `localhost`.

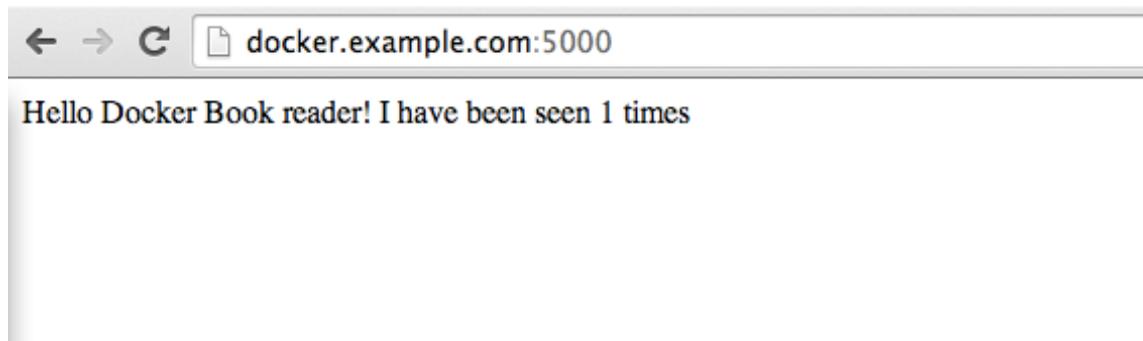


Figure 7.1: Sample Fig application.

We can see a message displaying the current counter value. We can increment the counter by refreshing the site. Each refresh stores the increment in Redis. The Redis update is done via the link between the Docker containers controlled by Fig.

TIP By default, Fig tries to connect to a local Docker daemon but it'll also honor

the `DOCKER_HOST` environment variable to connect to a remote Docker host.

Using Fig

Now let's explore some of Fig's other options. Firstly, let's use `Ctrl-C` to cancel our running services and then restart them as daemonized services.

Press `Ctrl-C` inside the `figapp` directory and then re-run the `fig up` command, this time with the `-d` flag.

Listing 7.17: Restarting Fig as daemonized

```
$ sudo fig up -d
Recreating figapp_redis_1...
Recreating figapp_web_1...
$ . . .
```

We can see that Fig has recreated our services, launched them and returned to the command line.

Our Fig-managed services are now running daemonized on the host. Let's look at them now using the `fig ps` command; a close cousin of the `docker ps` command.

TIP You can get help on Fig commands by running `fig help` and the command you wish to get help on, for example `fig help ps`.

The `fig ps` command lists all of the currently running services from our local `fig.yml` file.

Listing 7.18: Running the fig ps command

```
$ cd figapp
$ sudo fig ps
      Name      Command      State      Ports
-----
figapp_redis_1  redis-server  Up      6379/tcp
figapp_web_1    python app.py  Up      5000->5000/tcp
```

This shows some basic information about our running Fig services. The name of each service, what command we used to start the service, and the ports that are mapped on each service.

We can also drill down further using the `fig logs` command to show us the log events from our services.

Listing 7.19: Showing a Fig services logs

```
$ sudo fig logs
fig logs
Attaching to figapp_redis_1, figapp_web_1
redis_1 | (          ,          .-' | `,      )      Running in ←
      stand alone mode
redis_1 | |`-.`-....`__....`--`-..| `` _.-'|      Port: 6379
redis_1 | | |`-.`-..`_.` /      _.-'|      PID: 1
. . .
```

This will tail the log files of your services, much as the `tail -f` command. Like the `tail -f` command you'll need to use `Ctrl-C` or the like to exit from it.

We can also stop our running services with the `fig stop` command.

Listing 7.20: Stopping running services

```
$ sudo fig stop
Stopping figapp_web_1...
Stopping figapp_redis_1...
```

This will stop both services. If the services don't stop you can use the `fig kill`

command to force kill the services.

We can verify this with the `fig ps` command again.

Listing 7.21: Verifying our Fig services have been stopped

```
$ sudo fig ps
      Name      Command      State      Ports
-----
figapp_redis_1  redis-server  Exit 0
figapp_web_1    python app.py  Exit 0
```

If you've stopped services using `fig stop` or `fig kill` you can also restart with again with the `fig start` command. This is much like using the `docker start` command and will restart these services.

Finally, we can remove services using the `fig rm` command.

Listing 7.22: Removing Fig services

```
$ sudo fig rm
Going to remove figapp_redis_1, figapp_web_1
Are you sure? [yN] y
Removing figapp_redis_1...
Removing figapp_web_1...
```

You'll be prompted to confirm you wish to remove the services and then both services will be deleted. The `fig ps` command will now show no running or stopped services.

Listing 7.23: Showing no Fig services

```
$ sudo fig ps
      Name      Command      State      Ports
-----

```

Fig in summary

Now in one file we have a simple Python-Redis stack built! You can see how much easier this can make constructing applications from multiple Docker containers. This, however, just scratches the surface of what you can do with Fig. There are some more examples using [Rails](#), [Django](#) and [Wordpress](#) on the Fig website that introduce some more advanced concepts. You can also use Fig in conjunction with [Shipyard](#) to provide a graphical user interface.

TIP You can see a full command line reference [here](#).

Consul, Service Discovery and Docker

Service discovery is the mechanism by which distributed applications manage their relationships. A distributed application is usually made up of multiple components. These components can be located together locally or distributed across data centres or geographical regions. Each of these components usually provides or consumes services to or from other components.

Service discovery allows these components to find each other when they want to interact. Due to the distributed nature of these applications, service discovery mechanisms also need to be distributed. As they are usually the "glue" between components of distributed applications they also need to be dynamic, reliable, resilient and able to quickly and consistently share data about these services.

Docker, with its focus on distributed applications and service orientated and microservices architectures, is an ideal candidate for integration with a service discovery tool. Each Docker container can register its running service or services with the tool. This provides the information needed, for example an IP address or port or both, to allow interaction between services.

Our example service discovery tool, [Consul](#), is a specialized datastore that uses consensus algorithms. Consul specifically uses the [Raft](#) consensus algorithm, to

require a quorum for writes. It also exposes a key value store and service catalog that is highly available, fault-tolerant, and maintains strong consistency guarantees. Services can register themselves with Consul and share that registration information in a highly-available and distributed manner.

Consul is also interesting because it provides:

- A service catalog with an API instead of the traditional key=value store of most service discovery tools.
- Both a DNS-based query interface through inbuilt DNS server and a HTTP-based REST API to query the information. The choice of interfaces, especially the DNS-based interface, allows you to easily drop Consul into your existing environment.
- Service monitoring AKA health checks. Consul has powerful service monitoring built into the tool.

To get a better understanding of how Consul works, we're going to see how to run distributed Consul inside Docker containers. We're then going to register services from Docker containers to Consul and query that data from other Docker containers. To make it more interesting we're going to do this across multiple Docker hosts.

To do this we're going to:

- Create a Docker image for the Consul service.
- Build three hosts running Docker and then run Consul on each. The three hosts will provide us with a distributed environment to see how resiliency and failover works with Consul.
- Build services that we'll register with Consul and then query that data from another service.

NOTE You can see a more generic introduction to Consul [here](#).

Building a Consul image

We're going to start with creating a Dockerfile to build our Consul image. Let's create a directory to hold our Consul image first.

Listing 7.24: Creating a Consul Dockerfile directory

```
$ mkdir consul
$ cd consul
$ touch Dockerfile
```

Now let's look at the Dockerfile for our Consul image.

Listing 7.25: The Consul Dockerfile

```

FROM ubuntu:14.04
MAINTAINER James Turnbull <james@example.com>
ENV REFRESHED_AT 2014-08-01

RUN apt-get -qqy update
RUN apt-get -qqy install curl unzip

ADD https://dl.bintray.com/mitchellh/consul/0.3.1_linux_amd64.zip /tmp/consul.zip
RUN cd /usr/sbin && unzip /tmp/consul.zip && chmod +x /usr/sbin/consul && rm /tmp/consul.zip

ADD https://dl.bintray.com/mitchellh/consul/0.3.1_web_ui.zip /tmp/webui.zip
RUN cd /tmp/ && unzip webui.zip && mv dist/ /webui/

ADD consul.json /config/

EXPOSE 53/udp 8300 8301 8301/udp 8302 8302/udp 8400 8500

VOLUME [ "/data" ]

ENTRYPOINT [ "/usr/sbin/consul", "agent", "-config-dir=/config" ]
CMD []

```

Our Dockerfile is pretty simple. It's based on an Ubuntu 14.04 image. It installs curl and unzip. We then download the Consul zip file containing the consul binary. We move that binary to /usr/sbin/ and make it executable. We also download Consul's web interface and place it into a directory called /webui. We're going to see this web interface in action a little later.

We then add a configuration file for Consul, `consul.json`, to the `/config` directory. Let's look at that file now.

Listing 7.26: The `consul.json` configuration file

```
{
  "data_dir": "/data",
  "ui_dir": "/webui",
  "client_addr": "0.0.0.0",
  "ports": {
    "dns": 53
  },
  "recursor": "8.8.8.8"
}
```

The `consul.json` configuration file is JSON formatted and provides Consul with the information needed to get running. We've specified a data directory, `/data`, to hold Consul's data. We also specify the location of the web interface files: `/webui`. We use the `client_addr` variable to bind Consul to all interfaces inside our container.

We also use the `ports` block to configure on which ports various Consul services run. In this case we're specifying that Consul's DNS service should run on port 53. Lastly, we've used the `recursor` option to specify a DNS server to use for resolution if Consul can't resolve a DNS request. We've specified `8.8.8.8` which is one of the IP addresses of [Google's public DNS service](#).

TIP You can find the full list of available Consul configuration options [here](#).

Back in our Dockerfile we've use the `EXPOSE` instruction to open up a series of ports that Consul requires to operate. I've added a table showing each of these ports and what they do.

Table 7.1: Consul's default ports.

Port	Purpose
53/udp	DNS server

Port	Purpose
8300	Server RPC
8301 + udp	Serf LAN port
8302 + udp	Serf WAN port
8400	RPC endpoint
8500	HTTP API

You don't need to worry about most of them for the purposes of this chapter. The important ones for us are 53/udp which is the port Consul is going to be running DNS on. We're going to use DNS to query service information. We're also going to use Consul's HTTP API and its web interface, both of which are bound to port 8500. The rest of the ports handle the backend communication and clustering between Consul nodes. We'll configure them in our Docker container but we don't do anything specific with them.

NOTE You can find more details of what each port does [here](#).

Next, we've also made our `/data` directory a volume using the `VOLUME` instruction. This is useful if we want to manage or work with this data as we saw in Chapter 6.

Finally, we've specified an `ENTRYPOINT` instruction to launch Consul using the `consul` binary when a container is launched from our image.

Let's step through the command line options we've used. We've specified the `consul` binary in `/usr/sbin/`. We've passed it the `agent` command which tells Consul to run as an agent and the `-config-dir` flag and specified the location of our `consul.json` file in the `/config` directory.

Let's build our image now.

Listing 7.27: Building our Consul image

```
$ sudo docker build -t="jamtur01/consul" .
```

NOTE You can get our Consul Dockerfile and configuration file [here](#) or on [GitHub](#).

Testing a Consul container locally

Before we run Consul on multiple hosts, let's see it working locally on a single host. To do this we'll run a container from our new `jamtur01/consul` image.

Listing 7.28: Running a local Consul node

```
$ sudo docker run -p 8500:8500 -p 53:53/udp \
-h node1 jamtur01/consul -server -bootstrap
==> Starting Consul agent...
==> Starting Consul agent RPC...
==> Consul agent running!
      Node name: 'node1'
      Datacenter: 'dc1'
.
.
.
2014/08/25 21:47:49 [WARN] raft: Heartbeat timeout reached, ←
      starting election
2014/08/25 21:47:49 [INFO] raft: Node at 172.17.0.26:8300 [←
      Candidate] entering Candidate state
2014/08/25 21:47:49 [INFO] raft: Election won. Tally: 1
2014/08/25 21:47:49 [INFO] raft: Node at 172.17.0.26:8300 [Leader←
      ] entering Leader state
2014/08/25 21:47:49 [INFO] consul: cluster leadership acquired
2014/08/25 21:47:49 [INFO] consul: New leader elected: node1
2014/08/25 21:47:49 [INFO] consul: member 'node1' joined, marking←
      health alive
```

We've used the `docker run` command to create a new container. We've mapped two ports, port 8500 in the container to 8500 on the host and port 53 in the con-

tainer to 53 on the host. We've also used the `-h` flag to specify the hostname of the container, here `node1`. This is going to be both the hostname of the container and the name of the Consul node. We've then specified the name of our Consul image, `jamtur01/consul`.

Lastly, we've passed two flags to the `consul` binary: `-server` and `-bootstrap`. The `-server` flag tells the Consul agent to operate in server mode. The `-bootstrap` flag tells Consul that this node is allowed to self-elect as a leader. This allows us to see a Consul agent in server mode doing a Raft leadership election.

WARNING It is important that no more than one server per datacenter be running in bootstrap mode. Otherwise consistency cannot be guaranteed if multiple nodes are able to self-elect. We'll see some more on this when we add other nodes to the cluster.

We can see that Consul has started `node1` and done a local leader election. As we've got no other Consul nodes running it is not connected to anything else.

We can also see this via the Consul web interface if we browse to our local host's IP address on port 8500.

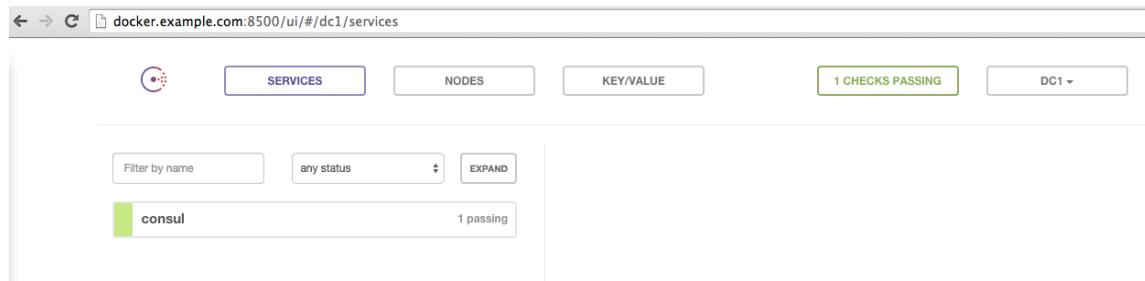


Figure 7.2: The Consul web interface.

Running a Consul cluster in Docker

As Consul is distributed we'd normally create three (or more) hosts to run in separate data centres, clouds or regions. Or even add an agent to every application

server. This will provide us with sufficient distributed resilience. We're going to mimic this required distribution by creating three hosts each with a Docker daemon to run Consul. We've created three new Ubuntu 14.04 hosts: `larry`, `curly`, and `moe`. On each host we've installed a Docker daemon. We've also pulled down the `jamtur01/consul` image.

Listing 7.29: Pulling down the Consul image

```
$ sudo docker pull jamtur01/consul
```

On each host we're going to run a Docker container with the `jamtur01/consul` image. To do this we need to choose a network to run Consul over. In most cases this would be a private network but as we're just simulating a Consul cluster I am going to use the public interfaces of each host. To start Consul on this public network I am going to need the public IP address of each host. This is the address we're going to bind each Consul agent too.

Let's grab that now on `larry` and assign it to an environment variable, `$PUBLIC_UP`.

Listing 7.30: Assigning public IP on larry

```
larry$ PUBLIC_IP="$(ifconfig eth0 | awk -F ' *|: ' '/inet addr/{  
    print $4}'")  
larry$ echo $PUBLIC_IP  
104.131.38.54
```

And then create the same `$PUBLIC_IP` variable on `curly` and `moe` too.

Listing 7.31: Assigning public IP on curly and moe

```
curly$ PUBLIC_IP=$(ifconfig eth0 | awk -F ' *|:' '/inet addr/{←
    print $4}'")
curly$ echo $PUBLIC_IP
104.131.38.55
moe$ PUBLIC_IP=$(ifconfig eth0 | awk -F ' *|:' '/inet addr/{←
    print $4}'")
moe$ echo $PUBLIC_IP
104.131.38.56
```

We can see we've got three hosts and three IP addresses, each assigned to the \$PUBLIC_IP environmental variable.

Table 7.2: Consul host IP addresses

Host	IP Address
larry	104.131.38.54
curly	104.131.38.55
moe	104.131.38.56

We're also going to need to nominate a host to bootstrap to start the cluster. We're going to choose `larry`. This means we'll need `larry`'s IP address on `curly` and `moe` to tell them which Consul node's cluster to join. Let's set that up now by adding `larry`'s IP address of `104.131.38.54` to `curly` and `moe` as the environment variable, `$JOIN_IP`.

Listing 7.32: Adding the cluster IP address

```
curly$ JOIN_IP=104.131.38.54
moe$ JOIN_IP=104.131.38.54
```

Finally, we've made one network configuration change to the Docker daemon on each host to make it easier to use Consul. We've configured the DNS lookup of the Docker daemon to use:

- The local Docker IP address so we can use Consul to resolve DNS.

- Google DNS to resolve any other queries.
- Specified a search domain for Consul queries.

To do this we first need the IP address of the Docker interface docker0.

Listing 7.33: Getting the docker0 IP address

```
larry$ ip addr show docker0
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc ↵
    noqueue state UP group default
        link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
        inet 172.17.42.1/16 scope global docker0
            valid_lft forever preferred_lft forever
        inet6 fe80::5484:7aff:fe:9799/64 scope link
            valid_lft forever preferred_lft forever
```

We can see the interface has the IP of 172.17.42.1.

We've taken this address and altered Docker's startup options in the /etc/←
default/docker file from:

Listing 7.34: Original Docker defaults

```
#DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4"
```

To:

Listing 7.35: New Docker defaults on larry

```
DOCKER_OPTS='--dns 172.17.42.1 --dns 8.8.8.8 --dns-search service←
    .consul'
```

We now do the same on curly and moe, finding the docker0 IP address and updating the DOCKER_OPTS flag in /etc/default/docker.

TIP On other distributions you'd updated the Docker daemon defaults using the appropriate mechanism. See Chapter 2 for further information.

We then restart the Docker daemon on each host, for example:

Listing 7.36: Restarting the Docker daemon on larry

```
larry$ sudo service docker restart
```

Starting the Consul bootstrap node

Let's start our initial bootstrap node on `larry`. Our `docker run` command is going to be a little complex because we're mapping a lot of ports. Indeed, we need to map all the ports listed in Table 7.1 above. And, as we're both running Consul in a container and connecting to containers on other hosts, we're going to map each port to the corresponding port on the local host. This will allow both internal and external access to Consul.

Let's see our `docker run` command now.

Listing 7.37: Start the Consul bootstrap node

```
larry$ sudo docker run -d -h $HOSTNAME \
-p 8300:8300 -p 8301:8301 \
-p 8301:8301/udp -p 8302:8302 \
-p 8302:8302/udp -p 8400:8400 \
-p 8500:8500 -p 53:53/udp \
--name larry_agent jamtur01/consul \
-server -advertise $PUBLIC_IP -bootstrap-expect 3
```

Here we've launched a daemonized container using the `jamtur01/consul` image to run our Consul agent. We can see we've set the `-h` flag to set the hostname of the container to the value of the `$HOSTNAME` environment variable. This sets our Consul agent's name to be the local hostname, here `larry`. We're also mapped a series of eight ports from inside the container to the respective ports on the local host.

We've also specified some command line options for the Consul agent.

Listing 7.38: Consul agent command line arguments

```
-server -advertise $PUBLIC_IP -bootstrap-expect 3
```

The `-server` flag tell the agent to run in server mode. The `-advertise` flag tells that server to advertise itself on the IP address specified in the `$PUBLIC_IP` environment variable. Lastly, the `-bootstrap-expect` flag tells Consul how many agents to expect in this cluster. In this case, 3 agents. It also bootstraps the cluster.

Let's look at the logs of our initial Consul container with the `docker logs` command.

Listing 7.39: Starting bootstrap Consul node

```

larry$ sudo docker logs larry_agent
==> Starting Consul agent...
==> Starting Consul agent RPC...
==> Consul agent running!
    Node name: 'larry'
    Datacenter: 'dc1'
    Server: true (bootstrap: false)
    Client Addr: 0.0.0.0 (HTTP: 8500, DNS: 53, RPC: 8400)
    Cluster Addr: 104.131.38.54 (LAN: 8301, WAN: 8302)
    Gossip encrypt: false, RPC-TLS: false, TLS-Incoming: false
    ...
2014/08/31 18:10:07 [WARN] memberlist: Binding to public address ←
    without encryption!
2014/08/31 18:10:07 [INFO] serf: EventMemberJoin: larry ←
    104.131.38.54
2014/08/31 18:10:07 [WARN] memberlist: Binding to public address ←
    without encryption!
2014/08/31 18:10:07 [INFO] serf: EventMemberJoin: larry.dc1 ←
    104.131.38.54
2014/08/31 18:10:07 [INFO] raft: Node at 104.131.38.54:8300 [←
    Follower] entering Follower state
2014/08/31 18:10:07 [INFO] consul: adding server larry (Addr: ←
    104.131.38.54:8300) (DC: dc1)
2014/08/31 18:10:07 [INFO] consul: adding server larry.dc1 (Addr:←
    104.131.38.54:8300) (DC: dc1)
2014/08/31 18:10:07 [ERR] agent: failed to sync remote state: No ←
    cluster leader
2014/08/31 18:10:08 [WARN] raft: EnableSingleNode disabled, and ←
    no known peers. Aborting election.

```

We can see that the agent on `larry` is started but because we don't have any more nodes yet no election has taken place. We can see this from the only error returned.

Listing 7.40: Cluster leader error

```
[ERR] agent: failed to sync remote state: No cluster leader
```

Starting the remaining nodes

Now we've bootstrapped our cluster we can start our remaining nodes on `curly` and `moe`. Let's start with `curly`. We use the `docker run` command to launch our second agent.

Listing 7.41: Starting the agent on curly

```
curly$ sudo docker run -d -h $HOSTNAME \
-p 8300:8300 -p 8301:8301 \
-p 8301:8301/udp -p 8302:8302 \
-p 8302:8302/udp -p 8400:8400 \
-p 8500:8500 -p 53:53/udp \
--name curly_agent jamtur01/consul \
-server -advertise $PUBLIC_IP -join $JOIN_IP
```

We see our command is very similar to our bootstrapped node on `larry` with the exception of the command we're passing to the Consul agent.

Listing 7.42: Launching the Consul agent on curly

```
-server -advertise $PUBLIC_IP -join $JOIN_IP
```

Again we've enabled the Consul agent's server mode with `-server` and bound the agent to the public IP address using the `-advertise` flag. Finally, we've told Consul to join our Consul cluster by specifying `larry`'s IP address using the `$JOIN_IP` environment variable.

Let's see what happened when we launched our container.

Listing 7.43: Looking at the Curly agent logs

```
curly$ sudo docker logs curly_agent
==> Starting Consul agent...
==> Starting Consul agent RPC...
==> Joining cluster...
    Join completed. Synced with 1 initial agents
==> Consul agent running!
    Node name: 'curly'
    Datacenter: 'dc1'
    Server: true (bootstrap: false)
    Client Addr: 0.0.0.0 (HTTP: 8500, DNS: 53, RPC: 8400)
    Cluster Addr: 104.131.38.55 (LAN: 8301, WAN: 8302)
    Gossip encrypt: false, RPC-TLS: false, TLS-Incoming: false
    .
    .
2014/08/31 21:45:49 [INFO] agent: (LAN) joining: [104.131.38.54]
2014/08/31 21:45:49 [INFO] serf: EventMemberJoin: larry ←
    104.131.38.54
2014/08/31 21:45:49 [INFO] agent: (LAN) joined: 1 Err: <nil>
2014/08/31 21:45:49 [ERR] agent: failed to sync remote state: No ←
    cluster leader
2014/08/31 21:45:49 [INFO] consul: adding server larry (Addr: ←
    104.131.38.54:8300) (DC: dc1)
2014/08/31 21:45:51 [WARN] raft: EnableSingleNode disabled, and ←
    no known peers. Aborting election.
```

We can see `curly` has joined `larry`, indeed on `larry` we should see something like the following.

Listing 7.44: Curly joining Larry

```
2014/08/31 21:45:49 [INFO] serf: EventMemberJoin: curly ←
    104.131.38.55
2014/08/31 21:45:49 [INFO] consul: adding server curly (Addr: ←
    104.131.38.55:8300) (DC: dc1)
```

But we've still not got a quorum in our cluster, remember we told `-bootstrap` to expect to expect 3 nodes. So let's start our final agent on `moe`.

Listing 7.45: Starting the agent on `curly`

```
moe$ sudo docker run -d -h $HOSTNAME \
-p 8300:8300 -p 8301:8301 \
-p 8301:8301/udp -p 8302:8302 \
-p 8302:8302/udp -p 8400:8400 \
-p 8500:8500 -p 53:53/udp \
--name moe_agent jamtur01/consul \
-server -advertise $PUBLIC_IP -join $JOIN_IP
```

Our `docker run` command is basically the same as what we ran on `curly`. But this time we have three agents in our cluster. Now, if we look at the container's logs, we will see a full cluster.

Listing 7.46: Consul logs on moe

```
moe$ sudo docker logs moe_agent
==> Starting Consul agent...
==> Starting Consul agent RPC...
==> Joining cluster...
    Join completed. Synced with 1 initial agents
==> Consul agent running!
    Node name: 'moe'
    Datacenter: 'dc1'
    Server: true (bootstrap: false)
    Client Addr: 0.0.0.0 (HTTP: 8500, DNS: 53, RPC: 8400)
    Cluster Addr: 104.131.38.56 (LAN: 8301, WAN: 8302)
    Gossip encrypt: false, RPC-TLS: false, TLS-Incoming: false
    ...
2014/08/31 21:54:03 [ERR] agent: failed to sync remote state: No ←
    cluster leader
2014/08/31 21:54:03 [INFO] consul: adding server curly (Addr: ←
    104.131.38.55:8300) (DC: dc1)
2014/08/31 21:54:03 [INFO] consul: adding server larry (Addr: ←
    104.131.38.54:8300) (DC: dc1)
2014/08/31 21:54:03 [INFO] consul: New leader elected: larry
```

We can see from our container's logs that `moe` has joined the cluster. This causes Consul to reach its expected number of cluster members and triggers a leader election. In this case `larry` is elected cluster leader.

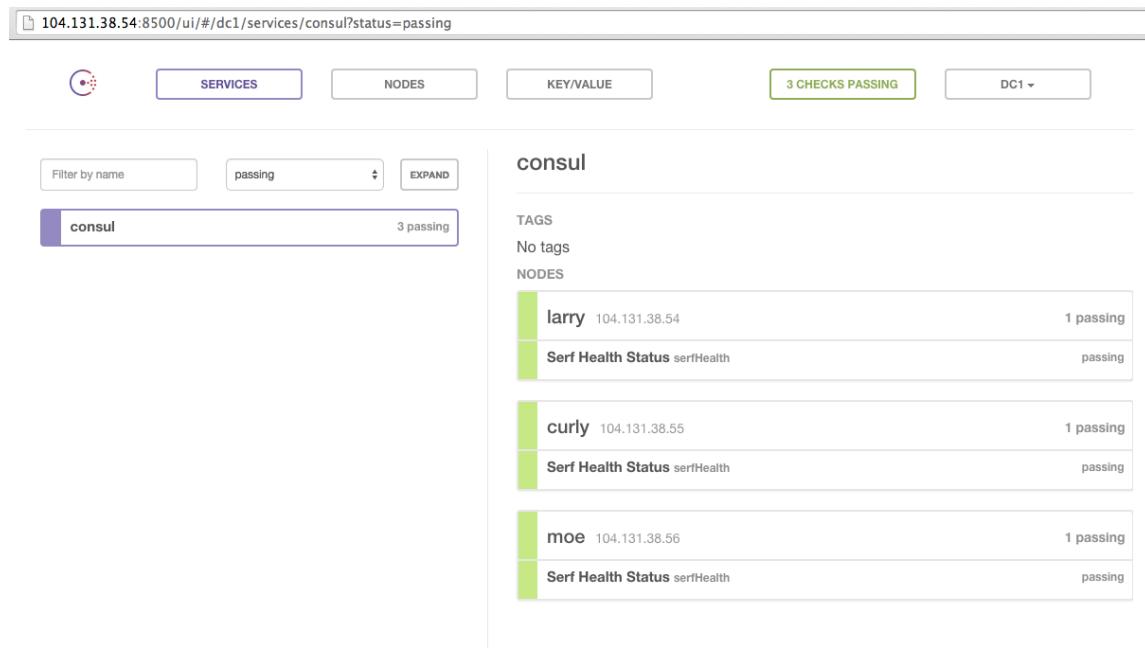
We can see the result of this final agent joining in the Consul logs on `larry` too.

Listing 7.47: Consul leader election on larry

```
2014/08/31 21:54:03 [INFO] consul: Attempting bootstrap with ←
  nodes: [104.131.38.55:8300 104.131.38.56:8300 ←
  104.131.38.54:8300]
2014/08/31 21:54:03 [WARN] raft: Heartbeat timeout reached, ←
  starting election
2014/08/31 21:54:03 [INFO] raft: Node at 104.131.38.54:8300 [←
  Candidate] entering Candidate state
2014/08/31 21:54:03 [WARN] raft: Remote peer 104.131.38.56:8300 ←
  does not have local node 104.131.38.54:8300 as a peer
2014/08/31 21:54:03 [INFO] raft: Election won. Tally: 2
2014/08/31 21:54:03 [INFO] raft: Node at 104.131.38.54:8300 [←
  Leader] entering Leader state
2014/08/31 21:54:03 [INFO] consul: cluster leadership acquired
2014/08/31 21:54:03 [INFO] consul: New leader elected: larry
  . . .
2014/08/31 21:54:03 [INFO] consul: member 'larry' joined, marking←
  health alive
2014/08/31 21:54:03 [INFO] consul: member 'curly' joined, marking←
  health alive
2014/08/31 21:54:03 [INFO] consul: member 'moe' joined, marking ←
  health alive
```

We can also browse to the Consul web interface and select the Consul service to see the current state

Chapter 7: Docker Orchestration and Service Discovery



The screenshot shows the Consul web interface at the URL `104.131.38.54:8500/ui/#/dc1/services/consul?status=passing`. The interface has a navigation bar with tabs for SERVICES, NODES, KEY/VALUE, and a status indicator (3 CHECKS PASSING) and DC1 dropdown. The main content area is titled 'consul' and shows the following data:

Node	Address	Status
larry	104.131.38.54	1 passing
curly	104.131.38.55	1 passing
moe	104.131.38.56	1 passing

Under the 'Tags' section, it says 'No tags'. The 'Nodes' section lists the three nodes with their addresses and status counts.

Figure 7.3: The Consul service in the web interface.

Finally, we can test the DNS is working using the `dig` command.

Listing 7.48: Testing the Consul DNS

```
larry$ dig @172.17.42.1 consul.service.consul

; <>> DiG 9.9.5-3-Ubuntu <>> @172.17.42.1 consul.service.consul
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 13502
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 0, ←
;; ADDITIONAL: 0

;; QUESTION SECTION:
;consul.service.consul.      IN  A

;; ANSWER SECTION:
consul.service.consul. 0  IN  A  104.131.38.55
consul.service.consul. 0  IN  A  104.131.38.54
consul.service.consul. 0  IN  A  104.131.38.56

;; Query time: 2 msec
;; SERVER: 172.17.42.1#53(172.17.42.1)
;; WHEN: Sun Aug 31 21:30:27 EDT 2014
;; MSG SIZE  rcvd: 150
```

Here we've queried the IP of the local Docker interface as a DNS server and asked it to return any information on `consul.service.consul`. This format is Consul's DNS shorthand for services: `consul` is the host and `service.consul` is the domain. Here `consul.service.consul` represent the DNS entry for the Consul service itself.

For example:

Listing 7.49: Querying another Consul service via DNS

```
larry$ dig @172.17.42.1 webservice.service.consul
```

Would return all DNS A records for the service `webservice`.

TIP You can see more details on Consul's DNS interface [here](#).

We now have a running Consul cluster inside Docker containers running on three separate hosts. That's pretty cool but it's not overly useful. Let's see how we can register a service in Consul and then retrieve that data.

Running a distributed service with Consul in Docker

To register our service we're going to create a phony distributed application written in the [uWSGI](#) framework. We're going to build our application in two pieces.

- A web application, `distributed_app`. It runs web workers and registers them as services with Consul when it starts.
- A client for our application, `distributed_client`. The client reads data about `distributed_app` from Consul and reports the current application state and configuration.

We're going run the `distributed_app` on two of our Consul nodes: `larry` and `curly`. We'll run the `distributed_client` client on the `moe` node.

Building our distributed application

We're going to start with creating a Dockerfile to build `distributed_app`. Let's create a directory to hold our image first.

Listing 7.50: Creating a `distributed_app` Dockerfile directory

```
$ mkdir distributed_app
$ cd distributed_app
$ touch Dockerfile
```

Now let's look at the Dockerfile for our `distributed_app` application.

Listing 7.51: The distributed_app Dockerfile

```
FROM ubuntu:14.04
MAINTAINER James Turnbull "james@example.com"
ENV REFRESHED_AT 2014-06-01

RUN apt-get -qqy update
RUN apt-get -qqy install ruby-dev git libcurl4-openssl-dev curl ←
    build-essential python
RUN gem install --no-ri --no-rdoc uwsgi sinatra
RUN uwsgi --build-plugin https://github.com/unbit/uwsgi-consul

RUN mkdir -p /opt/distributed_app
WORKDIR /opt/distributed_app

ADD uwsgi-consul.ini /opt/distributed_app/
ADD config.ru /opt/distributed_app/

ENTRYPOINT [ "uwsgi", "--ini", "uwsgi-consul.ini", "--ini", "←
    uwsgi-consul.ini:server1", "--ini", "uwsgi-consul.ini:server2" ]
CMD []
```

Our Dockerfile installs some required packages including the uWSGI and Sinatra frameworks as well as [a plugin to allow uWSGI to write to Consul](#). We create a directory called /opt/distributed_app/ and make it our working directory. We then add two files, uwsgi-consul.ini and config.ru to that directory.

The uwsgi-consul.ini file configured uWSGI itself. Let's look at it now.

Listing 7.52: The uWSGI configuration

```
[uwsgi]
plugins = consul
socket = 127.0.0.1:9999
master = true
enable-threads = true

[server1]
consul-register = url=http://%h.node.consul:8500,name=<-
    distributed_app,id=server1,port=2001
mule = config.ru

[server2]
consul-register = url=http://%h.node.consul:8500,name=<-
    distributed_app,id=server2,port=2002
mule = config.ru
```

The uwsgi-consul.ini file uses uWSGI's Mule construct to run two identical applications that do "Hello World" in the Sinatra framework. Let's look at those in the config.ru file.

Listing 7.53: The distributed_app config.ru file

```
require 'rubygems'
require 'sinatra'

get '/' do
  "Hello World!"
end

run Sinatra::Application
```

Each application is defined in a block, labelled server1 and server2 respectively. Also inside these blocks is a call to the uWSGI Consul plugin. This call connects to our Consul instance and registers a service called distributed_app with an ID

of server1 or server2. Each service is assigned a different port, 2001 and 2002 respectively.

When the framework runs this will create our two web application workers and register a service for each on Consul. The application will use the local Consul node to create the service with the %h configuration shortcut populating the Consul URL with the right hostname.

Listing 7.54: The Consul plugin URL

```
url=http://%h.node.consul:8500...
```

Lastly, we've configured an ENTRYPPOINT instruction to automatically run our web application workers.

Let's build our image now.

Listing 7.55: Building our distributed_app image

```
$ sudo docker build -t="jamtur01/distributed_app" .
```

NOTE You can get our distributed_app Dockerfile and configuration and application files [here](#) or on [GitHub](#).

Building our distributed client

We're now going to create a Dockerfile to build our distributed_client image. Let's create a directory to hold our image first.

Listing 7.56: Creating a distributed_client Dockerfile directory

```
$ mkdir distributed_client
$ cd distributed_client
$ touch Dockerfile
```

Now let's look at the Dockerfile for the distributed_client application.

Listing 7.57: The distributed_client Dockerfile

```
FROM ubuntu:14.04
MAINTAINER James Turnbull "james@example.com"
ENV REFRESHED_AT 2014-06-01

RUN apt-get -qqy update
RUN apt-get -qqy install ruby ruby-dev build-essential
RUN gem install --no-ri --no-rdoc json

RUN mkdir -p /opt/distributed_client
ADD client.rb /opt/distributed_client/

WORKDIR /opt/distributed_client

ENTRYPOINT [ "ruby", "/opt/distributed_client/client.rb" ]
CMD []
```

The Dockerfile installs Ruby and some prerequisite packages and gems. It creates the /opt/distributed_client directory and makes it the working directory. It copies our client application code, contained in the client.rb file, into the /opt/distributed_client directory.

Let's take a quick look at our application code now.

Listing 7.58: The distributed_client application

```

require "rubygems"
require "json"
require "net/http"
require "uri"
require "resolv"

uri = URI.parse("http://consul.service.consul:8500/v1/catalog/←
  service/distributed_app")

http = Net::HTTP.new(uri.host, uri.port)
request = Net::HTTP::Get.new(uri.request_uri)
response = http.request(request)

while true
  if response.body == "{}"
    puts "There are no distributed applications registered in ←
      Consul"
    sleep(1)
  elsif
    result = JSON.parse(response.body)
    result.each do |service|
      puts "Application #{service['ServiceName']} with element #{←
        service["ServiceID"]} on port #{service["ServicePort"]} ←
        found on node #{service["Node"]} (#{service["Address"]})."
      dns = Resolv::DNS.new.getresources("distributed_app.service←
        .consul", Resolv::DNS::Resource::IN::A)
      puts "We can also resolve DNS - #{service['ServiceName']} ←
        resolves to #{dns.collect { |d| d.address }.join(" and ")}."
      sleep(1)
    end
  end
end

```

Our client checks the Consul HTTP API and the Consul DNS for the presence of a service called `distributed_app`. It queries the host `consul.service.consul` which is the DNS CNAME entry we saw earlier that contains all the A records of our Consul cluster nodes. This provides us with a simple DNS round robin for our queries.

If no service is present it puts a message to that effect on the console. If it detects a `distributed_app` service then it:

- Parses out the JSON output from the API call and returns some useful information to the console.
- Performs a DNS lookup for any A records for that service and returns them to the console.

This will allow us to see the results of launching our `distributed_app` containers on our Consul cluster.

Lastly our Dockerfile specifies an `ENTRYPOINT` instruction that runs the `client.rb` application when the container is started.

Let's build our image now.

Listing 7.59: Building our `distributed_client` image

```
$ sudo docker build -t="jamtur01/distributed_client" .
```

NOTE You can get our `distributed_client` Dockerfile and configuration and application files [here](#) or on [GitHub](#).

Starting our distributed application

Now we've built the required images we can launch our `distributed_app` application container on `larry` and `curly`. We've assumed that you have Consul running as we've configured it earlier in the chapter. Let's start by running one application instance on `larry`.

Listing 7.60: Starting distributed_app on larry

```
larry$ sudo docker run -h $HOSTNAME -d --name larry_distributed \
jamtur01/distributed_app
```

Here we've launched the `jamtur01/distributed_app` image and specified the `-h` flag to set the hostname. This is important because we're using this hostname to tell uWSGI what Consul node to register the service on. We've called our container `larry_distributed` and run it daemonized.

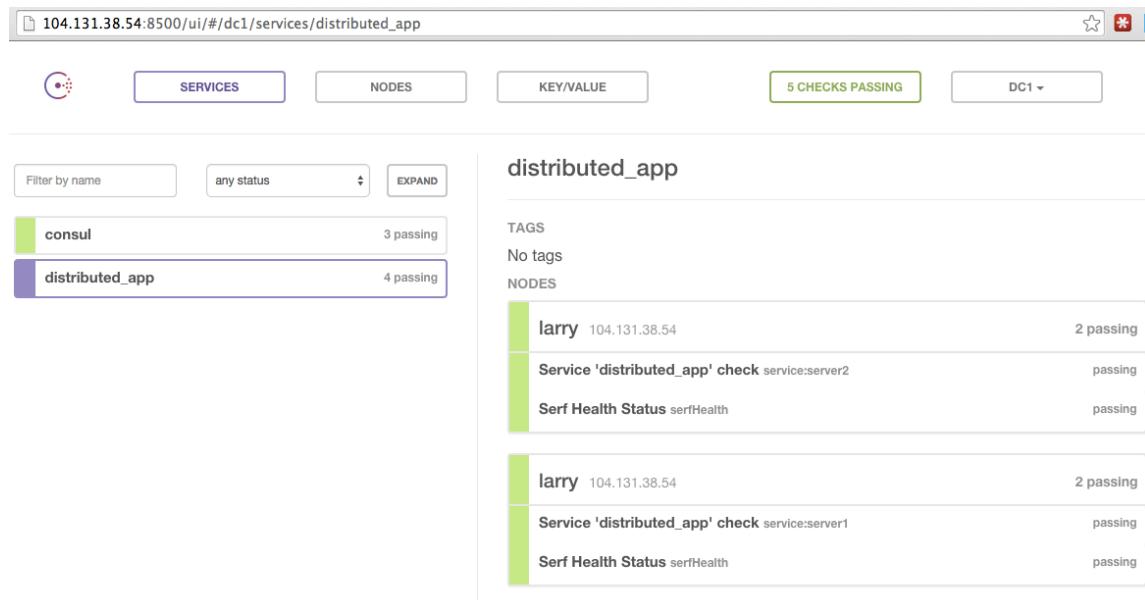
If we check the log output from the container we should see uWSGI starting our web application workers and registering the service on Consul.

Listing 7.61: The distributed_app log output

```
larry$ sudo docker logs larry_distributed
[uWSGI] getting INI configuration from uwsgi-consul.ini
*** Starting uWSGI 2.0.6 (64bit) on [Tue Sep 2 03:53:46 2014] ←
***                                         ←
. . .
[consul] built service JSON: {"Name":"distributed_app","ID":"←
    server1","Check":{"TTL":"30s"},"Port":2001}
[consul] built service JSON: {"Name":"distributed_app","ID":"←
    server2","Check":{"TTL":"30s"},"Port":2002}
[consul] thread for register_url=http://larry.node.consul:8500/v1←
    /agent/service/register check_url=http://larry.node.consul:8500/←
    v1/agent/check/pass/service:server1 name=distributed_app id=←
    server1 started
. . .
Tue Sep 2 03:53:47 2014 - [consul] workers ready, let's register←
    the service to the agent
[consul] service distributed_app registered successfully
```

We can see a subset of the logs here. We see that uWSGI has started. The Consul plugin has constructed a [service entry](#) for each `distributed_app` worker and then registered them with Consul. If we now look at the Consul web interface we should be able to see our new services.

Chapter 7: Docker Orchestration and Service Discovery



The screenshot shows the Consul web interface at the URL `104.131.38.54:8500/ui/#/dc1/services/distributed_app`. The interface has a navigation bar with tabs for SERVICES, NODES, KEY/VALUE, and a status indicator for 5 CHECKS PASSING. A dropdown menu for DC1 is also present. Below the navigation, there are filters for 'Filter by name' and 'any status', and an 'EXPAND' button. The main content area is titled 'distributed_app'. It shows two service entries: 'consul' with 3 passing checks and 'distributed_app' with 4 passing checks. The 'distributed_app' entry is selected. Under 'NODES', there are two entries for the node 'larry' (IP 104.131.38.54). The first entry for 'larry' shows two checks: 'Service 'distributed_app' check service:server2' and 'Serf Health Status serfHealth', both marked as 'passing'. The second entry for 'larry' shows two checks: 'Service 'distributed_app' check service:server1' and 'Serf Health Status serfHealth', also marked as 'passing'.

Figure 7.4: The distributed_app service in the Consul web interface.

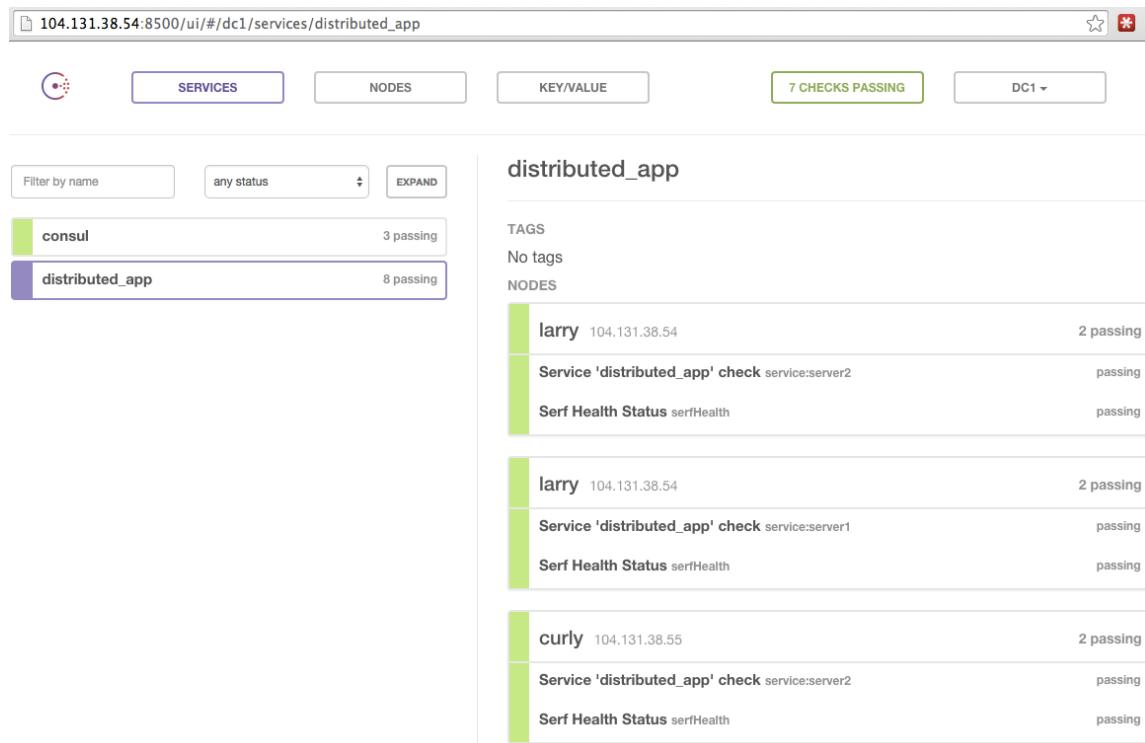
Let's start some more web application workers on curly now.

Listing 7.62: Starting distributed_app on curly

```
curly$ sudo docker run -h $HOSTNAME -d --name curly_distributed \
jamtur01/distributed_app
```

If we check the logs and the Consul web interface we should now see more services registered.

Chapter 7: Docker Orchestration and Service Discovery



The screenshot shows the Consul UI at 104.131.38.54:8500/ui/#/dc1/services/distributed_app. The top navigation bar includes tabs for SERVICES, NODES, KEY/VALUE, and a green button for 7 CHECKS PASSING. A dropdown for DC1 is also present. The main content area is titled "distributed_app". It shows a table of nodes and their service status. The table has three rows, each representing a node: "larry", "curly", and "moe". Each row contains three columns: the node name, its IP address (104.131.38.54), and the number of passing checks (2). Below each node name, there are three service entries: "Service 'distributed_app' check service:server2" and "Serf Health Status serfHealth", both marked as "passing".

larry	104.131.38.54	2 passing
Service 'distributed_app' check	service:server2	passing
Serf Health Status	serfHealth	passing
curly	104.131.38.55	2 passing
Service 'distributed_app' check	service:server2	passing
Serf Health Status	serfHealth	passing
moe	104.131.38.54	2 passing
Service 'distributed_app' check	service:server2	passing
Serf Health Status	serfHealth	passing

Figure 7.5: More distributed_app services in the Consul web interface.

Starting our distributed application client

Now we've got web application workers running on `larry` and `curly` let's start our client on `moe` and see if we can query data from Consul.

Listing 7.63: Starting distributed_client on moe

```
moe$ sudo docker run -h $HOSTNAME -d --name moe_distributed \
jamtur01/distributed_client
```

This time we've run the `jamtur01/distributed_client` image on `moe` and created a container called `moe_distributed`. Let's look at the log output to see if our distributed client has found anything about our web application workers.

Listing 7.64: The distributed_client logs on moe

```
moe$ sudo docker logs moe_distributed
Application distributed_app with element server2 on port 2002 ←
    found on node larry (104.131.38.54).
We can also resolve DNS - distributed_app resolves to ←
    104.131.38.55 and 104.131.38.54.
Application distributed_app with element server1 on port 2001 ←
    found on node larry (104.131.38.54).
We can also resolve DNS - distributed_app resolves to ←
    104.131.38.54 and 104.131.38.55.
Application distributed_app with element server2 on port 2002 ←
    found on node curly (104.131.38.55).
We can also resolve DNS - distributed_app resolves to ←
    104.131.38.55 and 104.131.38.54.
Application distributed_app with element server1 on port 2001 ←
    found on node curly (104.131.38.55).
```

We can see that our `distributed_client` application has queried the HTTP API and found service entries for `distributed_app` and its `server1` and `server2` workers on both `larry` and `curly`. It has also done a DNS lookup to discover the IP address of the nodes running that service, `104.131.38.54` and `104.131.38.55`.

If this was a real distributed application our client and our workers could take advantage of this information to configure, connect, route between elements of the distributed application. This provides a simple, easy and resilient way to build distributed applications running inside separate Docker containers and hosts.

Orchestration alternatives and components

As we mentioned earlier, Fig and Consul aren't the only games in town when it comes to Docker orchestration tools. There's a fast growing ecosystem of them. This is a non-comprehensive list of some of the tools available in that ecosystem. Not all of them have matching functionality and broadly fall into two categories:

- Scheduling and cluster management.
- Service discovery.

NOTE All of the tools listed are open source under various licenses.

Fleet and etcd

Fleet and etcd are released by the [CoreOS](#) team. [Fleet](#) is a cluster management tool and [etcd](#) is highly-available key value store for shared configuration and service discovery. Fleet combines systemd and etcd to provide cluster management and scheduling for Docker containers. Think of it as an extension of systemd that operates at the cluster level instead of the machine level.

It's a fairly new project and it is currently only available as a preview release.

Kubernetes

[Kubernetes](#) is a container cluster management tool open sourced by Google. It allows you to deploy and scale applications using Docker across multiple hosts. Kubernetes is primarily targeted at applications comprised of multiple containers, such as elastic, distributed micro-services.

It's a relatively new and lacks comprehensive documentation but is rapidly growing a community around it.

Apache Mesos

The [Apache Mesos](#) project is a highly-available cluster management tool. Since Mesos 0.20.0 it has [built-in Docker integration](#) to allow you to use containers with Mesos. Mesos is popular with a number of startups, notably Twitter and AirBnB.

Helios

The [Helios](#) project has been released by the team at Spotify and is a Docker orchestration platform for deploying and managing containers across an entire fleet. It creates a "job" abstraction that you can deploy to one or more Helios hosts running Docker.

Centurion

[Centurion](#) is focussed on being a Docker-based deployment tool open sourced by the New Relic team. Centurion takes containers from a Docker registry and runs them on a fleet of hosts with the correct environment variables, host volume mappings, and port mappings. It is designed to help you do continuous deployment with Docker.

Libswarm

Docker Inc's own orchestration efforts are focussed around [Libswarm](#). Libswarm is more of a library or toolkit and is designed to help you compose network services. It provides standard interfaces for connecting services across distributed systems. It's got some initial, Docker-focussed services, but is building a library of services to allow you to integrate a variety of other services, including many of those listed in this section.

Summary

In this chapter we've introduced you to orchestration with Fig. We've shown you how to add a Fig configuration file to create simple application stacks. We've shown you how to run Fig and build those stacks and how to perform basic management tasks on them.

Chapter 7: Docker Orchestration and Service Discovery

We've also shown you a service discovery tool, Consul. We've installed Consul onto Docker and created a cluster of Consul nodes. We've also demonstrated how a simple distributed application might work on Docker.

Finally, we've seen some of the other orchestration tools available to us in the Docker ecosystem.

In the next chapter we'll look at the Docker API, how we can use it, and how we can secure connections to our Docker daemon via TLS.

Chapter 8

Using the Docker API

In Chapter 6, we saw some excellent examples of how to run services and build applications and workflow around Docker. One of those examples, the TProv application, focused on using the `docker` binary on the command line and capturing the resulting output. This is not a very elegant approach to integrating with Docker; especially when Docker comes with a powerful API you can use to integrate directly.

In this chapter, we're going to introduce you to the Docker API and see how we can make use of it. We're going to take you through binding the Docker daemon on a network port. We'll then take you through the API at a high level and hit on the key aspects of it. We'll also look at the TProv application we saw in Chapter 6 and rewrite some portions of it to use the API instead of the `docker` binary. Lastly, we'll look at authenticating the API via TLS.

The Docker APIs

There are [three specific APIs](#) in the Docker ecosystem.

- The Registry API - provides integration with the Docker registry, which stores our images.
- The Docker Hub API - provides integration with the [Docker Hub](#).

- The Docker Remote API - provides integration with the Docker daemon.

All three APIs are broadly [RESTful](#). In this chapter, we'll focus on the Remote API because it is key to any programmatic integration and interaction with Docker.

First steps with the Remote API

Let's explore the Docker Remote API and see its capabilities. Firstly, we need to remember the Remote API is provided by the Docker daemon. By default, the Docker daemons binds to a socket, `unix:///var/run/docker.sock`, on the host on which it is running. The daemon runs with root privileges so as to have the access needed to manage the appropriate resources. As we also discovered in Chapter 2, if a group named `docker` exists on your system, Docker will apply ownership of the socket to that group. Hence, any user that belongs to the `docker` group can run Docker without needing root privileges.

WARNING Remember that although the `docker` group makes life easier, it is still a security exposure. The `docker` group is root-equivalent and should be limited to those users and applications that absolutely need it.

This works fine if we're querying the API from the same host running Docker, but if we want remote access to the API, we need to bind the Docker daemon to a network interface. This is done by passing or adjusting the `-H` flag to the Docker daemon.

On most distributions, we can do this by editing the daemon's startup configuration files. For Ubuntu or Debian, this would be the `/etc/default/docker` file; for those releases with Upstart, it would be the `/etc/init/docker.conf` file. For Red Hat, Fedora, and related distributions, it would be the `/etc/sysconfig/docker` file; for those releases with Systemd, it is the `/usr/lib/systemd/system/docker.service` file.

Let's bind the Docker daemon to a network interface on a Red Hat derivative running Systemd. We'll edit the `/usr/lib/systemd/system/docker.service` file and change:

Listing 8.1: Default systemd daemon start options

```
ExecStart=/usr/bin/docker -d --selinux-enabled
```

To:

Listing 8.2: Network binding systemd daemon start options

```
ExecStart=/usr/bin/docker -d --selinux-enabled -H tcp<-->
://0.0.0.0:2375
```

This will bind the Docker daemon to all interfaces on the host using port 2375. We then need to reload and restart the daemon using the `systemctl` command.

Listing 8.3: Reloading and restarting the Docker daemon

```
$ sudo systemctl --system daemon-reload
```

TIP You'll also need to ensure that any firewall on the Docker host or between you and the host allows TCP communication to the IP address on port 2375.

We can now test that this is working using the `docker` client binary, passing the `-H` flag to specify our Docker host. Let's connect to our Docker daemon from a remote host.

Listing 8.4: Connecting to a remote Docker daemon

```
$ sudo docker -H docker.example.com:2375 info
Containers: 0
Images: 0
Driver: devicemapper
  Pool Name: docker-252:0-133394-pool
  Data file: /var/lib/docker/devicemapper/devicemapper/data
  Metadata file: /var/lib/docker/devicemapper/devicemapper/←
    metadata
  . . .
```

This assumes the Docker host is called `docker.example.com`; we've used the `-H` flag to specify this host. Docker will also honor the `DOCKER_HOST` environment variable rather than requiring the continued use of the `-H` flag.

Listing 8.5: Revisiting the DOCKER_HOST environment variable

```
$ export DOCKER_HOST="tcp://docker.example.com:2375"
```

WARNING Remember this connection is unauthenticated and open to the world! Later in this chapter, we'll see how we can add authentication to this connection.

Testing the Docker Remote API

Now that we've established and confirmed connectivity to the Docker daemon via the `docker` binary, let's try to connect directly to the API. To do so, we're going to use the `curl` command. We're going to connect to the `info` API endpoint, which provides roughly the same information as the `docker info` command.

Listing 8.6: Using the info API endpoint

```
$ curl http://docker.example.com:2375/info
{
  "Containers": 0,
  "Debug": 0,
  "Driver": "devicemapper",
  ...
  "IPv4Forwarding": 1,
  "Images": 0,
  "IndexServerAddress": "https://index.docker.io/v1/",
  "InitPath": "/usr/libexec/docker/dockerinit",
  "InitSha1": "dafd83a92eb0fc7c657e8eae06bf493262371a7a",
  "KernelVersion": "3.9.8-300.fc19.x86_64",
  "LXCVersion": "0.9.0",
  "MemoryLimit": 1,
  "NEventsListener": 0,
  "NFd": 10,
  "NGoroutines": 14,
  "SwapLimit": 0
}
```

We've connected to the Docker API on `http://docker.example.com:2375` using the `curl` command, and we've specified the path to the Docker API: `docker` on `example.com` on port 2375 with endpoint `info`.

We can see that the API has returned a JSON hash, of which we've included a sample, containing the system information for the Docker daemon. This demonstrates that the Docker API is working and we're getting some data back.

Managing images with the API

Let's start with some API basics: working with Docker images. We're going to start by getting a list of all the images on our Docker daemon.

Listing 8.7: Getting a list of images via API

```
$ curl http://docker.example.com:2375/images/json | python -mjson.tool
[
{
  "Created": 1404088258,
  "Id": "2←
e9e5fdd46221b6d83207aa62b3960a0472b40a89877ba71913998ad9743e065←
",
  "ParentId": "7←
cd0eb092704d1be04173138be5caee3a3e4bea5838dcde9ce0504cdc1f24cbb←
",
  "RepoTags": [
    "docker:master"
  ],
  "Size": 186470239,
  "VirtualSize": 1592910576
},
. . .
{
  "Created": 1403739688,
  "Id": "15←
d0178048e904fee25354db77091b935423a829f171f3e3cf27f04ffcf7cf56←
",
  "ParentId": "74830←
af969b02bb2cec5fe04bb2e168a4f8d3db3ba504e89cacba99a262baf48"←
",
  "RepoTags": [
    "jamtur01/jekyll:latest"
  ],
  "Size": 0,
  "VirtualSize": 607622922
}
. . .
]
```

NOTE We've passed the output through Python's JSON tool to prettify it.

We've used the `/images/json` endpoint, which will return a list of all images on the Docker daemon. It'll give us much the same information as the `docker images` command. We can also query specific images via ID, much like `docker inspect` on an image ID.

Listing 8.8: Getting a specific image

```
curl http://docker.example.com:2375/images/15←
d0178048e904fee25354db77091b935423a829f171f3e3cf27f04ffcf7cf56/←
  json | python -mjson.tool
{
  "Architecture": "amd64",
  "Author": "James Turnbull <james@example.com>",
  "Comment": "",
  "Config": {
    "AttachStderr": false,
    "AttachStdin": false,
    "AttachStdout": false,
    "Cmd": [
      "--config=/etc/jekyll.conf"
    ],
    . . .
  }
}
```

Here we can see a subset of the output of inspecting our `jamturn01/jekyll` image. And finally, like the command line, we can search for images on the Docker Hub.

Listing 8.9: Searching for images with the API

```
$ curl "http://docker.example.com:2375/images/search?term=<jamtur01" | python -mjson.tool
[
  {
    "description": "",
    "is_official": false,
    "is_trusted": true,
    "name": "jamtur01/docker-presentation",
    "star_count": 2
  },
  {
    "description": "",
    "is_official": false,
    "is_trusted": false,
    "name": "jamtur01/dockerjenkins",
    "star_count": 1
  },
  ...
]
```

Here we've searched for all images containing the term `jamtur01` and displayed a subset of the output returned. This is just a sampling of the actions we can take with the Docker API. We can also build, update, and remove images.

Managing containers with the API

The Docker Remote API also exposes all of the container operations available to us on the command line. We can list running containers using the `/containers` endpoint much as we would with the `docker ps` command.

Listing 8.10: Listing running containers

```
$ curl -s "http://docker.example.com:2375/containers/json" | ←
  python -mjson.tool
[  
  {  
    "Command": "/bin/bash",  
    "Created": 1404319520,  
    "Id": "←  
      cf925ad4f3b9fea231aee386ef122f8f99375a90d47fc7cbe43fac1d962dc51b←  
      ",  
    "Image": "ubuntu:14.04",  
    "Names": [  
      "/desperate_euclid"  
    ],  
    "Ports": [],  
    "Status": "Up 3 seconds"  
  }  
]
```

Our query will show all running containers on the Docker host, in our case, a single container. To see running and stopped containers, we can add the `all` flag to the endpoint and set it to 1.

Listing 8.11: Listing all containers via the API

```
http://docker.example.com:2375/containers/json?all=1
```

We can also use the API to create containers by using a POST request to the `/containers/create` endpoint. Here is the simplest possible container creation API call.

Listing 8.12: Creating a container via the API

```
$ curl -X POST -H "Content-Type: application/json" \
http://docker.example.com:2375/containers/create \
-d '{
  "Image": "jamtur01/jekyll"
}'
{"Id": "591←
ba02d8d149e5ae5ec2ea30ffe85ed47558b9a40b7405e3b71553d9e59bed3", "←
Warnings": null}
```

We call the `/containers/create` endpoint and POST a JSON hash containing an image name to the endpoint. The API returns the ID of the container we've just created and potentially any warnings. This will create a container.

We can further configure our container creation by adding key/value pairs to our JSON hash.

Listing 8.13: Configuring container launch via the API

```
$ curl -X POST -H "Content-Type: application/json" \
"http://docker.example.com:2375/containers/create?name=jekyll" \
-d '{
  "Image": "jamtur01/jekyll",
  "Hostname": "jekyll"
}'
{"Id": "591←
ba02d8d149e5ae5ec2ea30ffe85ed47558b9a40b7405e3b71553d9e59bed3", "←
Warnings": null}
```

Here we've specified the `Hostname` key with a value of `jekyll` to set the hostname of the resulting container.

To start the container we use the `/containers/start` endpoint.

Listing 8.14: Starting a container via the API

```
$ curl -X POST -H "Content-Type: application/json" \
http://docker.example.com:2375/containers/591<-
    ba02d8d149e5ae5ec2ea30ffe85ed47558b9a40b7405e3b71553d9e59bed3/<-
        start \
-d '{
    "PublishAllPorts":true
}'
```

In combination, this provides the equivalent of running:

Listing 8.15: API equivalent for docker run command

```
$ sudo docker run jamtur01/jekyll
```

We can also inspect the resulting container via the `/containers/` endpoint.

Listing 8.16: Listing all containers via the API

```
$ curl http://docker.example.com:2375/containers/591←
ba02d8d149e5ae5ec2ea30ffe85ed47558b9a40b7405e3b71553d9e59bed3/←
json | python -mjson.tool
{
    "Args": [
        "build",
        "--destination=/var/www/html"
    ],
    ...
    "Hostname": "591ba02d8d14",
    "Image": "jamtur01/jekyll",
    ...
    "Id": "591←
ba02d8d149e5ae5ec2ea30ffe85ed47558b9a40b7405e3b71553d9e59bed3←
",
    "Image": "29←
d4355e575cff59d7b7ad837055f231970296846ab58a037dd84be520d1cc31←
",
    ...
    "Name": "/hopeful_davinci",
    ...
}
```

Here we can see we've queried our container using the container ID and shown a sampling of the data available to us.

Improving TProv

Now let's look at the methods inside the TProv application that we used in Chapter 6. We're going to look specifically at the methods which create and delete Docker containers.

Listing 8.17: The legacy TProv container launch methods

```

def get_war(name, url)
  cid = `docker run --name #{name} jamtur01/fetcher #{url} 2>&1`.  

    chop
  puts cid
  [$?.exitstatus == 0, cid]
end

def create_instance(name)
  cid = `docker run -P --volumes-from #{name} -d -t jamtur01/←  

    tomcat7 2>&1`.chop
  [$?.exitstatus == 0, cid]
end

def delete_instance(cid)
  kill = `docker kill #{cid} 2>&1`  

  [$?.exitstatus == 0, kill]
end

```

NOTE You can see the previous TProv code at [here](#) or on [Git Hub](#).

Pretty crude, eh? We're directly calling out to the `docker` binary and capturing its output. There are lots of reasons that that will be problematic, not least of which is that you can only run the TProv application somewhere with the Docker client installed.

We can improve on this interface by using the Docker API via one of its client libraries, in this case the [Ruby Docker-API client library](#).

TIP You can find a full list of the available client libraries [here](#). There are client libraries for Ruby, Python, Node.JS, Go, Erlang, Java, and others.

Let's start by looking at how we establish our connection to the Docker API.

Listing 8.18: The Docker Ruby client

```
require 'docker'

. . .

module TProv
  class Application < Sinatra::Base

  . . .

  Docker.url = ENV['DOCKER_URL'] || 'http://localhost:2375'
  Docker.options = {
    :ssl_verify_peer => false
  }
```

We've added a require for the docker-api gem. We'd need to install this gem first to get things to work or add it to the TProv application's gem specification.

We can then use the Docker.url method to specify the location of the Docker host we wish to use. In our code, we specify this via an environment variable, DOCKER_URL, or use a default of http://localhost:2375.

We've also used the Docker.options to specify options we want to pass to the Docker daemon connection.

We can test this idea using the IRB shell locally. Let's try that now. You'll need to have Ruby installed on the host on which you are testing. Let's assume we're using a Fedora host.

Listing 8.19: Installing the Docker Ruby client API prerequisites

```
$ sudo yum -y install ruby ruby-irb
.
.
$ sudo gem install docker-api json
.
```

Now we can use `irb` to test our Docker API connection.

Listing 8.20: Testing our Docker API connection via irb

```
$ irb
irb(main):001:0> require 'docker'; require 'pp'
=> true
irb(main):002:0> Docker.url = 'http://docker.example.com:2375'
=> "http://docker.example.com:2375"
irb(main):003:0> Docker.options = { :ssl_verify_peer => false }
=> {:ssl_verify_peer=>false}
irb(main):004:0> pp Docker.info
{"Containers"=>9,
 "Debug"=>0,
 "Driver"=>"aufs",
 "DriverStatus"=>[["Root Dir", "/var/lib/docker/aufs"], ["Dirs", <-- "882"]],
 "ExecutionDriver"=>"native-0.2",
 .
.
irb(main):005:0> pp Docker.version
{"ApiVersion"=>"1.12",
 "Arch"=>"amd64",
 "GitCommit"=>"990021a",
 "GoVersion"=>"go1.2.1",
 "KernelVersion"=>"3.8.0-29-generic",
 "Os"=>"linux",
 "Version"=>"1.0.1"}
.
```

We can see that we've launched `irb` and loaded the `docker` gem (via a `require`) and the `pp` library to help make our output look nicer. We've then specified the `Docker.url` and `Docker.options` methods to set the target Docker host and our required options (here disabling SSL peer verification to use TLS, but not authenticate the client).

We've then run two global methods, `Docker.info` and `Docker.version`, which provide the Ruby client API equivalents of the binary commands `docker info` and `docker version`.

We can now update our `TProv` container management methods to use the API via the `docker-api` client library. Let's look at some code that does this now.

Listing 8.21: Our updated `TProv` container management methods

```
def get_war(name, url)
  container = Docker::Container.create('Cmd' => url, 'Image' => 'jamtur01/fetcher', 'name' => name)
  container.start
  container.id
end

def create_instance(name)
  container = Docker::Container.create('Image' => 'jamtur01/tomcat7')
  container.start('PublishAllPorts' => true, 'VolumesFrom' => name)
  container.id
end

def delete_instance(cid)
  container = Docker::Container.get(cid)
  container.kill
end
```

You can see we've replaced the previous binary shell with a rather cleaner implementation using the Docker API. Our `get_war` method creates and starts our

jamtur01/fetcher container using the `Docker::Container.create` and `Docker::Container.start` methods. The `create_instance` method does the same for the `jamtur01/tomcat7` container. Finally, our `delete_instance` method has been updated to retrieve a container using the container ID via the `Docker::Container.get` method. We then kill the container with the `Docker::Container.kill` method.

NOTE You can see the updated TProv code at [here](#) or on [Git Hub](#).

Authenticating the Docker Remote API

Whilst we've shown that we can connect to the Docker Remote API, that means that anyone else can also connect to the API. That poses a bit of a security issue. Thankfully, the Remote API has an authentication mechanism that has been available since the 0.9 release of Docker. The authentication uses TLS/SSL certificates to secure your connection to the API.

TIP This authentication applies to more than just the API. By turning this authentication on, you will also need to configure our Docker client to support TLS authentication. We'll see how to do that in this section, too.

There are a couple of ways we could authenticate our connection, including using a full PKI infrastructure, either creating our own Certificate Authority (CA) or using an existing CA. We're going to create our own certificate authority because it is a simple and fast way to get started.

WARNING This relies on a local CA running on your Docker host. This is not as secure as using a full-fledged Certificate Authority.

Create a Certificate Authority

We're going to quickly step through creating the required CA certificate and key, as it is a pretty standard process on most platforms. It requires the `openssl` binary as a prerequisite.

Listing 8.22: Checking for openssl

```
$ which openssl  
/usr/bin/openssl
```

Let's create a directory on our Docker host to hold our CA and related materials.

Listing 8.23: Create a CA directory

```
$ sudo mkdir /etc/docker
```

Now let's create our CA.

We first generate a private key.

Listing 8.24: Generating a private key

```
$ cd /etc/docker  
$ echo 01 | sudo tee ca.srl  
$ sudo openssl genrsa -des3 -out ca-key.pem  
Generating RSA private key, 512 bit long modulus  
.....++++++  
.....++++++  
e is 65537 (0x10001)  
Enter pass phrase for ca-key.pem:  
Verifying - Enter pass phrase for ca-key.pem:
```

We'll specify a passphrase for the CA key, make note of this phrase, and secure it. We'll need it to create and sign certificates with our new CA.

This also creates a new file called `ca-key.pem`. This is our CA key; we'll not want to share it or lose it, as it is integral to the security of our solution.

Now let's create a CA certificate.

Listing 8.25: Creating a CA certificate

```
$ sudo openssl req -new -x509 -days 365 -key ca-key.pem -out ca.pem
Enter pass phrase for ca-key.pem:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:docker.example.com
Email Address []:
```

This will create the `ca.pem` file that is the certificate for our CA. We'll need this later to verify our secure connection.

Now that we have our CA, let's use it to create a certificate and key for our Docker server.

Create a server certificate signing request and key

We can use our new CA to sign and validate a certificate signing request or CSR and key for our Docker server. Let's start with creating a key for our server.

Listing 8.26: Creating a server key

```
$ sudo openssl genrsa -des3 -out server-key.pem
Generating RSA private key, 512 bit long modulus
.....+++++
.....+++++
e is 65537 (0x10001)
Enter pass phrase for server-key.pem:
Verifying - Enter pass phrase for server-key.pem:
```

This will create our server key, `server-key.pem`. As above, we need to keep this key safe: it's what secures our Docker server.

NOTE Specify any pass phrase here. We're going to strip it out before we use the key. You'll only need it for the next couple of steps.

Next let's create our server certificate signing request (CSR).

Listing 8.27: Creating our server CSR

```
$ sudo openssl req -new -key server-key.pem -out server.csr
Enter pass phrase for server-key.pem:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:*
Email Address []:

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

This will create a file called `server.csr`. This is the request that our CA will sign to create our server certificate. The most important option here is Common Name or CN. This should either be the FQDN (fully qualified domain name) of the Docker server (i.e., what is resolved to in DNS; for example, `docker.example.com`) or `*`, which will allow us to use the server certificate on any server.

Now let's sign our CSR and generate our server certificate.

Listing 8.28: Signing our CSR

```
$ sudo openssl x509 -req -days 365 -in server.csr -CA ca.pem \
-CAkey ca-key.pem -out server-cert.pem
Signature ok
subject=/C=AU/ST=Some-State/O=Internet Widgits Pty Ltd/CN=*
Getting CA Private Key
Enter pass phrase for ca-key.pem:
```

We'll enter the passphrase of the CA's key file, and a file called `server-cert.pem` will be generated. This is our server's certificate.

Now let's strip out the passphrase from our server key. We can't enter one when the Docker daemon starts, so we need to remove it.

Listing 8.29: Removing the passphrase from the server key

```
$ sudo openssl rsa -in server-key.pem -out server-key.pem
Enter pass phrase for server-key.pem:
writing RSA key
```

Now let's add some tighter permissions to the files to better protect them.

Listing 8.30: Securing the key and certificate on the Docker server

```
$ sudo chmod 0600 /etc/docker/server-key.pem /etc/docker/server-\
cert.pem \
/etc/docker/ca-key.pem /etc/docker/ca.pem
```

Configuring the Docker daemon

Now that we've got our certificate and key, let's configure the Docker daemon to use them. As we did to expose the Docker daemon to a network socket, we're going to edit its startup configuration. As before, for Ubuntu or Debian, we'll edit the `/etc/default/docker` file; for those distributions with Upstart, it's the `/etc/init/docker.conf` file. For Red Hat, Fedora, and related distributions, we'll edit

the `/etc/sysconfig/docker` file; for those releases with Systemd, it's the `/usr/lib/systemd/system/docker.service` file.

Let's again assume a Red Hat derivative running Systemd and edit the `/usr/lib/systemd/system/docker.service` file:

Listing 8.31: Enabling Docker TLS on systemd

```
ExecStart=/usr/bin/docker -d -H tcp://0.0.0.0:2376 --tlsverify --tlscacert=/etc/docker/ca.pem --tlscert=/etc/docker/server-cert.pem --tlskey=/etc/docker/server-key.pem
```

NOTE You can see that we've used port number 2376; this is the default TLS/SSL port for Docker. You should only use 2375 for unauthenticated connections.

This code will enable TLS using the `--tlsverify` flag. We've also specified the location of our CA certificate, certificate, and key using the `--tlscacert`, `--tlscert` and `--tlskey` flags, respectively. There are a variety of other TLS options that [we could also use](#).

TIP You can use the `--tls` flag to enable TLS, but not client-side authentication.

We then need to reload and restart the daemon using the `systemctl` command.

Listing 8.32: Reloading and restarting the Docker daemon

```
$ sudo systemctl --system daemon-reload
```

Creating a client certificate and key

Our server is now TLS enabled; next, we need to create and sign a certificate and key to secure our Docker client. Let's start with a key for our client.

Listing 8.33: Creating a client key

```
$ sudo openssl genrsa -des3 -out client-key.pem
Generating RSA private key, 512 bit long modulus
.....+++++
.....+++++
e is 65537 (0x10001)
Enter pass phrase for client-key.pem:
Verifying - Enter pass phrase for client-key.pem:
```

This will create our key file `client-key.pem`. Again, we'll need to specify a temporary passphrase to use during the creation process.

Now let's create a client CSR.

Listing 8.34: Creating a client CSR

```
$ sudo openssl req -new -key client-key.pem -out client.csr
Enter pass phrase for client-key.pem:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:
Email Address []:

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

We next need to enable client authentication for our key by adding some extended SSL attributes.

Listing 8.35: Adding Client Authentication attributes

```
$ echo extendedKeyUsage = clientAuth > extfile.cnf
```

Now let's sign our CSR with our CA.

Listing 8.36: Signing our client CSR

```
$ sudo openssl x509 -req -days 365 -in client.csr -CA ca.pem \
-CAkey ca-key.pem -out client-cert.pem -extfile extfile.cnf
Signature ok
subject=/C=AU/ST=Some-State/O=Internet Widgits Pty Ltd
Getting CA Private Key
Enter pass phrase for ca-key.pem:
```

Again, we use the CA key's passphrase and generate another certificate: `client-cert.pem`.

Finally, we strip the passphrase from our `client-key.pem` file to allow us to use it with the Docker client.

Listing 8.37: Stripping out the client key pass phrase

```
$ sudo openssl rsa -in client-key.pem -out client-key.pem
Enter pass phrase for client-key.pem:
writing RSA key
```

Configuring our Docker client for authentication

Next let's configure our Docker client to use our new TLS configuration. We need to do this because the Docker daemon now expects authenticated connections for both the client and the API.

We'll need to copy our `ca.pem`, `client-cert.pem`, and `client-key.pem` files to the host on which we're intending to run the Docker client.

TIP Remember that these keys provide root-level access to the Docker daemon. You should protect them carefully.

Let's install them into the `.docker` directory. This is the default location where

Docker will look for certificates and keys. Docker will specifically look for `key.pem`, `cert.pem`, and our CA certificate: `ca.pem`.

Listing 8.38: Copying the key and certificate on the Docker client

```
$ mkdir -p ~/.docker/
$ cp ca.pem ~/.docker/ca.pem
$ cp client-key.pem ~/.docker/key.pem
$ cp client-cert.pem ~/.docker/cert.pem
$ chmod 0600 ~/.docker/key.pem ~/.docker/cert.pem
```

Now let's test the connection to the Docker daemon from the client. To do this, we're going to use the `docker info` command.

Listing 8.39: Testing our TLS-authenticated connection

```
$ sudo docker -H=docker.example.com:2376 --tlsverify info
Containers: 33
Images: 104
Storage Driver: aufs
  Root Dir: /var/lib/docker/aufs
  Dirs: 170
Execution Driver: native-0.1
Kernel Version: 3.8.0-29-generic
Username: jamtur01
Registry: [https://index.docker.io/v1/]
WARNING: No swap limit support
```

We can see that we've specified the `-H` flag to tell the client to which host it should connect. We could instead specify the host using the `DOCKER_HOST` environment variable if we didn't want to specify the `-H` flag each time. We've also specified the `--tlsverify` flag, which enables our TLS connection to the Docker daemon. We don't need to specify any certificate or key files, because Docker has automatically looked these up in our `~/.docker/` directory. If we did need to specify these files, we could with the `--tlscacert`, `--tlscert`, and `--tlskey` flags.

So what happens if we don't specify a TLS connection? Let's try again now without

the `--tlsverify` flag.

Listing 8.40: Testing our TLS-authenticated connection

```
$ sudo docker -H=docker.example.com:2376 info
2014/04/13 17:50:03 malformed HTTP response "\x15\x03\x01\x00\x02<-
\x02"
```

Ouch. That's not good. If you see an error like this, you know you've probably not enabled TLS on the connection, you've not specified the right TLS configuration, or you have an incorrect certificate or key.

Assuming you've got everything working, you should now have an authenticated Docker connection!

Summary

In this chapter, we've been introduced to the Docker Remote API. We've also seen how to secure the Docker Remote API via SSL/TLS certificates. We've explored the Docker API and how to use it to manage images and containers. We've also seen how to use one of the Docker API client libraries to rewrite our TProv application to directly use the Docker API.

In the next and last chapter, we'll look at how you can contribute to Docker.

Chapter 9

Getting help and extending Docker

Docker is in its infancy -- sometimes things go wrong. This chapter will talk about:

- How and where to get help.
- Contributing fixes and features to Docker.

You'll find out where to find Docker folks and the best way to ask for help. You'll also learn how to engage with Docker's developer community: there's a huge amount of development effort surrounding Docker with hundreds of committers in the open-source community. If you're excited by Docker, then it's easy to make your own contribution to the project. This chapter will also cover the basics of contributing to the Docker project, how to build a Docker development environment, and how to create a good pull request.

NOTE This chapter assumes some basic familiarity with Git, GitHub, and Go, but doesn't assume you're a fully fledged developer.

Getting help

The Docker community is large and friendly. There's a [central Help page](#) on the Docker site that provides a list of all the places to get help. Generally, however, most Docker folks congregate in three places:

NOTE Docker, Inc. also sells enterprise support for Docker. You can find the information on the [help](#) page.

The Docker user and dev mailing lists

These mailing lists are here:

- [Docker user list](#)
- [Docker developer list](#)

The Docker user list is generally for Docker usage or help questions. The Docker dev list is for more development-focused questions and issues.

Docker on IRC

The Docker community also has two strong IRC channels: `#docker` and `#docker-dev`. Both are on the [Freenode IRC network](#)

The `#docker` channel is generally for user help and general Docker issues, whereas `#docker-dev` is where contributors to Docker's source code gather.

You can find logs for `#docker` [here](#) and for `#docker-dev` [here](#).

Docker on GitHub

Docker (and most of its components and ecosystem) is hosted on [GitHub](#). The principal repository for Docker itself is [here](#).

Other repositories of note are:

- [docker-registry](#) - The stand-alone Docker registry.
- [libcontainer](#) - The Docker container format.
- [libswarm](#) - Docker's orchestration framework.

Reporting issues for Docker

Let's start with the basics around submitting issues and patches and interacting with the Docker community. When reporting [issues](#) with Docker, it's important to be an awesome open-source citizen and provide good information that can help the community resolve your issue. When you log a ticket, please remember to include the following background information:

- The output of `docker info` and `docker version`.
- The output of `uname -a`.
- Your operating system and version (e.g., Ubuntu 14.04).

Then provide a detailed explanation of your problem and the steps others can take to reproduce it.

If you're logging a feature request, carefully explain what you want and how you propose it might work. Think carefully about generic use cases: is your feature something that will make life easier for just you or for everyone?

Please take a moment to check that an issue doesn't already exist documenting your bug report or feature request. If it does, you can add a quick "+1" or "I have this problem too", or if you feel your input expands on the proposed implementation or bug fix, then add a more substantive update.

Setting up a build environment

To make it easier to contribute to Docker, we're going to show you how to build a development environment. The development environment provides all of the required dependencies and build tooling to work with Docker.

Install Docker

You must first install Docker in order to get a development environment, because the build environment is a Docker container in its own right. We use Docker to build and develop Docker. Use the steps from Chapter 2 to install Docker on your local host. You should install the most recent version of Docker available.

Install source and build tools

Next, you need to install Make and Git so that we can check out the Docker source code and run the build process. The source code is stored on GitHub, and the build process is built around a `Makefile`.

On Ubuntu, we would install the `git` package.

Listing 9.1: Installing git on Ubuntu

```
$ sudo apt-get -y install git make
```

On Red Hat and derivatives we would do the following:

Listing 9.2: Installing git on Red Hat et al

```
$ sudo yum install git make
```

Check out the source

Now let's check out the Docker source code (or, if you're working on another component, the relevant source code repository) and change into the resulting

directory.

Listing 9.3: Check out the Docker source code

```
$ git clone https://github.com/docker/docker.git  
$ cd docker
```

You can now work on the source code and fix bugs, update documentation, and write awesome features!

Contributing to the documentation

One of the great ways anyone, even if you're not a developer or skilled in Go, can contribute to Docker is to update, enhance, or develop new documentation. The Docker documentation lives on the [Docs website](#). The source documentation, the theme, and the tooling that generates this site are stored in the [Docker repo on GitHub](#).

You can find specific guidelines and a basic style guide for the documentation [here](#).

You can build the documentation locally using Docker itself.

Make any changes you want to the documentation, and then you can use the `make` command to build the documentation.

Listing 9.4: Building the Docker documentation

```
$ cd docker  
$ make docs  
.  
.  
.  
docker run --rm -it -e AWS_S3_BUCKET -p 8000:8000 "docker-docs:←  
master" mkdocs serve  
Running at: http://0.0.0.0:8000/  
Live reload enabled.  
Hold ctrl+c to quit.
```

You can then browse to a local version of the Docker documentation on port 8000.

Build the environment

If you want to contribute to more than just the documentation, you can now use `make` and Docker to build the development environment. The Docker source code ships with a `Dockerfile` that we use to install all the build and runtime dependencies necessary to build and test Docker.

Listing 9.5: Building the Docker environment

```
$ sudo make build
```

TIP This command will take some time to complete when you first execute it.

This command will create a full, running Docker development environment. It will upload the current source directory as build context for a Docker image, build the image containing Go and any other required dependencies, and then launch a container from this image.

Using this development image, we can also create a Docker binary to test any fixes or features. We do this using the `make` tool again.

Listing 9.6: Building the Docker binary

```
$ sudo make binary
```

This command will create a Docker binary in a volume at `./bundles/<version>-dev/binary/`. For example, we would create a binary like so:

Listing 9.7: The Docker dev binary

```
$ ls -l ~/docker/bundles/1.0.1-dev/binary/docker
lrwxrwxrwx 1 root root 16 Jun 29 19:53 ~/docker/bundles/1.0.1-dev-
/binary/docker -> docker-1.0.1-dev
```

You can then use this binary for live testing by running it instead of the local Docker daemon. To do so, we need to stop Docker and run this new binary instead.

Listing 9.8: Using the development daemon

```
$ sudo service docker stop
$ ~/docker/bundles/1.0.1-dev/binary/docker -d
```

This will run the development Docker daemon interactively. You can also background the daemon if you wish.

We can then test this binary by running it against this daemon.

Listing 9.9: Using the development binary

```
$ ~/docker/bundles/1.0.1-dev/binary/docker version
Client version: 1.0.1-dev
Client API version: 1.12
Go version (client): go1.2.1
Git commit (client): d37c9a4
Server version: 1.0.1-dev
Server API version: 1.12
Go version (server): go1.2.1
Git commit (server): d37c9a
```

You can see that we're running a 1.0.1-dev client, this binary, against the 1.0.1-dev daemon we just started. You can use this combination to test and ensure any changes you've made to the Docker source are working correctly.

Running the tests

It's also important to ensure that all of the Docker tests pass before contributing code back upstream. To execute all the tests, you need to run this command:

Listing 9.10: Running the Docker tests

```
$ sudo make test
```

This command will again upload the current source as build context to an image and then create a development image. A container will be launched from this

image, and the test will run inside it. Again, this may take some time for the initial build.

If the tests are successful, then the end of the output should look something like this:

Listing 9.11: Docker test output

```
.
.
.
[PASSED]: save - save a repo using stdout
[PASSED]: load - load a repo using stdout
[PASSED]: save - save a repo using -o
[PASSED]: load - load a repo using -i
[PASSED]: tag - busybox -> testfoobarbaz
[PASSED]: tag - busybox's image ID -> testfoobarbaz
[PASSED]: tag - busybox fooo/bar
[PASSED]: tag - busybox fooaa/test
[PASSED]: top - sleep process should be listed in non privileged ←
mode
[PASSED]: top - sleep process should be listed in privileged mode
[PASSED]: version - verify that it works and that the output is ←
properly formatted
PASS
PASS      github.com/docker/docker/integration-cli      178.685s
```

TIP You can use the `$TESTFLAGS` environment variable to pass in arguments to the test run.

Use Docker inside our development environment

You can also launch an interactive session inside the newly built development container:

Listing 9.12: Launching an interactive session

```
$ sudo make shell
```

To exit the container, type `exit` or `Ctrl-D`.

Submitting a pull request

If you're happy with your documentation update, bug fix, or new feature, you can submit a pull request for it on GitHub. To do so, you should fork the Docker repository and make changes on your fork in a feature branch:

- If it is a bug fix branch, name it `XXXX-something`, where `XXXX` is the number of the issue.
- If it is a feature branch, create a feature issue to announce your intentions, and name it `XXXX-something`, where `XXXX` is the number of the issue.

You should always submit unit tests for your changes. Take a look at the existing tests for inspiration. You should also always run the full test suite on your branch before submitting a pull request.

Any pull request with a feature in it should include updates to the documentation. You should use the process above to test your documentation changes before you submit your pull request. There are also specific guidelines (as we mentioned above) for documentation that you should follow.

We have some other simple rules that will help get your pull request reviewed and merged quickly:

- Always run `gofmt -s -w file.go` on each changed file before committing your changes. This produces consistent, clean code.
- Pull requests descriptions should be as clear as possible and include a reference to all the issues that they address.
- Pull requests must not contain commits from other users or branches.

- Commit messages must start with a capitalized and short summary (50 characters maximum) written in the imperative, followed by an optional, more detailed explanatory text that is separated from the summary by an empty line.
- Squash your commits into logical units of work using `git rebase -i` and `git push -f`. Include documentation changes in the same commit so that a revert would remove all traces of the feature or fix.

Lastly, the Docker project uses a Developer Certificate of Origin to verify that you wrote any code you submit or otherwise have the right to pass it on as an open-source patch. You can read about why we do this [here](#). The certificate is very easy to apply. All you need to do is add a line to each Git commit message.

Listing 9.13: The Docker DCO

```
Docker-DCO-1.1-Signed-off-by: Joe Smith <joe.smith@email.com> (←  
github: github_handle)
```

NOTE You must use your real name. We do not allow pseudonyms or anonymous contributions for legal reasons.

There are several small exceptions to the signing requirement. Currently these are:

- Your patch fixes spelling or grammar errors.
- Your patch is a single-line change to documentation contained in the `docs` directory.
- Your patch fixes Markdown formatting or syntax errors in the documentation contained in the `docs` directory.

It's also pretty easy to automate the signing of your Git commits using [Git hooks](#). You can customize your `Git commit.template` by adding a `prepare-commit-msg` hook to your Docker checkout:

Listing 9.14: Git commit signing

```
$ cd docker
$ curl -o .git/hooks/prepare-commit-msg \
  https://raw.githubusercontent.com/docker/docker/master/contrib/←
  prepare-commit-msg.hook
$ chmod +x .git/hooks/prepare-commit-msg
```

NOTE The signing script expects to find your GitHub user name in `git config --get github.user`. You can set this option with the `git config --set github.user username` command.

Merge approval and maintainers

Once you've submitted your pull request, it will be reviewed, and you will potentially receive feedback. Docker uses a maintainer system much like the Linux kernel. Each component inside Docker is managed by one or more maintainers who are responsible for ensuring the quality, stability, and direction of that component. The maintainers are supplemented by Docker's benevolent dictator and chief maintainer, [Solomon Hykes](#). He's the only one who can override a maintainer, and he has sole responsibility for appointing new maintainers.

Docker maintainers use the shorthand `LGTM` (or Looks Good To Me) in comments on the code review to indicate acceptance of a pull request. A change requires `LGTM`s from an absolute majority of the maintainers of each component affected by the changes. If a change affects `docs/` and `registry/`, then it needs an absolute majority from the maintainers of `docs/` and an absolute majority of the maintainers of `registry/`.

TIP For more details, see [the maintainer process](#) documentation.

Summary

In this chapter, we've learned about how to get help with Docker and the places where useful Docker community members and developers hang out. We've also learned about the best way to log an issue with Docker, including the sort of information you need to provide to get the best response.

We've also seen how to set up a development environment to work on the Docker source or documentation and how to build and test inside this environment to ensure your fix or feature works. Finally, we've learned about how to create a properly structured and good-quality pull request with your update.

Index

- .dockerignore, 82
- /etc/hosts, 48, 143, 146, 147
- /var/lib/docker, 45, 60, 65, 152, 179
- Apache, 172, 177
- API, 266
 - /containers, 273
 - /containers/create, 274
 - /images/json, 270
 - /info, 270
 - Client libraries, 278
 - containers, 276
 - info, 269
- API documentation, 266
- AUFS, 20
- Automated Builds, 106
- Automatically restarting containers, 58
- Back up volumes, 183
- Boot2Docker, 18, 29, 32
- boot2docker
 - ip, 35
 - btrfs, 20
 - Build content, 100
 - Build context, 78, 82, 299
 - .dockerignore, 82
 - Building images, 77
- Bypassing the Dockerfile cache, 84
- CentOS, 25
- cgroups, 15, 20, 23
- Chef, 13, 19
- chroot, 6
- CI, 13, 149
- Consul, 232
 - configuration, 235
 - DNS, 232, 235, 258
 - HTTP API, 232, 236, 258
 - ports, 235
- Console
 - web interface, 236
- container
 - linking, 141
 - logging, 54
 - names, 51, 141
- container ID, 48, 50--53, 57
- containers
 - introduction, 6
- Context, 78
- Continuous Integration, 13, 118, 149
- Copy-on-write, 15
- curl, 133
- DCO, 303

Debian, 20
Debugging Dockerfiles, 84
default storage driver, 20
Developer Certificate of Origin, *see also*
 DCO
Device Mapper, 20, 22, 26
dind, 149, 191
DNS, 146, 202
Docker
 API, 266, 282
 Client libraries, 278
 List images, 270
 APT repository, 23
 Authentication, 282
 automatic container restart, 58
 binary installation, 38
 Bind UDP ports, 89
 build context, 299
 build environment, 297, 299
 Configuration Management, 13
 container ID, 48, 50--53, 57
 container names, 51
 curl installation, 37
 daemon, 38, 241
 --tls, 288
 --tlsverify, 292
 -H flag, 39
 defaults, 40
 DOCKER_HOST, 39, 269, 292
 DOCKER_OPTS, 40
 icc flag, 143
 network configuration, 39
 systemd, 40
 Unix socket, 39
 Upstart, 40
 DCO, 303
 dind, 191
 DNS, 146, 202, 241
 environment variables, 143
 docker binary, 38
 docker group, 38, 267
 Docker Hub, 66
 Docker-in-Docker, *see also* dind
 docker0, 135
 Dockerfile
 ADD, 99
 CMD, 91, 130, 131
 COPY, 100
 ENTRYPOINT, 93, 134
 ENV, 96, 145, 152
 EXPOSE, 80, 89, 90, 145
 FROM, 79
 MAINTAINER, 79
 ONBUILD, 101
 RUN, 79, 80
 USER, 97
 VOLUME, 98, 152
 WORKDIR, 95
 Documentation, 298
 Fedora
 installation, 28
 Getting help, 295
 Hub API, 266
 installation, 20, 25
 iptables, 138
 IRC, 296
 kernel versions, 19
 launching containers, 45
 license, 7
 limiting container communication,

143
linking containers, 141
links, 141
listing containers, 50
Mailing lists, 295
naming containers, 51, 141
NAT, 138
networking, 135
OS X, 18
 installation, 29
packages, 23
privileged mode, 153
Red Hat Enterprise Linux
 installation, 25
registry, 47
Registry API, 266
Remote API, 267
remote installation script, 37
required kernel version, 21
Running your own registry, 115
set container hostname, 202
setting the working directory, 95
signals, 213
SSH, 213
tags, 68
testing, 118
TLS, 283
Ubuntu
 installation, 20
Ubuntu firewall, 24
ubuntu image, 47
UFW, 24
upgrading, 42
use of sudo, 21
volumes, 125, 179, 183
deleting, 180
Windows, 18
 installation, 32
docker
 attach, 52, 159
 build, 77, 80, 81, 123, 130, 153, 161, 174, 177, 221
 --no-cache, 84
 context, 78
 commit, 75
 exec, 56, 213
 -d, 56
 -i, 57
 -t, 57
 history, 86, 123
 images, 65, 70, 86, 115, 175, 177, 272
 info, 24, 29, 45, 269, 281
 inspect, 59, 77, 89, 101, 139, 214, 272
 kill, 61, 140, 213
 signals, 213
links
 environmental variables, 145
login, 74
logs, 54, 131, 202, 207
 --tail, 55
 -f, 54, 132, 211
 -t, 55
port, 88, 89, 190
ps, 50, 53, 58, 61, 87, 127, 228, 273
 -a, 50, 61
 -l, 50
 -n, 58
 -q, 61

pull, 67, 70, 239
push, 104, 112, 116
restart, 52, 140
rm, 61, 114, 159, 180
rmi, 112, 114
run, 45, 53, 58, 62, 69, 72, 80, 84, 87, 91, 92, 116, 125, 131, 162, 178, 224, 276
--cidfile, 159
--dns, 146
--dns-search, 146
--entrypoint, 95
--hostname, 144, 202
--link, 142, 204, 207
--name, 51, 190, 204, 207
--privileged, 153
--restart, 58
--rm, 183, 203, 205
--volumes-from, 180, 190, 203, 205
-P, 89
-d, 53
-e, 96
-h, 144, 202
-p, 142
-u, 97
-v, 142, 179, 183
-w, 95
set environment variables, 96
search, 71
start, 51, 181, 230
stop, 57, 61
tag, 116
top, 55, 132
version, 281
wait, 159
Docker API, 9
docker group, 38, 267
Docker Hub, 66, 71, 104, 106, 266
Logging in, 74
Private repositories, 104
Docker Inc, 7, 69, 295
docker run
-h, 238
Docker user interfaces
DockerUI, 42
Shipyard, 42
Docker-in-Docker, *see also* dind, 149, 191
docker0, 135
DOCKER_HOST, 39, 269, 292
DOCKER_HOST, 35, 227
DOCKER_OPTS, 241
Dockerfile, 77, 102, 106, 117, 119, 124, 129, 133, 145, 150, 157, 158, 162, 173, 174, 176, 177, 185, 188, 195, 198, 200, 201, 208, 299
ADD, 120, 122, 196, 209
CMD, 177, 186
ENTRYPOINT, 174, 177, 186, 189, 198, 200, 201, 236, 255
ENV, 177
exec format, 80
EXPOSE, 120, 177, 189
RUN, 121
template, 85
VOLUME, 174, 176, 186, 199, 236
WORKDIR, 174, 186, 187
DockerUI, 42

Documentation, 298
dotCloud, 7
Drone, 170
EPEL, 27
exec format, 80
Fedora, 25
Fig, 217
 Boot2Docker, 218
 Installation, 217
 services, 217
fig
 --version, 218
 kill, 230
 logs, 229
 ps, 228
 rm, 230
 start, 230
 stop, 229
 up, 225
Getting help, 295
GitHub, 106
gofmt, 302
Golden image, 14
Image management, 14
iptables, 138
IRC, 296
jail, 6
Jekyll, 172, 175
Jenkins CI, 13, 118, 149
 automated builds, 162
 parameterized builds, 162
 post commit hook, 162
JSON, 133
kernel, 19, 21
libcontainer, 15
license, 7
Links, 141
logging, 54
 timestamps, 55
lxc, 7, 15
Mailing lists, 295
Microservices, 9
names, 51
namespaces, 20, 23
NAT, 138
Nginx, 119
nsenter, 57, 213
openssl, 283
OpenVZ, 7
Orchestration, 217
PAAS, 7, 13
Platform-as-a-Service, 13
Port mapping, 80
Private repositories, 104
Puppet, 13, 19
Red Hat Enterprise Linux, 25
Redis, 133, 135
Registry
 private, 115
Registry API, 266
Remote API, 267
REST, 267
RFC1918, 136

Service Oriented Architecture, 9

Shipyard, 42

Signals, 213

Sinatra, 132

SOA, 9

Solaris Zones, 7

SSH, 213

SSL, 282

sudo, 21

Supervisor, 93

tags, 68

Testing applications, 118

Testing workflow, 118

TLS, 266, 282

Trusted builds, 106

Ubuntu, 20

Union mount, 63

Upstart, 40

vfs, 20

Volumes, 125, 179

 backing up, 183, 212

 deleting, 180

 logging, 208

Thanks! I hope you enjoyed the book.

© Copyright 2014 - James Turnbull <james@lovedthanlost.net>

