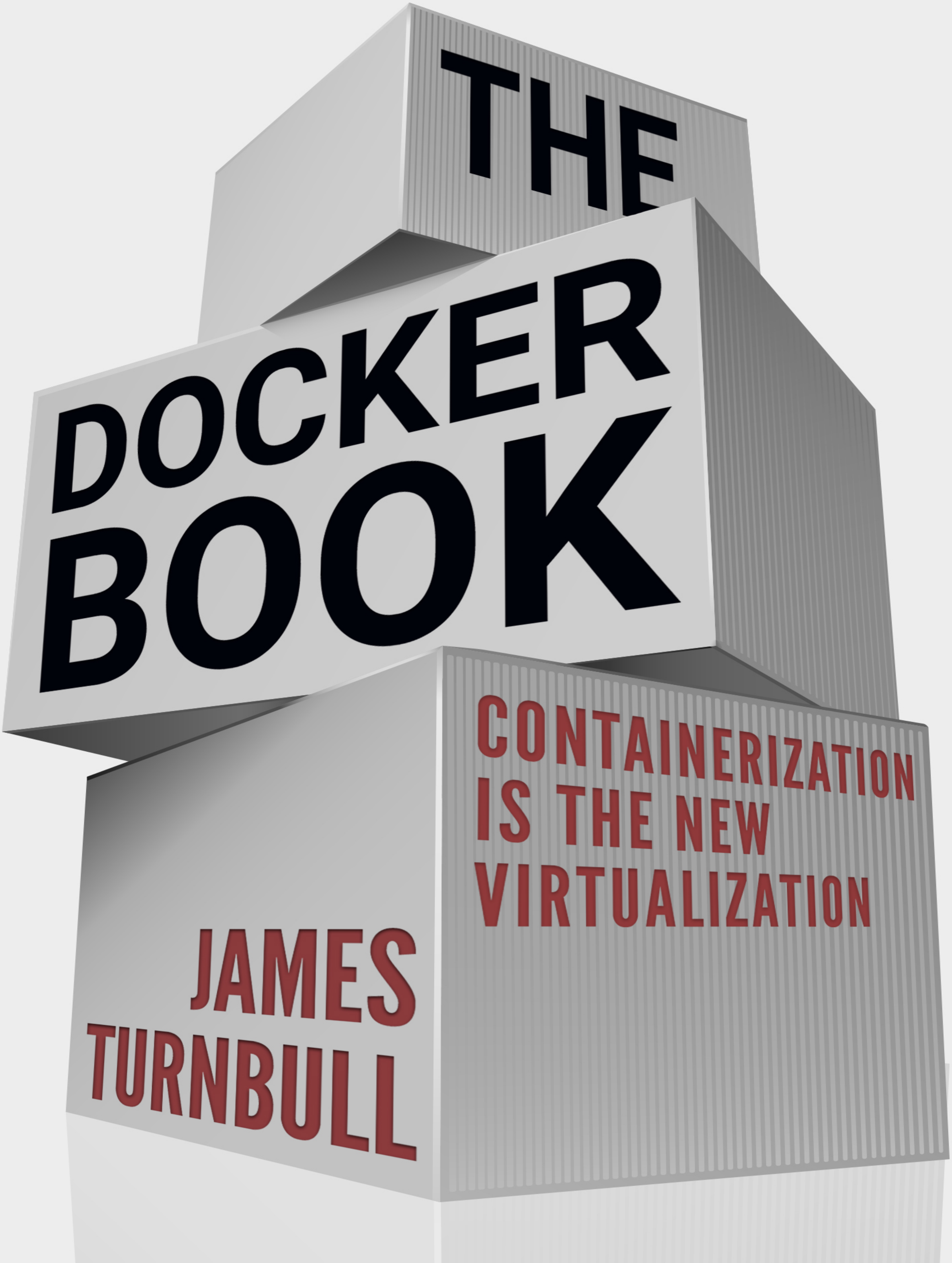
SIMPLE | SCALABLE | UP-TO-DATE



The Docker Book

James Turnbull

December 2, 2018

Version: v18.09 (6172afc)

Website: The Docker Book

ISBN 978-0-9888202-0-3

Contents

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.

A note about versions

This books focuses on Docker Community Edition version v18.08 and later. It is not generally backwards-compatible with earlier releases. Indeed, it is recommended that for production purposes you use Docker version v18.08 or later.

In March 2017 Docker re-versioned and renamed their product lines. The Docker Engine version went from Docker 1.13.1 to 17.03.0. The product was renamed to become the Docker Community Edition or Docker CE. When we refer to Docker in this book we're generally referencing the Docker Community Edition.

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.
- 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.
- Robert P. J. Day - who provided amazingly detailed errata for the book after release.

Images on pages 38, 45, 48, courtesy 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 at `http://colliernotes.com`.

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.

Pris Nasrat

Pris Nasrat works as an Engineering Manager at Etsy and is a Docker contributor. They have worked on a variety of open source tools in the systems engineering space, including boot loaders, package management, and configuration management.

Pris has worked in a variety of Systems Administration and Software Development roles, including working as an SRE at Google, a Software Engineer at Red Hat and as an Infrastructure Specialist Consultant at ThoughtWorks. Pris 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 ng-newsletter, and taking care of the fine folks at Stripe.

Author

James is an author and open-source geek. His most recent books are The Packer Book, The Terraform Book about infrastructure management tool Terraform, The Art of Monitoring about monitoring, The Docker Book about Docker, and 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.

He was formerly CTO at Kickstarter, at Docker as VP of Services and Support, Venmo as VP of Engineering, and Puppet as VP of Technical Operations. He likes food, wine, books, photography, and cats. He is not overly keen on long walks on the beach or 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.

Code and Examples

You can find all the code and examples from the book on GitHub https://github.com/turnbullpress/dockerbook-code.

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 to james+errata@lovedthanlost.net.

Version

This is version v18.09 (6172afc) 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 in 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 am an advisor at Docker Inc.

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:

NOTE Docker's underlying components are part of a project called Moby. These are brought together and constructed into the end user tool.

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 oriented architecture

Docker also encourages service-oriented 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 easy to distribute, scale, debug and introspect your applications.

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

Docker components

Let's look at the core components that compose the Docker Community Edition or Docker CE:

- The Docker client and server, also called the Docker Engine.
- 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. You'll also sometimes see the Docker daemon called the Docker Engine. Docker ships with a command line client binary, `docker`, as well as a full RESTful API to interact with the daemon: `dockerd`. 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:

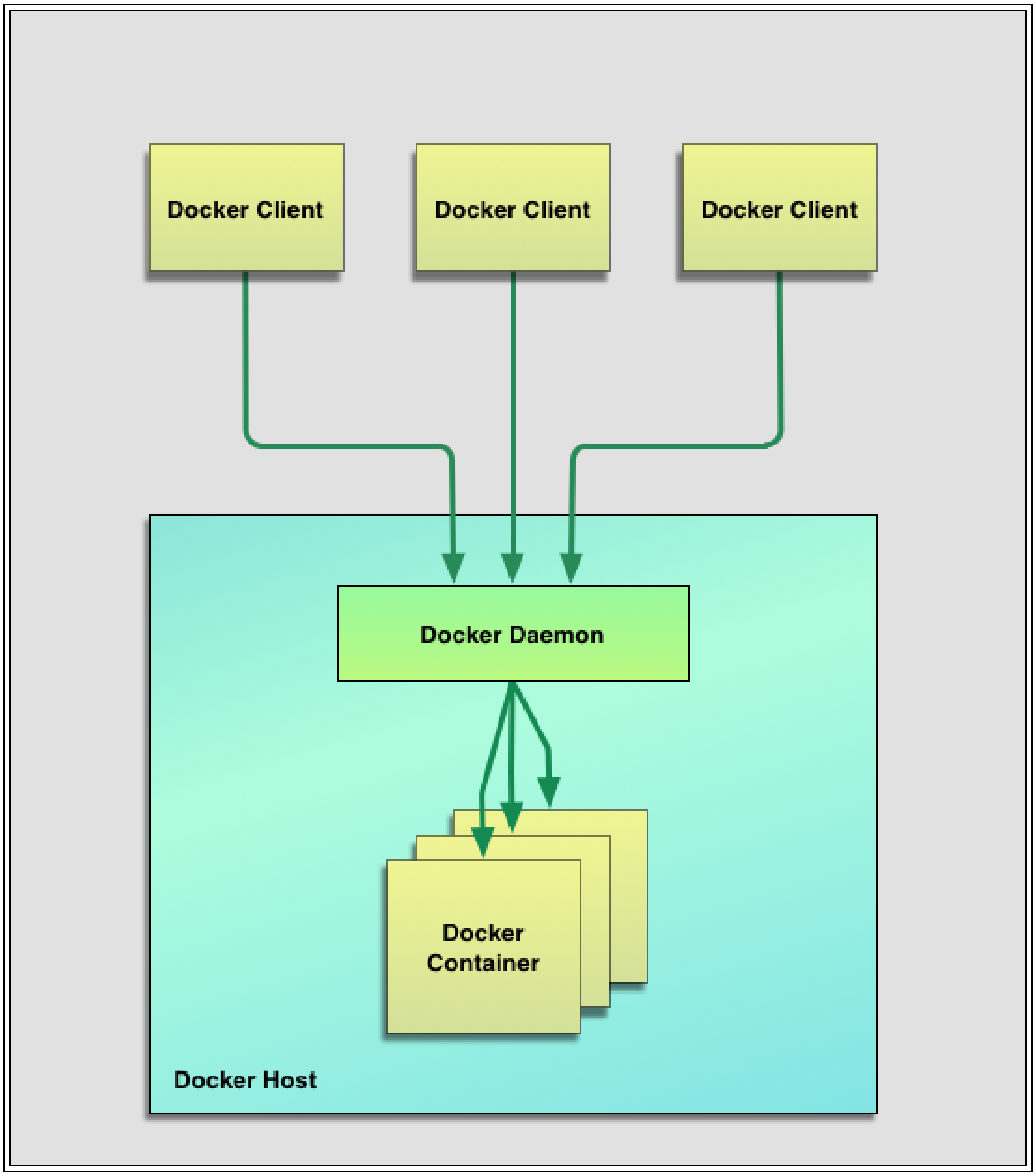


Figure 1.1: Docker architecture

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 have 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.

Compose, Swarm and Kubernetes

In addition to solitary containers we can also run Docker containers in stacks and clusters, what Docker calls swarms. The Docker ecosystem contains two more tools:

- Docker Compose - which allows you to run stacks of containers to represent application stacks, for example web server, application server and database server containers running together to serve a specific application.

- Docker Swarm - which allows you to create clusters of containers, called swarms, that allow you to run scalable workloads.

We'll look at Docker Compose and Swarm in Chapter 7.

In addition to Compose and Swarm, Docker provides the primary underlying compute layer in the orchestration tool Kubernetes

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;
- Highly performant, hyperscale deployments of hosts.

You can see a list of some of the early projects built on and around the Docker ecosystem in the blog post 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 are some legitimate arguments 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 supplement 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 3.10 and later. It has low overhead and can be used on servers, desktops, or laptops. Run inside a virtual machine, you can also deploy Docker on OS X and Microsoft Windows. It includes:

- A native Linux container format that Docker calls `libcontainer`.
- 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 components. 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 and stacks of Docker containers.
- Deploy Docker containers as part of your testing pipeline.
- Build multi-container applications and environments.

- Introduce the basics of Docker orchestration with Docker Compose, Consul, and Swarm.
- 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 introduces 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
- The Docker Forum
- Docker on IRC: irc.freenode.net and channel #docker
- Docker on Twitter
- Get Docker help on StackOverflow

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, Debian or the RHEL family (CentOS, Fedora, etc) 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 Docker for Mac.
- On Microsoft Windows using Docker for Windows.

TIP Docker for Mac and Docker for Windows are a collection of components that installs everything you need to get started with Docker. It includes 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`, is installed natively on these platforms and connected to the Docker daemon running in the virtual machine. It replaces the legacy Docker Toolbox and Boot2Docker.

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.

TIP You can find a full list of installation targets in the Docker installation guide.

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 Docker for Mac or Windows 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 architectures are **NOT** currently supported.
- Be running a Linux 3.10 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
  - ZFS (introduced in Docker 1.7)
  - The default storage driver is usually Device Mapper or AUFS.
- cgroups and namespaces kernel features must be supported and enabled.

Installing on Ubuntu and Debian

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

- Ubuntu Yakkety 16.10 (64-bit)
- Ubuntu Xenial 16.04 (64-bit)
- Ubuntu Trusty 14.04 (LTS) (64-bit)
- Debian Stretch (64-bit)
- Debian 8.0 Jessie (64-bit)
- Debian 7.7 Wheezy (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 16.04 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 4.4.0-64-generic #85-Ubuntu SMP Mon
   Feb 20 11:50:30 UTC 2017 x86_64 x86_64 x86_64 GNU/Linux
```

We see that we've got a 4.4.0 x86_64 kernel installed. This is the default for Ubuntu 16.04 and later.

We also want to install the `linux-image-extra` and `linux-image-extra-virtual` packages that contain the `aufs` storage driver.

Listing 2.2: Installing the linux-image-extra package

```
$ sudo apt-get install linux-image-extra-$(uname -r) linux-image-
   extra-virtual
```

If we're using an earlier release of Ubuntu we may have an earlier kernel. We should be able to upgrade our Ubuntu to the later kernel via `apt-get`:

Listing 2.3: Installing a 3.10 or later kernel on Ubuntu

```
$ sudo apt-get update
$ sudo apt-get install linux-headers-3.16.0-34-generic linux-
   image-3.16.0-34-generic linux-headers-3.16.0-34
```

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 2.4: Updating the boot loader on Ubuntu Precise

```
$ sudo update-grub
```

After installation, we'll need to reboot our host to enable the new 3.10 or later kernel.

Listing 2.5: 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.

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 need to install some prerequisite packages.

Listing 2.6: Adding prerequisite Ubuntu packages

```
sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    software-properties-common
```

Then add the official Docker GPG key.

Listing 2.7: Adding the Docker GPG key

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo
   apt-key add -
```

And then add the Docker APT repository. You may be prompted to confirm that you wish to add the repository and have the repository's GPG key automatically added to your host.

Listing 2.8: Adding the Docker APT repository

```
$ sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"
```

The `lsb_release` command should populate the Ubuntu distribution version of your host.

Now, we update our APT sources.

Listing 2.9: Updating APT sources

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

We can now install the Docker package itself.

Listing 2.10: Installing the Docker packages on Ubuntu

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

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

TIP Prior to Docker 1.8.0 the package name was `lxc-docker` and between Docker 1.8 and 1.13 the package name was `docker-engine`.

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

Listing 2.11: 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 2.12: Old UFW forwarding policy

```
DEFAULT_FORWARD_POLICY="DROP"
```

To:

Listing 2.13: New UFW forwarding policy

```
DEFAULT_FORWARD_POLICY="ACCEPT"
```

Save the update and reload UFW.

Listing 2.14: 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) 7 and later (64-bit)
- Fedora 24 and later (64-bit)
- Oracle Linux 6 or 7 with Unbreakable Enterprise Kernel Release 3 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 3.10 or later kernel version. We can do this using the `uname` command like so:

Listing 2.15: 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 the Red Hat family of platforms should have a kernel that supports Docker.

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 2.16: 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 2.17: 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 2.18: 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 several package name changes across versions of Fedora. For Fedora 19, we need to install the `docker-io` package.

TIP On newer Red Hat and family versions the `yum` command has been replaced with the `dnf` command. The syntax is otherwise unchanged.

Listing 2.19: Installing the Docker package on Fedora 19

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

On Fedora 20, the package was renamed to `docker`.

Listing 2.20: Installing the Docker package on Fedora 20

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

For Fedora 21 the package name reverted back to `docker-io`.

Listing 2.21: Installing the Docker package on Fedora 21

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

Finally, on Fedora 22 the package name became `docker` again. Also on Fedora 22 the `yum` command was deprecated and replaced with the `dnf` command.

Listing 2.22: Installing the Docker package on Fedora 22

```
$ sudo dnf install docker
```

TIP For Oracle Linux you can find documentation on the Docker site.

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 2.23: Starting the Docker daemon on Red Hat Enterprise Linux 6

```
$ sudo service docker start
```

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

Listing 2.24: Ensuring the Docker daemon starts at boot on Red Hat Enterprise Linux 6

```
$ sudo service docker enable
```

On Red Hat Enterprise Linux 7 and Fedora.

Listing 2.25: Starting the Docker daemon on Red Hat Enterprise Linux 7

```
$ sudo systemctl start docker
```

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

Listing 2.26: Ensuring the Docker daemon starts at boot on Red Hat Enterprise Linux 7

```
$ sudo systemctl enable docker
```

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

Listing 2.27: Checking Docker is installed on the Red Hat family

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

TIP Or you can directly download the latest RPMs from the Docker site for RHEL, CentOS and Fedora.

Docker for Mac

If you're using OS X, you can quickly get started with Docker using Docker for Mac. Docker for Mac is a collection of Docker components including 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.

Docker for Mac ships with a variety of components:

- Hyperkit.

- The Docker client and server.
- Docker Compose (see Chapter 7).
- Docker Machine - Which helps you create Docker hosts.
- Kitematic - is a GUI that helps you run Docker locally and interact with the Docker Hub.

Installing Docker for Mac

To install Docker for Mac we need to download its installer. You can find it here.

Let's grab the current release:

Listing 2.28: Downloading the Docker for Mac DMG file

```
$ wget https://download.docker.com/mac/stable/Docker.dmg
```

Launch the downloaded installer and follow the instructions to install Docker for Mac.

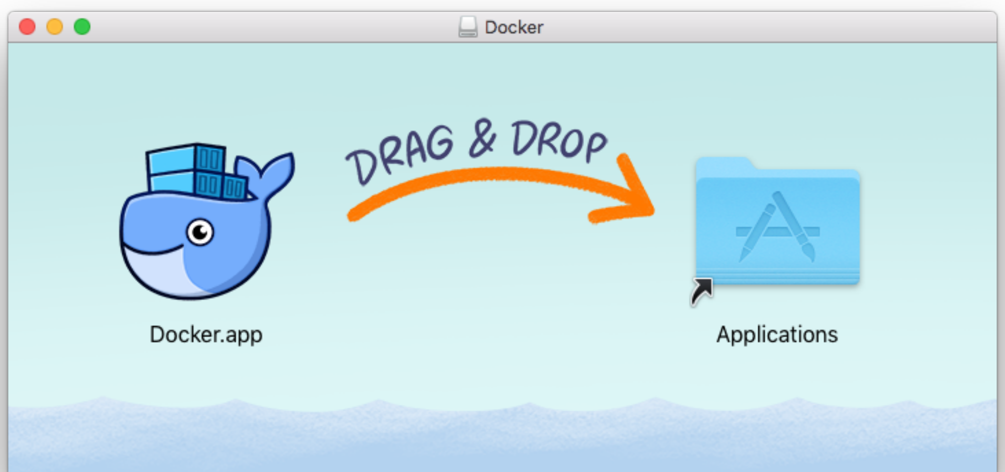


Figure 2.1: Installing Docker for Mac on OS X

Testing Docker for Mac

We can now test that our Docker for Mac installation is working by trying to connect our local client to the Docker daemon running on the virtual machine. Make sure the `Docker.app` is running and then open a terminal window and type:

Listing 2.29: Testing Docker for Mac on OS X

```
$ docker info
Containers: 2
 Running: 0
 Paused: 0
 Stopped: 2
Images: 13
Server Version: 1.12.1
Storage Driver: aufs
. . .
```

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

There's a lot more you can use and configure with Docker for Mac and you can read its documentation on the Docker for Mac site.

Docker for Windows installation

If you're using Microsoft Windows, you can quickly get started with Docker using Docker for Windows. Docker for Windows is a collection of Docker components including a tiny Hyper-V virtual machine with a supporting command line tool that is installed on a Microsoft Windows host and provides you with a Docker environment.

Docker for Windows ships with a variety of components:

- The Docker client and server.
- Docker Compose (see Chapter 7).
- Docker Machine - Which helps you create Docker hosts.
- Kitematic - is a GUI that helps you run Docker locally and interact with the Docker Hub.

Docker for Windows requires 64bit Windows 10 Pro, Enterprise or Education (with the 1511 November update, Build 10586 or later) and Microsoft Hyper-V. If your host does not satisfy these requirements, you can install the Docker Toolbox, which uses Oracle Virtual Box instead of Hyper-V.

TIP You can also install a Docker client via the Chocolatey package manager.

Installing Docker for Windows

To install Docker for Windows we need to download its installer. You can find it here.

Let's grab the current release:

Listing 2.30: Downloading the Docker for Windows .MSI file

```
https://download.docker.com/win/stable/InstallDocker.msi
```

Launch the downloaded installer and follow the instructions to install Docker for Windows.

Testing Docker for Windows

We can now test that our Docker for Windows installation is working by trying to connect our local client to the Docker daemon running on the virtual machine. Ensure the `Docker` application is running and open a terminal window and run:

Listing 2.31: Testing Docker for Windows

```
$ docker info
Containers: 2
 Running: 0
 Paused: 0
 Stopped: 2
Images: 13
Server Version: 1.12.1
Storage Driver: aufs
. . .
```

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

There's a lot more you can use and configure with Docker for Windows and you can read its documentation on the Docker for Windows site.

Using Docker on OSX and Windows with this book

If you are following the examples in this book you will sometimes be asked to use volumes or the `docker run` command with the `-v` flag to mount a local directory into a Docker container may not work on Windows. You can't mount a local directory on host into the Docker host running in the Docker virtual machine because they don't share a file system. If you want to use any examples with volumes, such as those in Chapters 5 and 6, I recommend you run Docker on a Linux-based host.

It's also worth reading the Docker for Mac File Sharing section or Docker for Windows Shared Drive section. This allows you enable volume usage by mounting directories into the Docker for Mac and Docker for Windows client applications.

TIP All the examples in the book assume you are using the latest Docker for Mac or Docker for Windows versions.

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.com 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 2.32: 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 2.33: Installing curl on Ubuntu

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

Or we can use `yum` or the newer `dnf` command on Fedora.

Listing 2.34: Installing curl on Fedora

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

Now we can use the script to install Docker.

Listing 2.35: Installing Docker from the installation script

```
$ curl https://get.docker.com/ | 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 2.36: Downloading the Docker binary

```
$ wget http://get.docker.com/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. You can control the Docker daemon via the `dockerd` binary.

NOTE Prior to Docker 1.12 the daemon was controlled with the `docker daemon` subcommand.

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 2.37: Changing Docker daemon networking

```
$ sudo dockerd -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 2.38: 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.sock`:

Listing 2.39: Binding the Docker daemon to a different socket

```
$ sudo dockerd -H unix:///home/docker.sock
```

Or we can specify multiple bindings like so:

Listing 2.40: Binding the Docker daemon to multiple places

```
$ sudo dockerd -H tcp://0.0.0.0:2375 -H unix:///home/docker.sock
```

TIP If you're running Docker behind a proxy or corporate firewall you can also use the `HTTPS_PROXY`, `HTTP_PROXY`, `NO_PROXY` options to control how the daemon connects.

We can also increase the verbosity of the Docker daemon by using the `-D` flag.

Listing 2.41: Turning on Docker daemon debug

```
$ sudo dockerd -D
```

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

On systemd-enabled distributions we would add an override file at:

/etc/systemd/system/docker.service.d/override.conf

With content like:

Listing 2.42: The systemd override file

```
[Service]
ExecStart=
ExecStart=/usr/bin/dockerd -H ...
```

In earlier Red Hat and Fedora releases, we'd edit the /etc/sysconfig/docker file.

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 2.43: 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 2.44: Starting and stopping Docker with Upstart

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

On systemd-enabled Ubuntu and Debian releases, as well as Red Hat and Fedora, we can do similarly using the `service` shortcuts.

Listing 2.45: Starting and stopping Docker on Red Hat and Fedora

```
$ sudo service docker stop
$ sudo service docker start
```

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

Listing 2.46: The Docker daemon isn't running

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

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. We're using the `apt-get install` command because the `docker-engine` (formerly `lxc-docker`) package is usually pinned.

Listing 2.47: Upgrade docker

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

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 from `https://github.com/ehazlett/shipyard`.
- Portainer - UI for Docker is a web interface that allows you to interact with the Docker Remote API. It's written in JavaScript using the AngularJS framework.
- Kitematic - is a GUI for OS X and Windows that helps you run Docker locally and interact with the Docker Hub. It's a free product released by Docker Inc.

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.

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 into 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 that the docker binary works

```
$ sudo docker info
Containers: 33
 Running: 0
 Paused: 0
 Stopped: 33
Images: 217
Server Version: 1.12.0
Storage Driver: aufs
 Root Dir: /var/lib/docker/aufs
 Backing Filesystem: extfs
 Dirs: 284
 Dirperm1 Supported: false
Logging Driver: json-file
Cgroup Driver: cgroupfs
. . .
Username: jamtur01
Registry: https://index.docker.io/v1/
WARNING: No swap limit support
Insecure Registries:
 127.0.0.0/8
```

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 two binaries, the Docker server provided via the `dockerd` binary and the `docker` binary, that acts as a client. 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.

NOTE Prior to Docker 1.12 all of this functionality was provided by a single binary: `docker`.

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`). This will not work on Docker for Mac or Docker for OSX as no man pages are shipped.

Listing 3.2: Running our first container

```
$ sudo docker run -i -t ubuntu /bin/bash
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
43db9dbdcb30: Pull complete
2dc64e8f8d4f: Pull complete
670a583e1b50: Pull complete
183b0bfcd10e: Pull complete
Digest: sha256:
    c6674c44c6439673bf56536c1a15916639c47ea04c3d6296c5df938add67b54b

Status: Downloaded newer image for ubuntu:latest
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.

TIP We'll hear a lot more about images in Chapter 4, including how to build our own images. Throughout the book we use the `ubuntu` image. This is a reasonably heavyweight image, measuring a couple of hundred megabytes in size. If you'd prefer something smaller the Alpine Linux image is recommended as extremely lightweight, generally 5Mb in size for the base image. Its image name is `alpine`.

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:/#
```

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 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:/# hostname -I
172.17.0.4
```

We see that we have 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.

Listing 3.10: Listing Docker containers

```
CONTAINER ID IMAGE   COMMAND     CREATED  STATUS PORTS NAMES
1cd57c2cdf7f ubuntu  "/bin/bash" A minute Exited       gray_cat
```

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. You can also use the `--format` flag to further control what and how information is outputted.

We 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 `f7cbdac22a02e03c9438c729345e54db9d20cfa2ac1fc3494b6eb60872e74778`), and a name (like `gray_cat`).

Container naming

Docker will automatically generate a name at random for each container we create. We 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 3.11: 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.

NOTE We'll see more about how to connect Docker containers in Chapter 5.

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 3.12: Starting a stopped container

```
$ sudo docker start bob_the_container
```

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

Listing 3.13: 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.

NOTE In a similar vein there is also the `docker create` command which creates a container but does not run it. This allows you more granular control over your container workflow.

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 3.14: Attaching to a running container

```
$ sudo docker attach bob_the_container
```

or via its container ID:

Listing 3.15: 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 3.16: 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 3.17: 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 see a running container.

Listing 3.18: Viewing our running daemon_dave container

```
CONTAINER ID IMAGE    COMMAND              CREATED      STATUS
    PORTS NAMES
1333bb1a66af ubuntu   /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 3.19: Fetching the logs of our daemonized container

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

Here we 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 3.20: Tailing the logs of our daemonized container

```
$ sudo docker logs -f daemon_dave
hello world
hello world
hello world
hello world
hello world
hello world
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` and `--tail` 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 3.21: Tailing the logs of our daemonized container

```
$ sudo docker logs -ft daemon_dave
2016-08-02T03:31:16.743679596Z hello world
2016-08-02T03:31:17.744769494Z hello world
2016-08-02T03:31:18.745786252Z hello world
2016-08-02T03:31:19.746839926Z hello world
. . .
```

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

Docker log drivers

Since Docker 1.6 you can also control the logging driver used by your daemon and container. This is done using the `--log-driver` option. You can pass this option to both the daemon and the `docker run` command.

There are a variety of options including the default `json-file` which provides the behavior we've just seen using the `docker logs` command.

Also available is `syslog` which disables the `docker logs` command and redirects all container log output to Syslog. You can specify this with the Docker daemon to output all container logs to Syslog or override it using `docker run` to direct output from individual containers.

Listing 3.22: Enabling Syslog at the container level

```
$ sudo docker run --log-driver="syslog" --name daemon_dwayne -d
    ubuntu /bin/sh -c "while true; do echo hello world; sleep 1;
    done"
. . .
```

This will cause the `daemon_dwayne` container to log to Syslog and result in the `docker logs` command showing no output.

Lastly, also available is `none`, which disables all logging in containers and results in the `docker logs` command being disabled.

TIP Additional logging drivers continue to be added. Docker 1.8 introduced support for Graylog's GELF protocol, Fluentd and a log rotation driver.

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 3.23: 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 3.24: 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
```

Docker statistics

In addition to the `docker top` command you can also use the `docker stats` command. This shows statistics for one or more running Docker containers. Let's see what these look like. We're going to look at the statistics for our `daemon_dave` and `daemon_dwayne` containers.

Listing 3.25: The docker stats command

```
$ sudo docker stats daemon_dave daemon_dwayne
CONTAINER     CPU %  MEM USAGE/LIMIT  MEM %  NET I/O
   BLOCK I/O
daemon_dave   0.14%  212 KiB/994 MiB  0.02%  5.062 KiB/648 B 1.69
    MB / 0 B
daemon_dwayne 0.11%  216 KiB/994 MiB  0.02%  1.402 KiB/648 B
   24.43 MB / 0 B
```

We see a list of daemonized containers and their CPU, memory and network and storage I/O performance and metrics. This is useful for quickly monitoring a group of containers on a host.

NOTE The `docker stats` command was introduced in Docker 1.5.0.

Running a process inside an already running container

Since Docker 1.3 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 3.26: 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.

TIP Since Docker 1.7 you can use the `-u` flag to specify a new process owner for `docker exec` launched processes.

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

Listing 3.27: 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 3.28: Stopping the running Docker container

```
$ sudo docker stop daemon_dave
```

or again via its container ID.

Listing 3.29: 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 3.30: Automatically restarting containers

```
$ sudo docker run --restart=always --name daemon_alice -d ubuntu
    /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 3.31: 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 3.32: Inspecting a container

```
$ sudo docker inspect daemon_alice
[{
  "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 3.33: Selectively inspecting a container

```
$ sudo docker inspect --format='{{ .State.Running }}'
   daemon_alice
true
```

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

Listing 3.34: Inspecting the container's IP address

```
$ sudo docker inspect --format '{{ .NetworkSettings.IPAddress }}'
    daemon_alice
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 3.35: Inspecting multiple containers

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

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 Since Docker 1.6.2 you can delete a running Docker container using the `-f` flag to the `docker rm` command. Prior to this version you must stop the container first using the `docker stop` command or `docker kill` command.

Listing 3.36: 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 3.37: Deleting all containers

```
$ sudo docker rm -f `sudo 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. The `-f` flag force removes any running containers. If you'd prefer to protect those containers, omit the flag.

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 running 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.

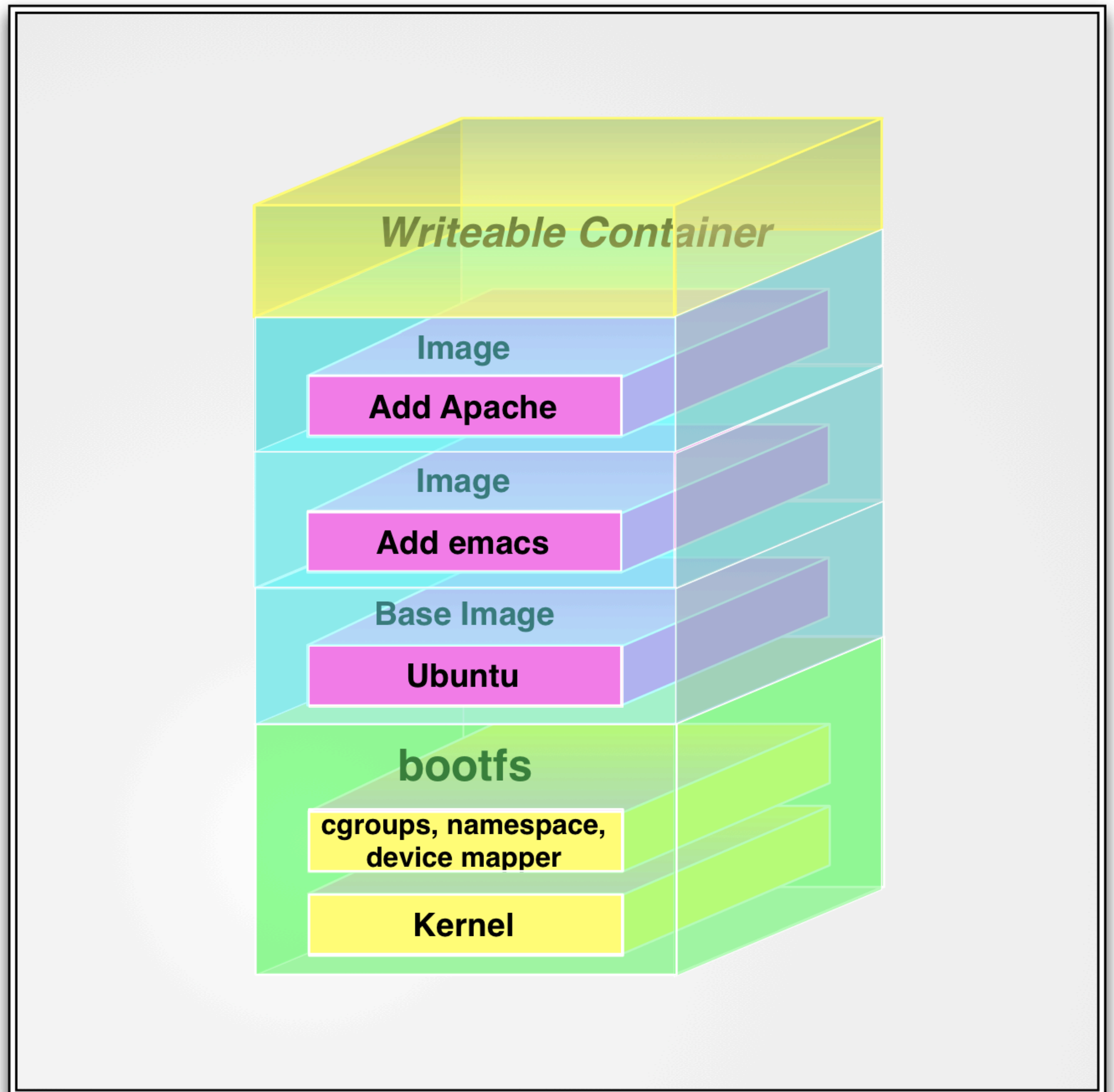


Figure 4.1: The Docker filesystem layers

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.

Listing 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 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, 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. The Docker Hub product is also available as a commercial "behind the firewall" product called Docker Trusted Registry, formerly Docker Enterprise Hub.

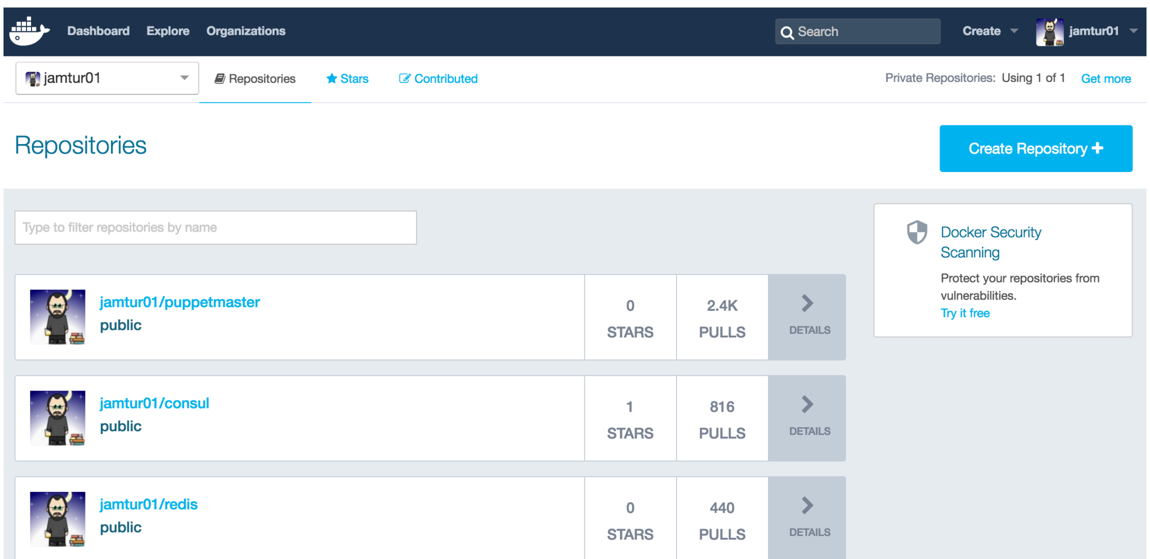


Figure 4.2: Docker Hub

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.

Each repository can contain multiple images (e.g., the `ubuntu` repository contains images for Ubuntu 12.04, 12.10, 13.04, 13.10, 14.04, 16.04). Let's get another image from the `ubuntu` repository now.

Listing 4.3: Pulling the Ubuntu 16.04 image

```
$ sudo docker pull ubuntu:16.04
16.04: Pulling from library/ubuntu
Digest: sha256:
   c6674c44c6439673bf56536c1a15916639c47ea04c3d6296c5df938add67b54b

Status: Downloaded newer image for ubuntu:16.04
```

Here we've used the `docker pull` command to pull down the Ubuntu 16.04 image from the `ubuntu` repository.

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

Listing 4.4: Listing the ubuntu Docker images

```
$ sudo docker images
REPOSITORY TAG     IMAGE ID     CREATED      VIRTUAL SIZE
ubuntu     latest  5506de2b643b 3 weeks ago  199.3 MB
ubuntu     16.04   0b310e6bf058 5 months ago 127.9 MB
```

TIP Throughout the book we use the `ubuntu` image. This is a reasonably heavyweight image, measuring a couple of hundred megabytes in size. If you'd prefer something smaller the Alpine Linux image is recommended as extremely lightweight, generally 5Mb in size for the base image. Its image name is `alpine`.

You can see we've now got the `latest` Ubuntu image and the `16.04` image. This shows us that the `ubuntu` image is actually a series of images collected under a single repository.

NOTE We call it the Ubuntu operating system, but really it is not the full operating system. It's a 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, `12.04`, `12.10`, `quantal`, or `precise` and so on. Each tag marks together a series of image layers that represent a specific image (e.g., the `16.04` tag collects together all the layers of the Ubuntu 16.04 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:16.04 /bin/
   bash
root@79e36bff89b4:/#
```

This launches a container from the `ubuntu:16.04` image, which is an Ubuntu 16.04 operating system.

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 14.04 and 16.04, so it would be useful to specifically state that we're using `ubuntu:16.04` 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.

In Docker 1.8 support was also added for managing the content security of images, essentially signed images. This is currently an optional feature and you can read more about it on the Docker blog.

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.

Alternatively, we can use the `docker pull` command to pull images down ourselves preemptively. Using `docker pull` saves us some time launching a container from a new image. Let's see that now by pulling down the 'fedora:21 base image.

Listing 4.7: Pulling the fedora image

```
$ sudo docker pull fedora:21
21: Pulling from library/fedora
d60b4509ad7d: Pull complete
Digest: sha256:4328
    c03e6cafef1676db038269fc9a4c3528700d04ca1572e706b4a0aa320000
Status: Downloaded newer image for fedora:21
```

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     21     7d3f07f8de5f 6 weeks ago 374.1 MB
```

We see that the `fedora:21` image has been downloaded. We could also download another 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
macadmins/puppetmaster Simple puppetmaster 21                    [
   OK]
devopsil/puppet         Dockerfile for a    18                    [
   OK]
. . .
```

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 an image.

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 server).

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 puppetmaster 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 in this guide.

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.

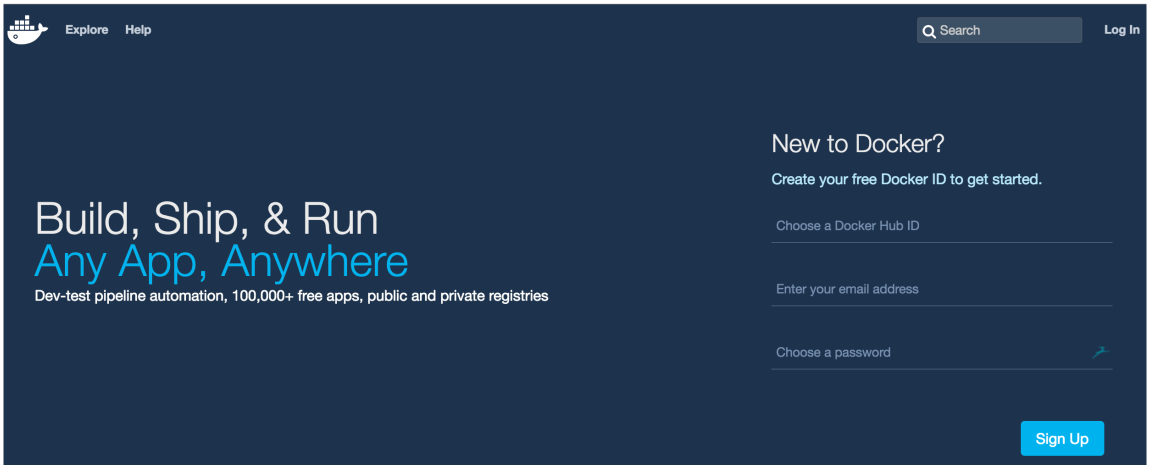


Figure 4.3: Creating a Docker Hub account.

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
Login with your Docker ID to push and pull images from Docker Hub
    . If you don't have a Docker ID, head over to https://hub.
    docker.com to create one.
Username (jamtur01): jamtur01
Password:
Login Succeeded
```

This command will log you into the Docker Hub and store your credentials for future use. You can use the `docker logout` command to log out from a registry server.

NOTE Your credentials will be stored in the `$HOME/.dockercfg` file. Since Docker 1.7.0 this is now `$HOME/.docker/config.json`.

Using Docker commit to create images

The first method of creating images uses 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 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" -a "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 `-a` 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
root@9c2d3a843b9e:/# service apache2 status
 * apache2 is not running
```

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 (Domain Specific Language) with instructions for building Docker images. We recommend the `Dockerfile` approach over `docker commit` because it provides a more repeatable, transparent, and idempotent mechanism for creating images.

Once we have a `Dockerfile` 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:18.04
LABEL maintainer="james@example.com"
RUN apt-get update; apt-get install -y nginx
RUN echo 'Hi, I am in your container' \
    >/var/www/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:18.04`. 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` must 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:16.04` image as our base image. This specification will build an image on top of an Ubuntu 16.04 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 `LABEL` instruction with a value of 'maintainer = "james@example.com", 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.

NOTE This `LABEL` instructions replaces the `MAINTAINER` instruction which was deprecated in Docker 1.13.0.

We've followed these instructions with two `RUN` instructions. The `RUN` instruction executes commands on the current image. The commands in our example: updating the installed APT repositories and installing the `nginx` package and then creating the `/var/www/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: A 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. You can expose ports at run time with the `docker run` command with the `--expose` option.

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:18.04
 ---> ba5877dc9bec
Step 1 : LABEL maintainer="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'
>/var/www/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 see here:

Listing 4.26: Building from a Git repository

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

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

TIP Since Docker 1.5.0 and later you can also specify a path to a file to use as a build source using the `-f` flag. For example, `docker build -t "jamtur01/static_web" -f /path/to/file`. The file specified doesn't need to be called `Dockerfile` but must still be within the build context.

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 treated as part of the build context, and therefore prevents them from being uploaded to the Docker daemon. 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:18.04
  ---> 8dbd9e392a96
Step 2 : LABEL maintainer="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, in this example using the image ID of `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 ngin` 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 `Dockerfile`s 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:18.04
LABEL maintainer="james@example.com"
ENV REFRESHED_AT 2016-07-01
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:16.04`. 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:21
LABEL maintainer="james@example.com"
ENV REFRESHED_AT 2016-07-01
RUN yum -q makecache
```

Which performs a similar caching 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 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

Let's 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 publishes 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 `32768` to `61000` 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.

The `docker run` command 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. The `-l` flag tells Docker to show us the last container launched.

Listing 4.36: Viewing the Docker port mapping

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

We 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`.

Or we could use the container name too.

Listing 4.38: The docker port command with container name

```
$ sudo docker port static_web 80
0.0.0.0:49154
```

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

Listing 4.39: Exposing a specific port with -p

```
$ sudo docker run -d -p 80:80 --name static_web_80 jamtur01/
   static_web nginx -g "daemon off;"
```

This will bind port 80 on the container to port 80 on the local host. 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 4.40: Binding to a different port

```
$ sudo docker run -d -p 8080:80 --name static_web_8080 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 4.41: Binding to a specific interface

```
$ sudo docker run -d -p 127.0.0.1:80:80 --name static_web_lb
    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 4.42: Binding to a random port on a specific interface

```
$ sudo docker run -d -p 127.0.0.1::80 --name static_web_random
    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 publish all ports we've exposed via `EXPOSE` instructions in our `Dockerfile`.

Listing 4.43: Exposing a port with docker run

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

This would publish port 80 on a random port on our local host. It would also publish 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 4.44: Connecting to the container via curl

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

Now we've got a 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`, `LABEL`, `STOPSIGNAL`, `ARG`, `SHELL`, `HEALTHCHECK` and `ENV`. You can see a full list of the available `Dockerfile` instructions here.

We'll also see a lot more `Dockerfile`s 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 4.45: 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 4.46: Using the CMD instruction

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

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

Listing 4.47: 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 4.48: 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 4.49: 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 4.50: 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 4.51: Specifying an ENTRYPOINT

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

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

Listing 4.52: 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 4.53: 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 4.54: Using docker run with ENTRYPOINT

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

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.

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 4.55: 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 4.56: Using the WORKDIR instruction

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

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 `rackup`.

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

Listing 4.57: 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 4.58: 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 4.59: Prefixing a RUN instruction

```
RUN gem install unicorn
```

would be executed as:

Listing 4.60: Executing with an ENV prefix

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

You can specify single environment variables in an `ENV` instruction or since Docker 1.4 you can specify multiple variables like so:

Listing 4.61: Setting multiple environment variables using ENV

```
ENV RVM_PATH=/home/rvm RVM_ARCHFLAGS="-arch i386"
```

We can also use these environment variables in other instructions.

Listing 4.62: Using an environment variable in other Dockerfile instructions

```
ENV TARGET_DIR /opt/app
WORKDIR $TARGET_DIR
```

Here we've specified a new environment variable, `TARGET_DIR`, and then used its value in a `WORKDIR` instruction. Our `WORKDIR` instruction would now be set to `/opt`

/app.

NOTE You can also escape environment variables when needed by prefixing them with a backslash.

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

Listing 4.63: 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 4.64: 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
```

Now 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 4.65: 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 4.66: 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 even if 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 4.67: Using the VOLUME instruction

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

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

TIP Also useful and related is the `docker cp` command. This allows you to copy files to and from your containers. You can read about it in the Docker command line documentation.

Or we can specify multiple volumes by specifying an array:

Listing 4.68: 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 you can read more about volumes in the Docker volumes documentation.

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 4.69: 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 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 4.70: 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 4.71: 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 4.72: 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.

LABEL

The `LABEL` instruction adds metadata to a Docker image. The metadata is in the form of key/value pairs. Let's see an example.

Listing 4.73: Adding LABEL instructions

```
LABEL version="1.0"
LABEL location="New York" type="Data Center" role="Web Server"
```

The `LABEL` instruction is written in the form of `label="value"`. You can specify one item of metadata per label or multiple items separated with white space. We recommend combining all your metadata in a single `LABEL` instruction to save creating multiple layers with each piece of metadata. You can inspect the labels on an image using the `docker inspect` command..

Listing 4.74: Using docker inspect to view labels

```
$ sudo docker inspect jamtur01/apache2
. . .

    "Labels": {
      "version": "1.0",
      "location": "New York",
      "type": "Data Center",
      "role": "Web Server"
    },
```

Here we see the metadata we just defined using the `LABEL` instruction.

NOTE The `LABEL` instruction was introduced in Docker 1.6.

STOPSIGNAL

The `STOPSIGNAL` instruction sets the system call signal that will be sent to the container when you tell it to stop. This signal can be a valid number from the kernel syscall table, for instance 9, or a signal name in the format `SIGNAME`, for instance `SIGKILL`.

> **NOTE** The `STOPSIGNAL` instruction was introduced in Docker 1.9.

ARG

The `ARG` instruction defines variables that can be passed at build-time via the `docker build` command. This is done using the `--build-arg` flag. You can only specify build-time arguments that have been defined in the `Dockerfile`.

Listing 4.75: Adding ARG instructions

```
ARG build
ARG webapp_user=user
```

The second `ARG` instruction sets a default, if no value is specified for the argument at build-time then the default is used. Let's use one of these arguments in a `docker build` now.

Listing 4.76: Using an ARG instruction

```
$ docker build --build-arg build=1234 -t jamtur01/webapp .
```

As the `jamtur01/webapp` image is built the `build` variable will be set to `1234` and the `webapp_user` variable will inherit the default value of `user`.

WARNING At this point you're probably thinking - this is a great way to pass secrets like credentials or keys. Don't do this. Your credentials will be exposed during the build process and in the build history of the image.

Docker has a set of predefined `ARG` variables that you can use at build-time without a corresponding `ARG` instruction in the `Dockerfile`.

Listing 4.77: The predefined ARG variables

```
HTTP_PROXY
http_proxy
HTTPS_PROXY
https_proxy
FTP_PROXY
ftp_proxy
NO_PROXY
no_proxy
```

To use these predefined variables, pass them using the `--build-arg <variable>=<value>` flag to the `docker build` command.

NOTE The `ARG` instruction was introduced in Docker 1.9 and you can read more about it in the Docker documentation.

SHELL

The `SHELL` instruction allows the default shell used for the shell form of commands to be overridden. The default shell on Linux is '`["/bin/sh", "-c"]` and on Windows is `["cmd", "/S", "/C"]`.

The `SHELL` instruction is useful on platforms such as Windows where there are multiple shells, for example running commands in the `cmd` or `powershell` environments. Or when need to run a command on Linux in a specific shell, for example Bash.

The `SHELL` instruction can be used multiple times. Each new `SHELL` instruction overrides all previous `SHELL` instructions, and affects any subsequent instructions.

HEALTHCHECK

The `HEALTHCHECK` instruction tells Docker how to test a container to check that it is still working correctly. This allows you to check things like a web site being served or an API endpoint responding with the correct data, allowing you to identify issues that appear, even if an underlying process still appears to be running normally.

When a container has a health check specified, it has a health status in addition to its normal status. You can specify a health check like:

Listing 4.78: Specifying a HEALTHCHECK instruction

```
HEALTHCHECK --interval=10s --timeout=1m --retries=5 CMD curl http
    ://localhost || exit 1
```

The `HEALTHCHECK` instruction contains options and then the command you wish to run itself, separated by a `CMD` keyword.

We've first specified three default options:

- `--interval` - defaults to 30 seconds. This is the period between health checks. In this case the first health check will run 10 seconds after container launch and subsequently every 10 seconds.
- `--timeout` - defaults to 30 seconds. If the health check takes longer the timeout then it is deemed to have failed.
- `--retries` - defaults to 3. The number of failed checks before the container is marked as unhealthy.

The command after the `CMD` keyword can be either a shell command or an exec array, for example as we've seen in the `ENTRYPOINT` instruction. The command should exit with `0` to indicate health or `1` to indicate an unhealthy state. In our `CMD` we're executing `curl` on the `localhost`. If the command fails we're exiting with an exit code of `1`, indicating an unhealthy state.

We can see the state of the health check using the `docker inspect` command.

Listing 4.79: Docker inspect the health state

```
$ sudo docker inspect --format '{{.State.Health.Status}}'
   static_web
healthy
```

The health check state and related data is stored in the `.State.Health` namespace and includes current state as well as a history of previous checks and their output. The output from each health check is also available via `docker inspect`.

Listing 4.80: Health log output

```
$ sudo docker inspect --format '{{range .State.Health.Log}} {{.
   ExitCode}} {{.Output}} {{end}}' static_web
  0  Hi, I am in your container
```

Here we're iterating through the array of `.Log` entries in the `docker inspect` output.

There can only be one `HEALTHCHECK` instruction in a `Dockerfile`. If you list more than one then only the last will take effect.

You can also disable any health checks specified in any base images you may have inherited with the instruction:

Listing 4.81: Disabling inherited health checks

```
HEALTHCHECK NONE
```

NOTE This instruction was added in Docker 1.12.

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 4.82: 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 see when we run `docker inspect` on the image.

Listing 4.83: 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 4.84: A new ONBUILD image Dockerfile

```
FROM ubuntu:18.04
LABEL maintainer="james@example.com"
RUN apt-get update; apt-get install -y apache2
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
ONBUILD ADD . /var/www/
EXPOSE 80
ENTRYPOINT ["/usr/sbin/apachectl"]
CMD ["-D", "FOREGROUND"]
```

Now we'll build this image.

Listing 4.85: 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 4.86: The webapp Dockerfile

```
FROM jamtur01/apache2
LABEL maintainer="james@example.com"
ENV APPLICATION_NAME webapp
ENV ENVIRONMENT development
```

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

Listing 4.87: 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 : LABEL maintainer="james@example.com"
...
Successfully built 04829a360d86
```

We 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 another 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 build an image without a user prefix and try and push it now.

Listing 4.88: Trying to push a root image

```
$ cd static_web
$ sudo docker build --no-cache -t="static_web" .
. . .
Successfully built a312a2ed58c7
$ sudo docker push static_web
The push refers to a repository [docker.io/library/static_web]
c0121fc36460: Preparing
8591faa9900d: Preparing
9a39129ae0ac: Preparing
98305c1a8f5e: Preparing
0185b3091e8e: Preparing
ea9f151abb7e: Waiting
unauthorized: authentication required
```

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 as unauthorized. Let's try again, rebuilding our image with a user prefix and then pushing it.

Listing 4.89: Pushing a Docker image

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

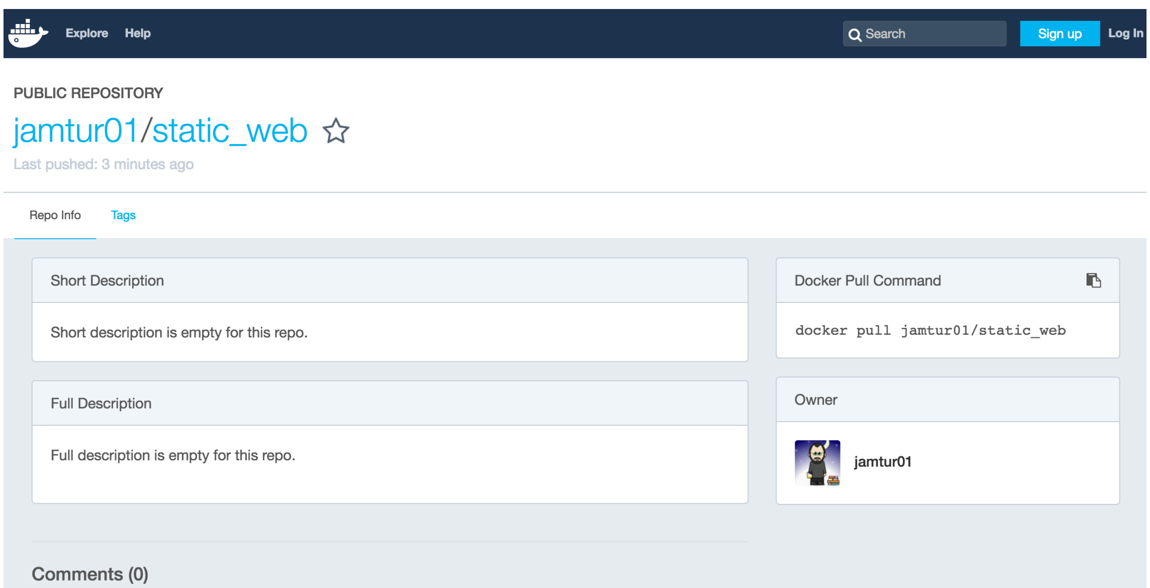


Figure 4.4: Your image on the Docker Hub.

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 `Create -> Create Automated Build` button.

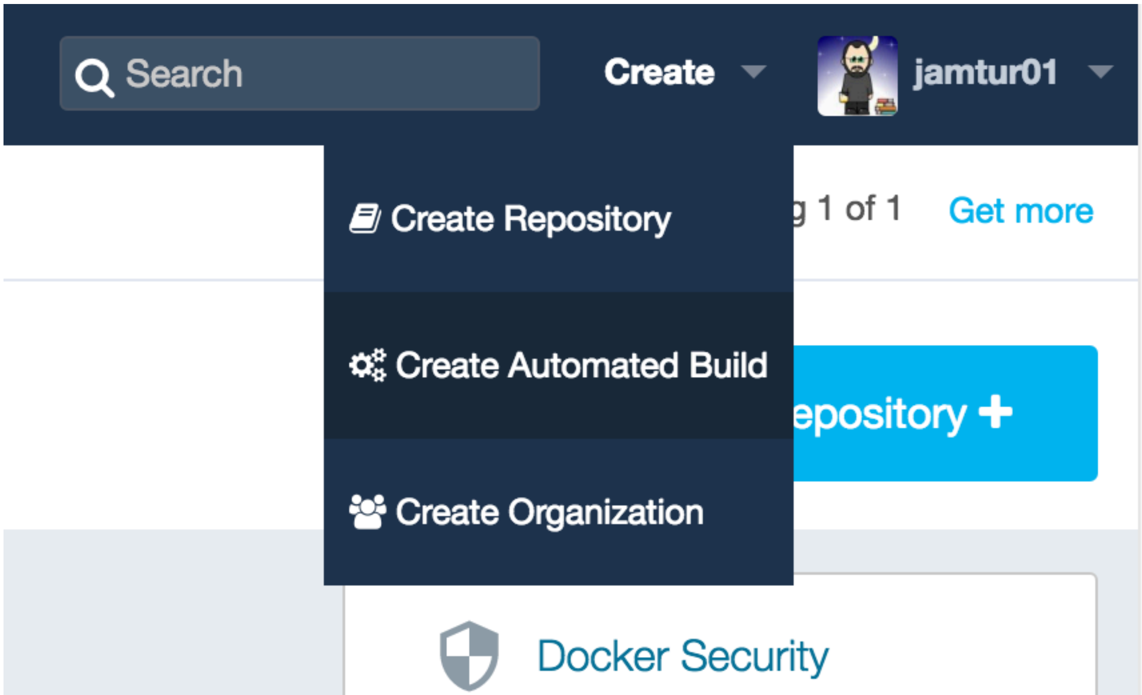


Figure 4.5: The Add Repository button.

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

On Github 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.

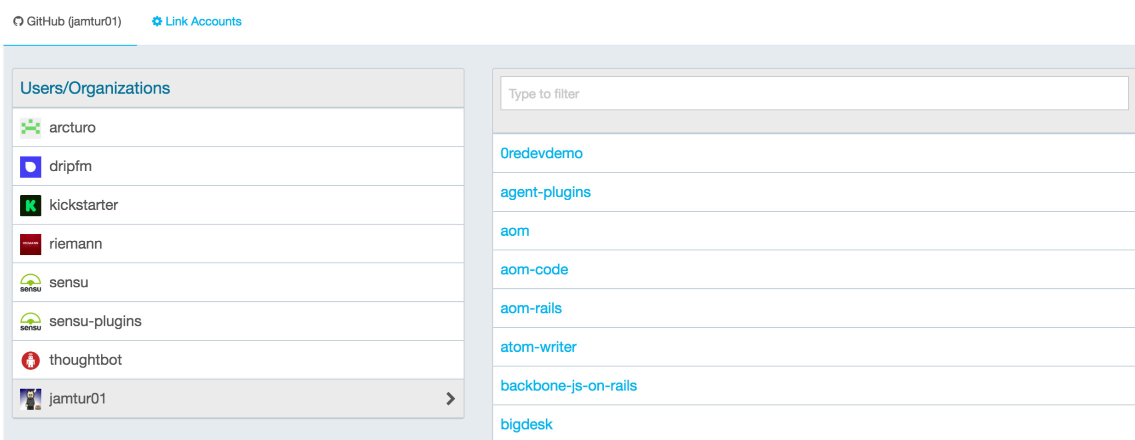


Figure 4.6: Selecting your repository.

Select the repository from which you wish to create an Automated Build and then configure the build.

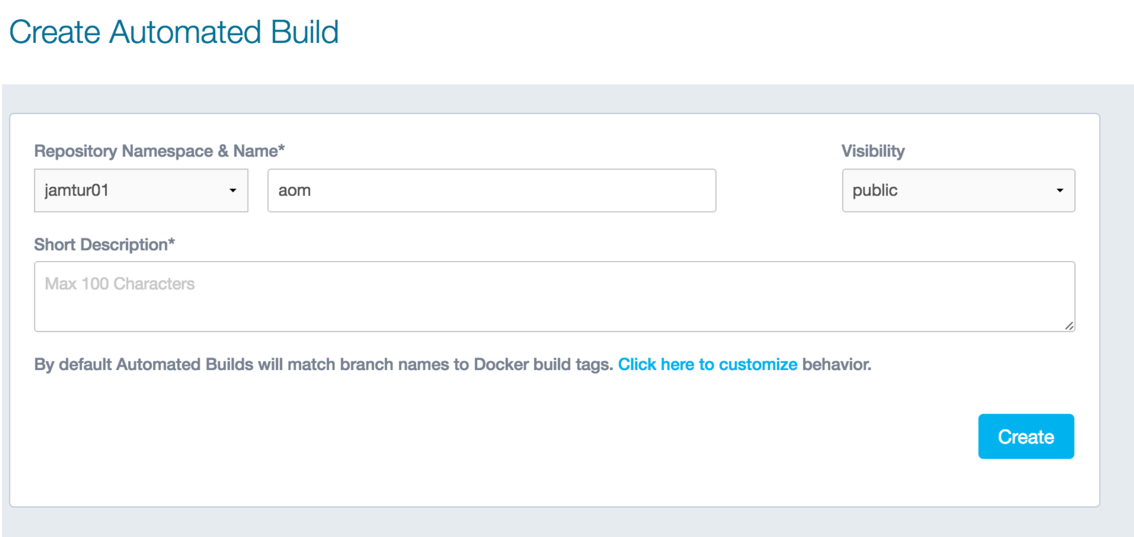


Figure 4.7: Configuring your Automated Build.

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` button to add your Automated Build to the Docker Hub.

You will now see your Automated Build submitted. Click on the `Build Details` 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 4.90: 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. If a running container is still using an image then you won't be able to delete it. You'll need to stop all containers running that image, remove them and then delete the image.

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 `Settings -> Delete` button.

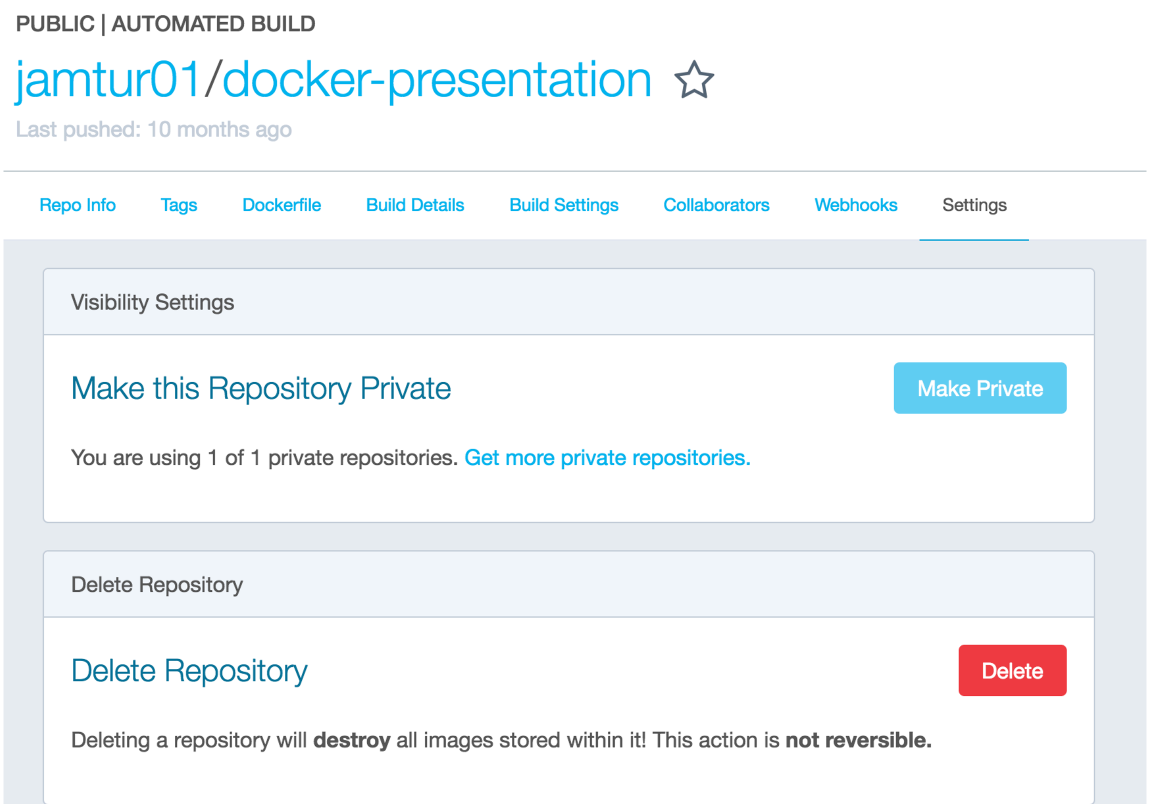


Figure 4.8: Deleting a repository.

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

Listing 4.91: 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 4.92: Deleting all images

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

Running your own Docker registry

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.

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

TIP If you're running Docker behind a proxy or corporate firewall you can also use the `HTTPS_PROXY`, `HTTP_PROXY`, `NO_PROXY` options to control how Docker connects.

Running a registry from a container

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

Listing 4.93: Running a container-based registry

```
$ docker run -d -p 5000:5000 --name registry registry:2
```

This will launch a container running version 2.0 of the registry application and bind port 5000 to the local host.

> **TIP** If you're running an older version of the Docker Registry, prior to 2.0, you can use the Migrator tool to upgrade to a new registry.

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 4.94: 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 4.95: 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 4.96: 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 4.97: 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 deployment 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 Hub. 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, including useful concepts like connecting Docker containers. The chapter 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. Such an environment allows you to replicate your production environment and ensure what you develop will also likely run in production. We're going to start with installing the Nginx web server 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 creating some structure, some configuration files for our container and a `Dockerfile` from which to build our image. 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
```

We're also going to need some Nginx configuration files to run our website. We can download some example files I've prepared earlier from GitHub into the `sample` directory.

Listing 5.2: Getting our Nginx configuration files

```
$ 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
```

Now let's look at the `Dockerfile` you're going to create for our Sample website.

Listing 5.3: The Dockerfile for the Sample website

```
FROM ubuntu:18.04
LABEL maintainer="james@example.com"
ENV REFRESHED_AT 2016-06-01
RUN apt-get -yqq update; apt-get -yqq install nginx
RUN mkdir -p /var/www/html/website
ADD global.conf /etc/nginx/conf.d/
ADD nginx.conf /etc/nginx/nginx.conf
EXPOSE 80
```

Here we've written a `Dockerfile` that:

- Installs Nginx.
- Creates a directory, `/var/www/html/website/`, 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 `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 file

```
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.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 on Docker Book GitHub site. 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
IMAGECREATED     CREATED BY    SIZE
f99cb0a6726d 7 secs ago  /bin/sh -c #(nop) EXPOSE 80/tcp       0
   B
d0741c80034e 7 secs ago  /bin/sh -c #(nop) ADD file:
   d6698a182fafaf3cb0 415 B
f1b8d3ab6b4f 8 secs ago  /bin/sh -c #(nop) ADD file:9778
   ae1b43896011cc 286 B
4e88da941d2b About a min /bin/sh -c mkdir -p /var/www/html/
   website     0 B
1224c6db31b7 About a min /bin/sh -c apt-get -yqq update; apt-get
   -yq 39.32 MB
2cfbed445367 About a min /bin/sh -c #(nop) ENV REFRESHED_AT=2016-
   06-01 0 B
6b5e0485e5fa About a min /bin/sh -c #(nop) LABEL maintainer= " 0
   B
91e54dfb1179 2 days ago  /bin/sh -c #(nop) CMD ["/bin/bash"]   0
   B
d74508fb6632 2 days ago  /bin/sh -c sed -i 's/^#\s*\(deb.*
   universe\)$/ 1.895 kB
c22013c84729 2 days ago  /bin/sh -c echo '#!/bin/sh' > /usr/sbin/
   polic 194.5 kB
d3a1f33e8a5a 2 days ago  /bin/sh -c #(nop) ADD file:5
   a3f9e9ab88e725d60 188.2 MB
```

The history starts with the final layer, our new `jamtur01/nginx` image and works backward to the original parent image, `ubuntu:16.04`. 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. To do that we need to add the Sample website's code. Let's download it now into the `sample` directory.

Listing 5.8: Downloading our Sample website

```
$ 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 run a container using the `docker run` command.

Listing 5.9: Running 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 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 directory or mount on the local host separated from the directory on the container with a `:`. If the container directory doesn't exist Docker will create it.

We can also specify the read/write status of the container directory by adding either `rw` or `ro` after that directory, 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 container directory `/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/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 on GitHub here. 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 see that it is active, it is named `website`, and port `80` on the container is mapped to port `49161` on the host.

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.

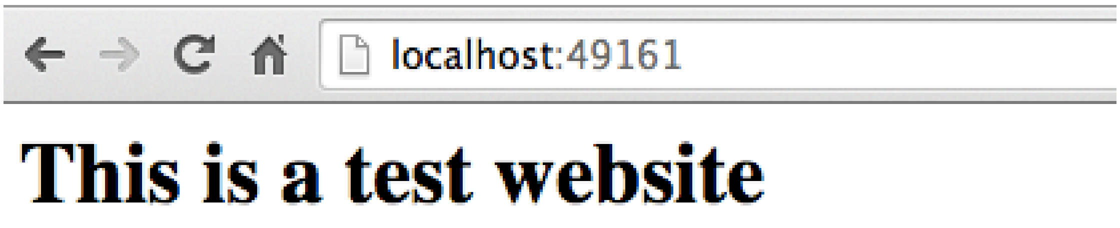


Figure 5.1: Browsing the Sample website.

Editing our website

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

Listing 5.12: Editing our Sample website

```
$ cd sample
$ 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 see that our Sample website has been updated. This is a 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. Sinatra is a Ruby-based web application framework. It contains a web application library and a simple Domain Specific Language or DSL for creating web applications. Unlike more complex web application frameworks, like Ruby on Rails, Sinatra does not follow the model–view–controller pattern but rather allows you to create quick and simple web applications.

As such it's perfect for creating a small sample application to test. In our case our new application is going to take incoming URL parameters and output them as a JSON hash. We're also going to take advantage of this application architecture to show you how to link Docker containers together.

Building our Sinatra application

Let's create a directory, `sinatra`, to hold our new application and any associated files we'll need for the build.

Listing 5.15: Create directory for web application testing

```
$ mkdir -p sinatra
$ cd sinatra
```

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

Listing 5.16: Dockerfile for our Sinatra container

```
FROM ubuntu:18.04
LABEL maintainer="james@example.com"
ENV REFRESHED_AT 2016-06-01

RUN apt-get update -yqq; apt-get -yqq 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. The `sinatra` and `json` gems contain Ruby's Sinatra library and support for JSON. The `redis` gem we're going to use a little later on to provide integration to a Redis database.

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 5.17: 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 at The Docker Book site. The application is made up of the `bin` and `lib` directories from the `webapp` directory.

Let's download it now into the `sinatra` directory.

Listing 5.18: Download our Sinatra web application

```
$ cd sinatra
$ wget --cut-dirs=7 -nH -r -e robots=off --reject="index.html","
   Dockerfile" --no-parent https://github.com/turnbullpress/
   dockerbook-code/tree/master/code/5/sinatra/webapp
```

Let's quickly look at the core of the `webapp` source code contained in the `webapp/lib/app.rb` file.

Listing 5.19: The Sinatra app.rb source code

```
require "rubygems"
require "sinatra"
require "json"

class App < Sinatra::Application

  set :bind, '0.0.0.0'

  get '/' do
    "<h1>DockerBook Test Sinatra app</h1>"
  end

  post '/json/?' do
    params.to_json
  end

end
```

This is a simple application that converts any parameters posted to the `/json` endpoint to JSON and displays them.

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

Listing 5.20: Making the webapp/bin/webapp binary executable

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

Now let's launch a new container from our image using the `docker run` command. To launch we should be inside the `sinatra` directory because we're going to mount our source code into the container using a volume.

Listing 5.21: 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, using the `webapp` directory 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're using the command we specified via the `CMD` instruction in the `Dockerfile` of the image.

Listing 5.22: 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 5.23: Checking the logs of our Sinatra container

```
$ sudo docker logs webapp
[2016-08-03 17:34:46] INFO  WEBrick 1.3.1
[2016-08-03 17:34:46] INFO  ruby 2.3.1 (2016-04-26) [x86_64-linux
   -gnu]
== Sinatra (v1.4.7) has taken the stage on 4567 for development
   with backup from WEBrick
[2016-08-03 17:34:46] 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 5.24: 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 5.25: 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 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 5.26: 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. 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 5.27: Testing our Sinatra application

```
$ curl -i -H 'Accept: application/json' \
-d 'name=Foo&status=Bar' http://localhost:49160/json
HTTP/1.1 200 OK
Content-Type: text/html;charset=utf-8
Content-Length: 29
X-Xss-Protection: 1; mode=block
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
Server: WEBrick/1.3.1 (Ruby/2.3.1/2016-04-26)
Date: Wed, 03 Aug 2016 18:30:06 GMT
Connection: Keep-Alive
{"name":"Foo","status":"Bar"}
```

We see that we've passed some URL parameters to our Sinatra application and 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 connecting to a service running in another container.

Extending our Sinatra application to use Redis

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

Updating our Sinatra application

Let's start with downloading an updated Sinatra-based application with a connection to Redis configured. From inside our `sinatra` directory let's download a Redis-enabled version of our application into a new directory: `webapp_redis`.

Listing 5.28: Download our updated Sinatra web application

```
$ cd sinatra
$ wget --cut-dirs=7 -nH -r -e robots=off --reject="index.html","
   Dockerfile" --no-parent https://github.com/turnbullpress/
   dockerbook-code/tree/master/code/5/sinatra/webapp_redis/
```

We see we've downloaded the new application. Let's look at its core code in `lib/app.rb` now.

Listing 5.29: The webapp_redis app.rb file

```
require "rubygems"
require "sinatra"
require "json"
require "redis"

class App < Sinatra::Application

      redis = Redis.new(:host => 'db', :port => '6379')

      set :bind, '0.0.0.0'

      get '/' do
"<h1>DockerBook Test Redis-enabled Sinatra app</h1>"
      end

      get '/json' do
params = redis.get "params"
params.to_json
      end

      post '/json/?' do
redis.set "params", [params].to_json
params.to_json
      end
end
```

NOTE You can see the full source for our updated Redis-enabled Sinatra application at The Docker Book site.

Our new application is basically the same as our previous application with support for Redis added. We now create a connection to a Redis database on a host called `db` on port `6379`. We also post our parameters to that Redis database and then get them back from it when required.

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

Listing 5.30: Making the webapp_redis/bin/webapp binary executable

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

Building a Redis database image

To build our Redis database, we're going to create a new image. Let's create a directory, `redis` inside our `sinatra` directory, to hold any associated files we'll need for the Redis container build.

Listing 5.31: Create directory for Redis container

```
$ mkdir redis
$ cd redis
```

Inside the `sinatra/redis` directory let's start with another `Dockerfile` for our Redis image.

Listing 5.32: Dockerfile for Redis image

```
FROM ubuntu:18.04
LABEL maintainer="james@example.com"
ENV REFRESHED_AT 2016-06-01
RUN apt-get -yqq update; apt-get -yqq 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 5.33: Building our Redis image

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

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

Listing 5.34: Launching a Redis container

```
$ sudo docker run -d -p 6379 --name redis jamtur01/redis
2df899db52baf469633459fa2abd34148ae4456a8c4a2343a0f372f2ee407756
```

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

Listing 5.35: Checking the Redis port

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

Our Redis port is published 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 on Ubuntu.

Listing 5.36: Installing the redis-tools package on Ubuntu

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

Or the `redis` package on Red Hat and related distributions.

Listing 5.37: Installing the redis package on Red Hat et al

```
$ sudo yum install -y -q redis
```

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

Listing 5.38: 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. You can use the `quit` command to exit the Redis CLI interface.

Connecting our Sinatra application 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 two ways we could do this using:

- Docker's own internal network.
- From Docker 1.9 and later, using Docker Networking and the `docker network` command.

So which method should I choose? Well the first method, Docker's internal network, is not an overly flexible or powerful solution. We're mostly going to discuss it to introduce you to how Docker networking functions. We don't recommend it as a solution for connecting containers.

The more realistic method for connecting containers is Docker Networking.

- Docker Networking can connect containers to each other across different hosts.
- Containers connected via Docker Networking can be stopped, started or restarted without needing to update connections.
- With Docker Networking you don't need to create a container before you can connect to it. You also don't need to worry about the order in which you run containers and you get internal container name resolution and discovery inside the network.

We're going to look at Docker Networking for connecting Docker containers together in the following sections.

Docker internal networking

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 published 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.

TIP Since Docker 1.5.0 IPv6 addresses are also supported. To enable this run the Docker daemon with the `--ipv6` flag.

Listing 5.39: 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
. . .
```

NOTE Depending on your distribution, you may need the `iproute2` package to run the `ip` command.

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 5.40: The veth interfaces

```
vethec6a  Link encap:Ethernet  HWaddr 86:e1:95:da:e2:5a
  inet6 addr: fe80::84e1:95ff:feda:e25a/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 sent 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 5.41: The eth0 interface in a container

```
$ sudo docker run -t -i ubuntu /bin/bash
root@b9107458f16a:/# hostname -I
172.17.0.29
```

We 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 5.42: 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 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.

Exit out of our container and let's look at the IPTables NAT configuration on our

Docker host.

Listing 5.43: 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 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 guide is useful.

Our Redis container's network

Let's examine our new Redis container and see its networking configuration using the `docker inspect` command.

Listing 5.44: Redis container's networking configuration

```
$ sudo docker inspect redis
. . .
    "NetworkSettings": {
      "Bridge": "",
. . .
      "Ports": {
    "6379/tcp": [
      {
"HostIp": "0.0.0.0",
"HostPort": "49161"
      }
    ]
      },
. . .
      "Gateway": "172.17.0.1",
      "GlobalIPv6Address": "",
      "GlobalIPv6PrefixLen": 0,
      "IPAddress": "172.17.0.18",
      "IPPrefixLen": 16,
      "IPv6Gateway": "",
      "MacAddress": "02:42:ac:11:00:08",
. . .
```

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 5.45: Finding the Redis container's IP address

```
$ sudo docker inspect -f '{{ .NetworkSettings.IPAddress }}' redis
172.17.0.18
```

Using the results of the `docker inspect` command we 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 5.46: Talking directly to the Redis container

```
$ redis-cli -h 172.17.0.18
redis 172.17.0.18:6379>
```

Once you've confirmed the connection is working you can exit the Redis interface using the `quit` command.

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 5.47: Restarting our Redis container

```
$ sudo docker restart redis
```

Let's inspect its IP address.

Listing 5.48: Finding the restarted Redis container's IP address

```
$ sudo docker inspect -f '{{ .NetworkSettings.IPAddress }}' redis
172.17.0.19
```

We 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 helpful.

Since Docker 1.9, Docker's networking has become a lot more flexible. Let's look at how we might connect our containers with this new networking framework.

Docker networking

Container connections are created using networks. This is called Docker Networking and was introduced in the Docker 1.9 release. Docker Networking allows you to setup your own networks through which containers can communicate. Essentially this supplements the existing `docker0` network with new, user managed networks. Importantly, containers can now communicate with each across hosts and your networking configuration can be highly customizable. Networking also integrates with Docker Compose and Swarm, we'll see more of both in Chapter 7.

NOTE The networking support is also pluggable, meaning you can add network drivers to support specific topologies and networking frameworks from vendors like Cisco and VMware.

To use Docker networks we first need to create a network and then launch a container inside that network.

Listing 5.49: Creating a Docker network

```
$ sudo docker network create app
ec8bc3a70094a1ac3179b232bc185fcda120dad85dec394e6b5b01f7006476d4
```

This uses the `docker network` command to create a bridge network called `app`. A network ID is returned for the network.

We can then inspect this network using the `docker network inspect` command.

Listing 5.50: Inspecting the app network

```
$ sudo docker network inspect app
[
 {
  "Name": "app",
  "Id": "ec8bc...",
  "Scope": "local",
  "Driver": "bridge",
  "IPAM": {
   "Driver": "default",
   "Config": [ {} ]
  },
  "Containers": {},
  "Options": {}
 }
]
```

Our new network is a local, bridged network much like our `docker0` network and that currently no containers are running inside the network.

TIP In addition to `bridge` networks, which exist on a single host, we can also create `overlay` networks, which allow us to span multiple hosts. You can read more about `overlay` networks in the Docker multi-host network documentation.

You can list all current networks using the `docker network ls` command.

Listing 5.51: The docker network ls command

```
$ sudo docker network ls
NETWORK ID          NAME       DRIVER
a74047bace7e        bridge     bridge
ec8bc3a70094        app        bridge
8f0d4282ca79        none       null
7c8cd5d23ad5        host       host
```

And you can remove a network using the `docker network rm` command.

Let's add some containers to our network, starting with a Redis container.

Listing 5.52: Creating a Redis container inside our Docker network

```
$ sudo docker run -d --net=app --name db jamtur01/redis
```

Here we've run a new container called `db` using our `jamtur01/redis` image. We've also specified a new flag: `--net`. The `--net` flag specifies a network to run our container inside.

Now if we re-run our `docker network inspect` command we'll see quite a lot more information.

Listing 5.53: The updated app network

```
$ sudo docker network inspect app
[
 {
  "Name": "app",
  "Id": "ec8bc3a...",
  "Scope": "local",
  "Driver": "bridge",
  "IPAM": {
    "Driver": "default",
    "Config": [ {} ]
  },
  "Containers": { "9a5ac1...": {
  "Name": "db"
  "EndpointID": "21a90...",
  "MacAddress": "02:42:ac:12:00:02",
  "IPv4Address": "172.18.0.2/16",
  "IPv6Address": "" }
  },
  "Options": {}
 }
]
```

Now, inside our network, we see a container with a MAC address and an IP address, `172.18.0.2`.

Now let's add a container to the network we've created. To do this we need to be back in the `sinatra` directory.

Listing 5.54: Linking our Redis container

```
$ cd sinatra
$ sudo docker run -p 4567 \
--net=app --name network_test -t -i \
jamtur01/sinatra /bin/bash
root@305c5f27dbd1:/#
```

We've launched a container named `network_test` inside the `app` network. We've launched it interactively so we can peek inside to see what's happening.

As the container has been started inside the `app` network, Docker will have taken note of all other containers running inside that network and populated their addresses in local DNS. Let's see this now in the `network_test` container.

We first need the `dnsutils` and `iputils-ping` packages to get the `nslookup` and `ping` binaries respectively.

Listing 5.55: Installing nslookup

```
root@305c5f27dbd1:/# apt-get install -y dnsutils iputils-ping
```

Then let's do the lookup.

Listing 5.56: DNS resolution in the network_test container

```
root@305c5f27dbd1:/# nslookup db
Server: 127.0.0.11
Address:127.0.0.11#53

Non-authoritative answer:
Name:   db
Address: 172.18.0.2
```

We see that using the `nslookup` command to resolve the `db` container it returns the IP address: `172.18.0.2`. A Docker network will also add the `app` network as a domain suffix for the network, any host in the `app` network can be resolved by `hostname.app`, here `db.app`. Let's try that now.

Listing 5.57: Pinging db.app in the network_test container

```
root@305c5f27dbd1:/# ping db.app
PING db.app (172.18.0.2) 56(84) bytes of data.
64 bytes from db (172.18.0.2): icmp_seq=1 ttl=64 time=0.290 ms
64 bytes from db (172.18.0.2): icmp_seq=2 ttl=64 time=0.082 ms
64 bytes from db (172.18.0.2): icmp_seq=3 ttl=64 time=0.111 ms
. . .
```

In our case we just need the `db` entry to make our application function. To make that work our webapp's Redis connection code already uses the `db` hostname.

Listing 5.58: The Redis DB hostname in code

```
redis = Redis.new(:host => 'db', :port => '6379')
```

We could now start our application and have our Sinatra application write its variables into Redis via the connection between the `db` and `webapp` containers that we've established via the `app` network.

Let's try it now by exiting the `network_test` container and starting up a new container running our Redis-enabled web application.

Listing 5.59: Starting the Redis-enabled Sinatra application

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

NOTE This is the Redis-enabled Sinatra application we installed earlier in the chapter. It's available on GitHub here.

Here we've launched a new container called `webapp_redis` running our Redis-enabled web application. Now let's just check, on the Docker host, what port our Sinatra container has bound the application.

Listing 5.60: Checking the Sinatra container's port mapping

```
$ sudo docker port webapp_redis 4567
0.0.0.0:49162
```

Okay port `4567` in the container is bound to port `49162` on the Docker host. Let's use this information to test our application from the Docker host using the `curl` command.

Listing 5.61: Testing our Redis-enabled Sinatra application

```
$ curl -i -H 'Accept: application/json' \
-d 'name=Foo&status=Bar' http://localhost:49162/json
HTTP/1.1 200 OK
Content-Type: text/html;charset=utf-8
Content-Length: 29
X-Xss-Protection: 1; mode=block
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
Server: WEBrick/1.3.1 (Ruby/2.3.1/2016-04-26)
Date: Wed, 03 Aug 2016 18:30:06 GMT
Connection: Keep-Alive
{"name":"Foo","status":"Bar"}
```

And now let's confirm that our Redis instance has received the update by querying the Sinatra web application in `webapp_redis`.

Listing 5.62: Confirming Redis contains data

```
$ curl -i http://localhost:49162/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!

Connecting existing containers to the network

You can also add already running containers to existing networks using the `docker network connect` command. So we can add an existing container to our `app` network. Let's say we have an existing container called `db2` that also runs Redis.

Listing 5.63: Running the db2 container

```
$ sudo docker run -d --name db2 jamtur01/redis
```

Let's add that to the `app` network (we could have also used the `--net` flag to automatically add the container to the network at runtime).

Listing 5.64: Adding a new container to the app network

```
$ sudo docker network connect app db2
```

Now if we inspect the `app` network we should see three containers.

Listing 5.65: The app network after adding db2

```
$ sudo docker network inspect app
. . .
"Containers": {
    "2
   fa7477c58d7707ea14d147f0f12311bb1f77104e49db55ac346d0ae961ac401
   ": {
"Name": "webapp_redis"
"EndpointID": "
   c510c78af496fb88f1b455573d4c4d7fdfc024d364689a057b98ea20287bfc0d
   ",
"MacAddress": "02:42:ac:12:00:02",
"IPv4Address": "172.18.0.2/16",
"IPv6Address": ""
    },
    "305
   c5f27dbd11773378f93aa58e86b2f710dbfca9867320f82983fc6ba79e779
   ": {

. . .

"Name": "db2"
"EndpointID": "47
   faec311dfac22f2ee8c1b874b87ce8987ee65505251366d4b9db422a749a1e
   ",
"MacAddress": "02:42:ac:12:00:04",
"IPv4Address": "172.18.0.4/16",
"IPv6Address": ""
    }
},
. . .
```

We can also disconnect a container from a network using the `docker network disconnect` command.

Listing 5.66: Disconnecting a host from a network

```
$ sudo docker network disconnect app db2
```

This would remove the `db2` container from the `app` network.

Containers can belong to multiple networks at once so you can create quite complex networking models.

TIP Further information on Docker Networking is available in the Docker documentation.

Connecting containers summary

We've now seen all the ways Docker can connect containers together. You can see that it is easy to create a fully functional web application stack consisting of:

- A web server container running Sinatra.
- A Redis database container.
- A secure connection between the two containers.

You should also be able to 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 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 in Docker that runs other Docker containers.

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, which actually make use of Docker.

Build a Jenkins and Docker server

To provide our Jenkins server, we're going to build an image from a `Dockerfile` that both installs Jenkins and Docker.

Listing 5.67: Jenkins and Docker Dockerfile

```
FROM jenkins
LABEL maintainer="james@example.com"
ENV REFRESHED_AT 2016-06-01

USER root
RUN apt-get -qqy update; apt-get install -qqy sudo
RUN echo "jenkins ALL=NOPASSWD: ALL" >> /etc/sudoers
RUN wget http://get.docker.com/builds/Linux/x86_64/docker-latest.
    tgz
RUN tar -xvzf docker-latest.tgz
RUN mv docker/* /usr/bin/

USER jenkins
RUN /usr/local/bin/install-plugins.sh junit git git-client ssh-
    slaves greenballs chucknorris ws-cleanup
```

We see that our `Dockerfile` inherits from the `jenkins` image. The `jenkins` image is the official Jenkins image maintained by their community on the Docker Hub. The `Dockerfile` 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.

We've first set the `USER` to `root`, installed the `sudo` package and allowed the `jenkins` user to make use of `sudo`. We then installed the Docker binary. We'll use this to connect to our Docker host and run containers for our builds.

Next we switch back to the `jenkins` user. This user is the default for the `jenkins` image and is required for containers launched from the image to run Jenkins correctly. We then use a `RUN` instruction to execute the `install-plugins.sh` command to install a list of Jenkins plugins we're going to use.

Next, let's create a directory, `/var/jenkins_home`, to hold our Jenkin's configuration. This means every time we restart Jenkins we won't lose our configuration.

TIP Another approach would be to use Docker data volumes, which we'll discuss further in Chapter 6.

Listing 5.68: Create directory for Jenkins

```
$ sudo mkdir -p /var/jenkins_home
$ cd /var/jenkins_home
$ sudo chown -R 1000 /var/jenkins_home
```

TIP If you're running this example on OS X you might need to create the directory at `/private/var/jenkins_home`.

We also set the ownership of the `jenkins_home` directory to `1000`, which is the UID of the `jenkins` user inside the image we're about to build. This will allow Jenkins to write into this directory and store our Jenkins configuration.

Now that we have our `Dockerfile` and our Jenkins home directory, let's build a new image using the `docker build` command.

Listing 5.69: Building our Docker-Jenkins image

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

We've called our new image, somewhat unoriginally, `jamtur01/jenkins`. We can now create a container from this image using the `docker run` command.

Listing 5.70: Running our Docker-Jenkins image

```
$ sudo docker run -d -p 8080:8080 -p 50000:50000 \
-v /var/jenkins_home:/var/jenkins_home \
-v /var/run/docker.sock:/var/run/docker.sock \
--name jenkins \
jamtur01/jenkins
cc130210491ee959a287f04b5e4c46340bbcb6a46971de15d3899699b7718656
```

We can see that we've used the `-p` flag to publish 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've also bound port `50000` on port `50000` which will be used by the Jenkins build API.

Next, we bind two volumes using the `-v` flag. The first mounts our `/var/jenkins_home` directory into the container at `/var/jenkins_home`. This will contain Jenkin's configuration data and allow us to perpetuate its state across container launches.

The second volume mounts `/var/run/docker.sock`, the socket for Docker's daemon into the Docker container. This will allow us to run Docker containers from inside our Jenkins container.

WARNING This is a security risk. By binding the Docker socket inside the Jenkins container you give the container access to the underlying Docker host. This is not overly secure. I recommend you only do this if you are comfortable that the Jenkins container, any other containers on that Docker host are at a comparable security level.

We see that our new container, `jenkins`, has been started. Let's check out its logs.

Listing 5.71: Checking the Docker Jenkins container logs

```
$ sudo docker logs jenkins
Running from: /usr/share/jenkins/jenkins.war
webroot: EnvVars.masterEnvVars.get("JENKINS_HOME")
Aug 04, 2016 3:11:50 AM org.eclipse.jetty.util.log.JavaUtilLog
   info
INFO: Logging initialized @1760ms
Aug 04, 2016 3:11:51 AM winstone.Logger logInternal
INFO: Beginning extraction from war file

. . .

*************************************************************
*************************************************************
*************************************************************

Jenkins initial setup is required. An admin user has been created
    and a password generated.
Please use the following password to proceed to installation:

e9eef9d4a4e44741b0368877a9efb17c

This may also be found at: /var/jenkins_home/secrets/
   initialAdminPassword

*************************************************************
*************************************************************
*************************************************************

. . .

INFO: Jenkins is fully up and running
```

You can keep checking the logs, or run `docker logs` with the `-f` flag, until you see a message similar to:

Listing 5.72: Checking that is Jenkins up and running

```
INFO: Jenkins is fully up and running
```

Take note of the initial admin password, in our case:

`e9eef9d4a4e44741b0368877a9efb17c`

This is also stored in a file in the `jenkins_home` directory at:

`/var/jenkins_home/secrets/initialAdminPassword`

Finally, our Jenkins server should now be available in your browser on port `8080`, as we see here:

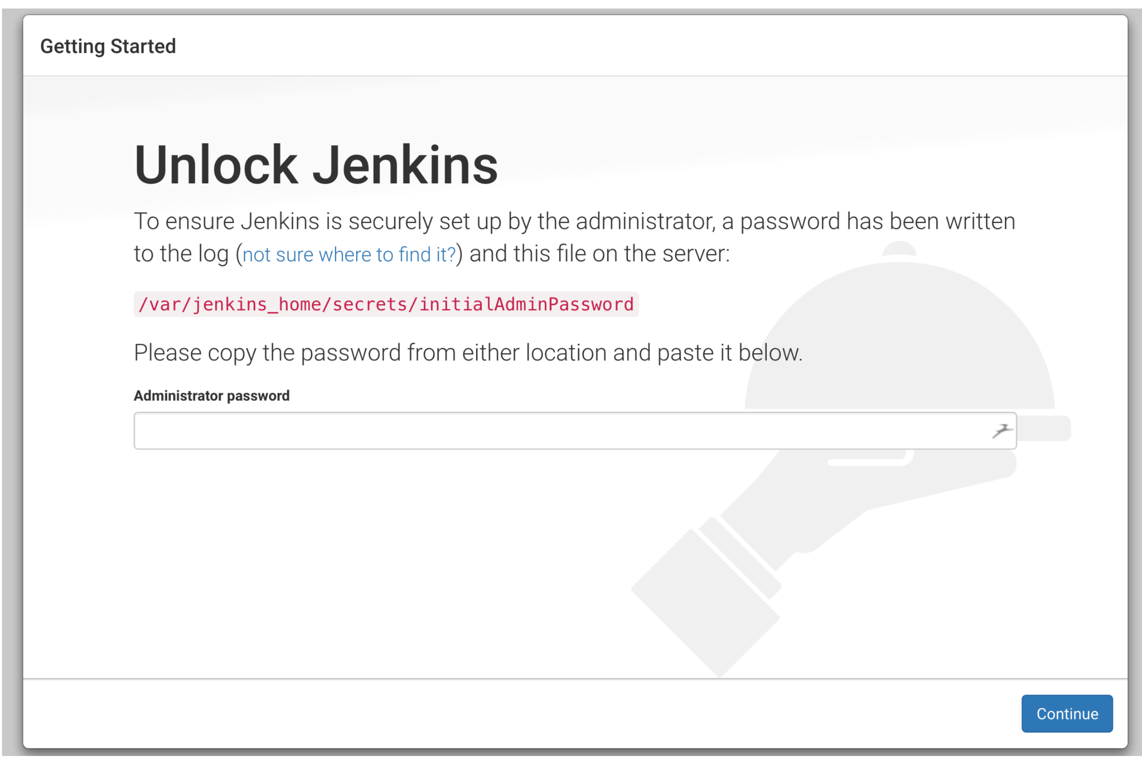


Figure 5.3: Browsing the Jenkins server.

Put in the admin password generated during installation and click the Continue button.

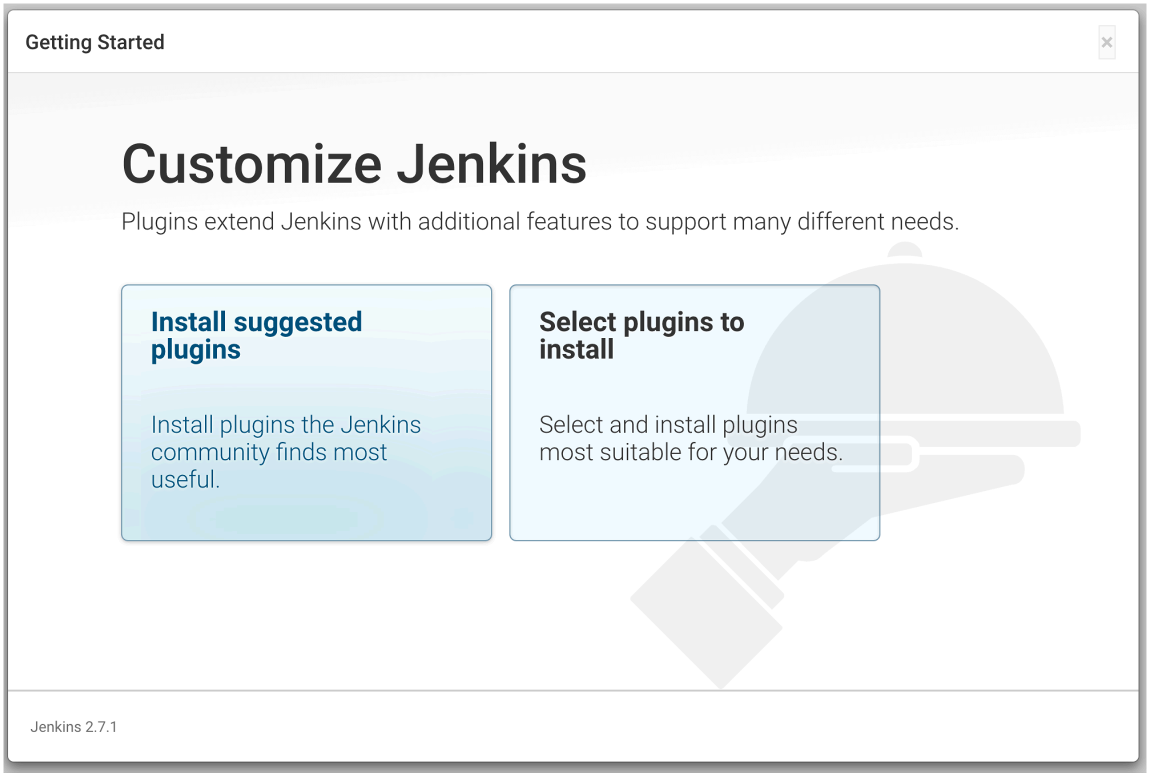


Figure 5.4: The Getting Started workflow

This will initiate the Jenkins Getting Started workflow. You can follow it or cancel it by clicking the X in the top right of the dialogue.

If you cancel the Getting Started dialogue you'll also skip creating any users. To log into Jenkins again we would use a user name of admin and our initial admin password.

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.

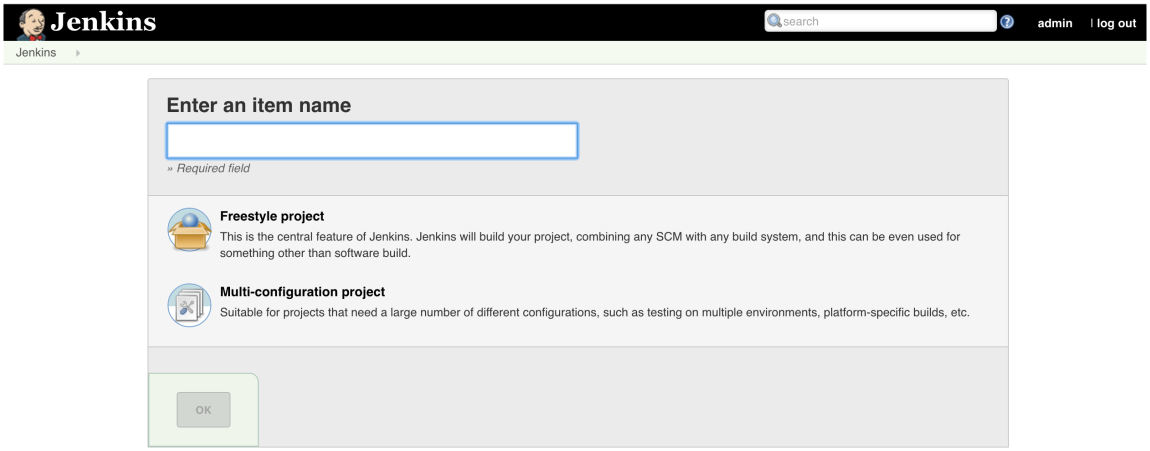


Figure 5.5: Creating a new Jenkins job.

Let's name our new job `Docker_test_job`, select a job type of `Freestyle 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, tick the `Use Custom workspace` radio button, and specify `/var/jenkins_home/jobs/${JOB_NAME}/workspace` as the `Directory`. This is the workspace in which our Jenkins job is going to run. It's also stored in our Jenkins home directory to ensure we maintain state across builds.

Under `Source Code Management`, select `Git` and specify the following test repository: `https://github.com/turnbullpress/docker-jenkins-sample.git`. This is a simple repository containing some Ruby-based RSpec tests.

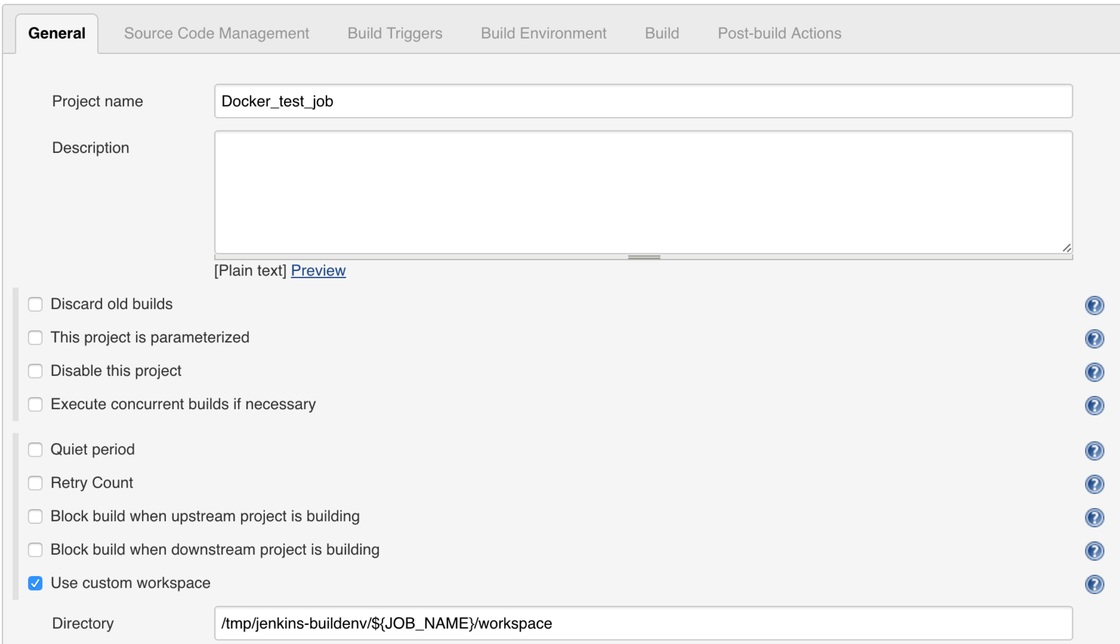


Figure 5.6: Jenkins job details part 1.

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 5.73: The Docker shell script for Jenkins jobs

```
# Build the image to be used for this job.
IMAGE=$(sudo docker build . | tail -1 | awk '{ print $NF }')

# Build the directory to be mounted into Docker.
MNT="$WORKSPACE/.."

# Execute the build inside Docker.
CONTAINER=$(sudo 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.
sudo docker attach $CONTAINER

# Get its exit code as soon as the container stops.
RC=$(sudo docker wait $CONTAINER)

# Delete the container we've just used.
sudo 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 5.74: The Docker test job Dockerfile

```
FROM ubuntu:18.04
LABEL maintainer="james@example.com"
ENV REFRESHED_AT 2016-06-01
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.

Here 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 new environment variable called `$MNT` using the `$WORKSPACE` variable. This is a variable, created by Jenkins, holding the workspace directory we defined earlier in our job. This is where our Git repository containing the code we want to test is going to be checked out to, and it is this directory we're going to mount into our Docker container. We can then execute our tests from this checkout.

Next we create a container from our image and run the tests. Inside this container, we've mounted our workspace via a volume to the `/opt/project` directory. When the container runs, we're executing a command that changes into this directory tree and executes the `rake spec` command, which actually runs our RSpec tests.

Now we've got a started container 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`

Whilst the container is running, 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. This will echo the test output into our Jenkins job. Finally, 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.

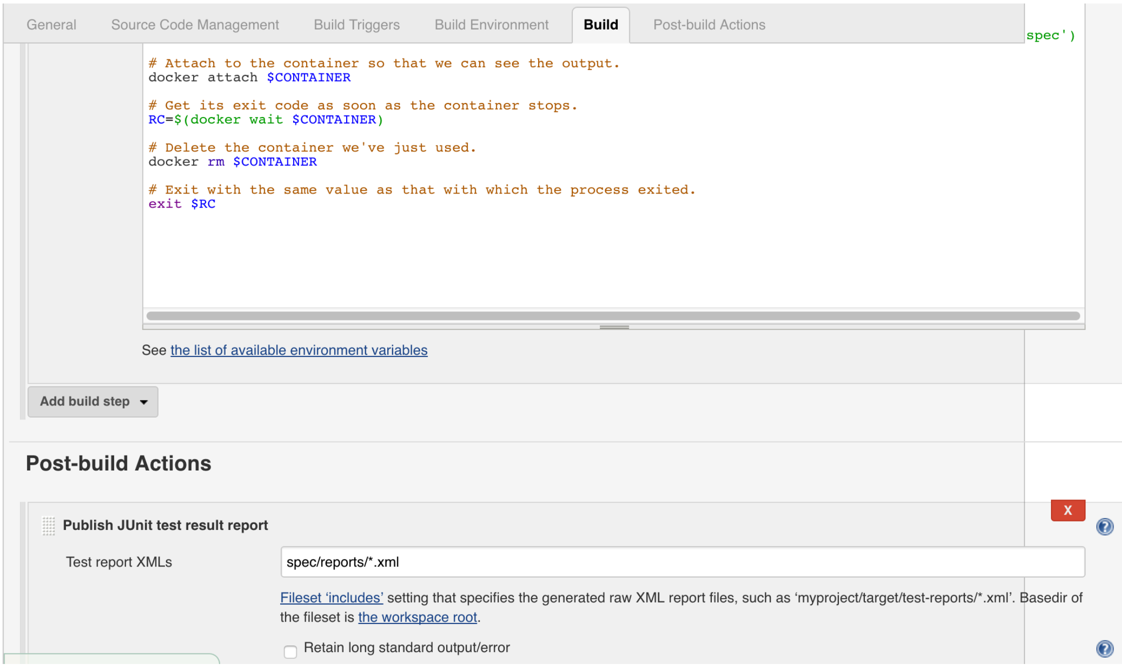


Figure 5.7: Jenkins job details part 2.

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.

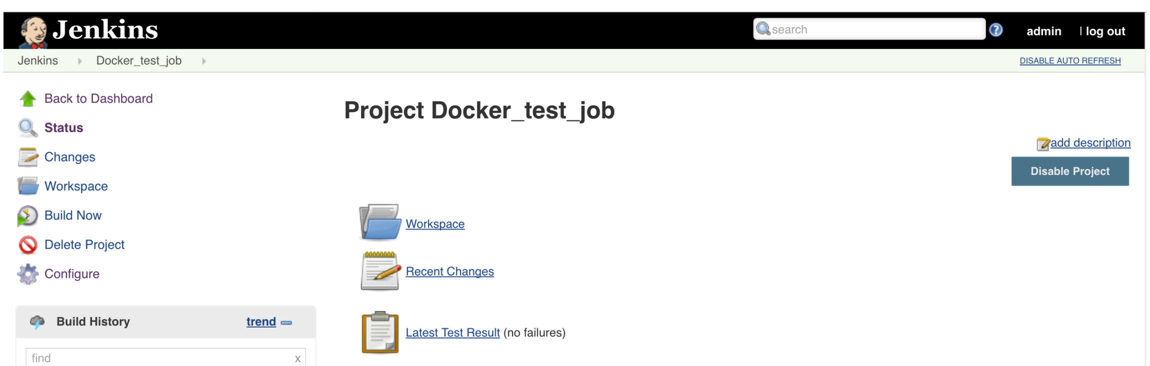


Figure 5.8: Running the Jenkins job.

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.

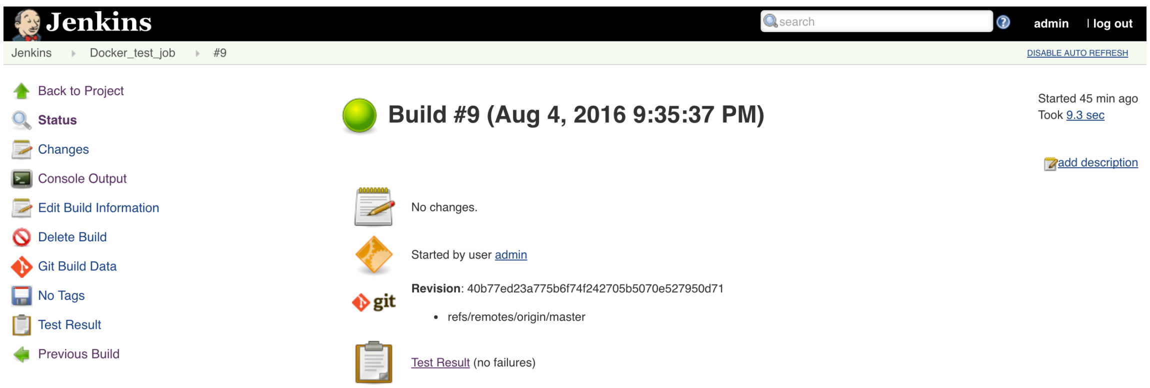


Figure 5.9: The Jenkins job details.

We can click on `Console Output` to see the commands that have been executed as part of the job.

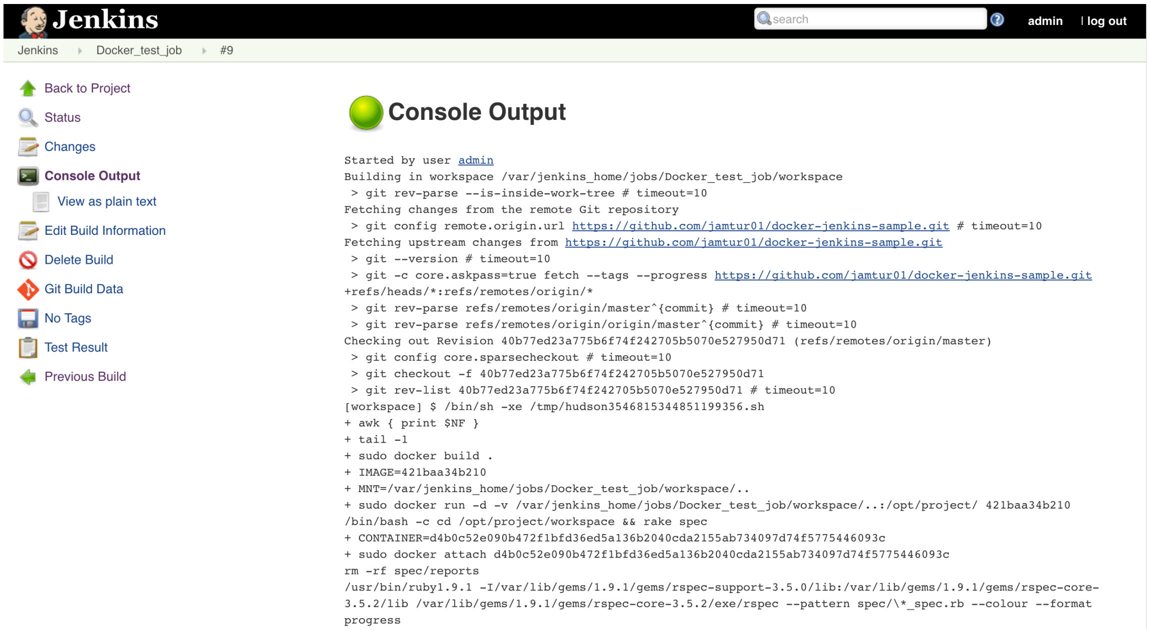


Figure 5.10: The Jenkins job console output.

We 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 Item` link from the Jenkins console. We're going to name our new job `Docker_matrix_job`, select `Multi-configuration project`, and click `OK`.

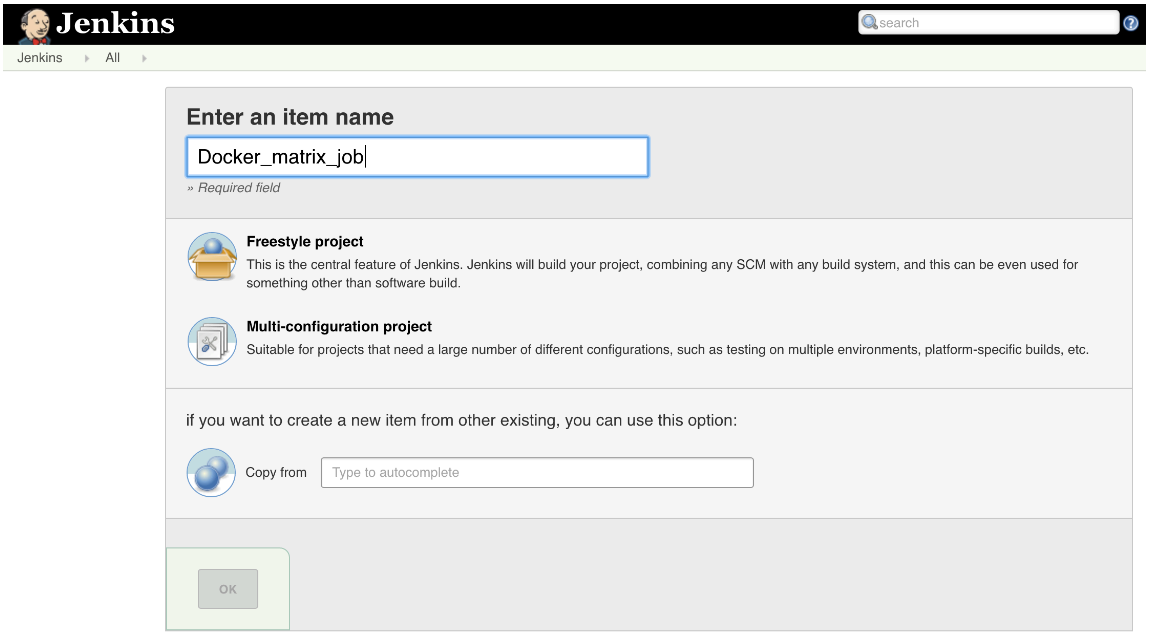


Figure 5.11: Creating a multi-configuration job.

We'll see a screen that is 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/turnbullpress/docker-jenkins-sample.git`.

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.

Then, in the `Build Environment` section, we click `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.

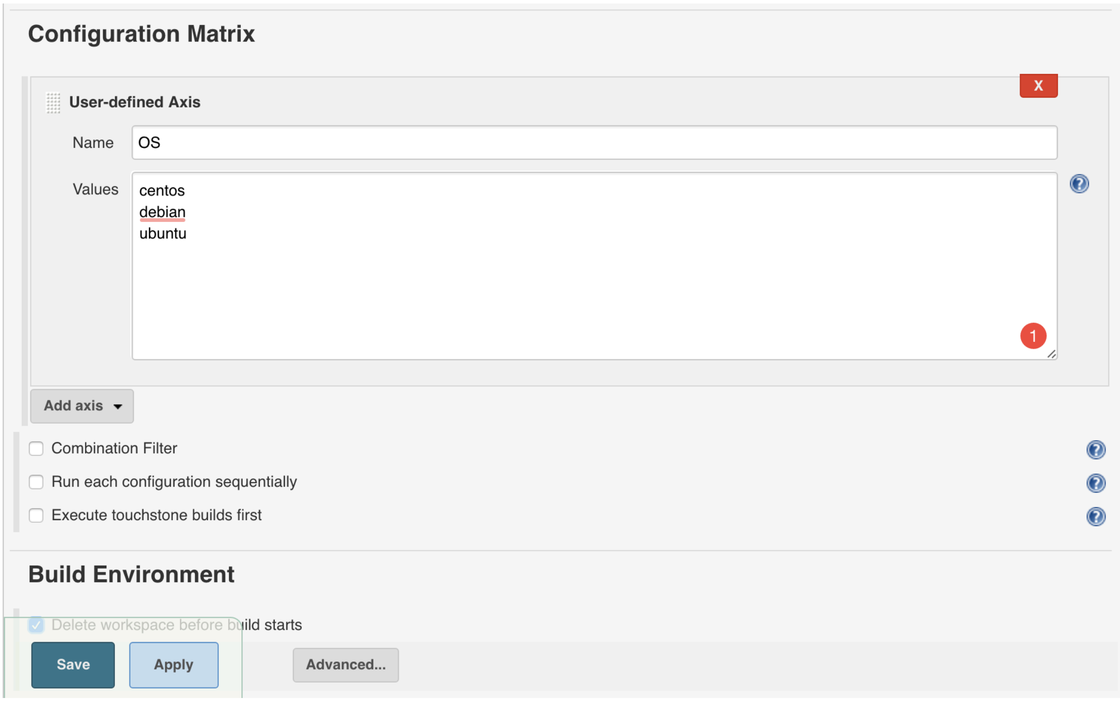


Figure 5.12: Configuring a multi-configuration job Part 2.

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 5.75: Jenkins multi-configuration shell step

```
# Build the image to be used for this run.
cd $OS; IMAGE=$(sudo docker build . | tail -1 | awk '{ print $NF
    }')

# Build the directory to be mounted into Docker.
MNT="$WORKSPACE/.."

# Execute the build inside Docker.
CONTAINER=$(sudo docker run -d -v "$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
    .
sudo docker attach $CONTAINER

# As soon as the process exits, get its return value.
RC=$(sudo docker wait $CONTAINER)

# Delete the container we've just used.
sudo docker rm $CONTAINER

# Exit with the same value as that with which the process exited.
exit $RC
```

We see that this script has a modification: we're changing into directories named for each operating system for which we're executing a job. 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 5.76: Our CentOS-based Dockerfile

```
FROM centos:latest
LABEL maintainer="james@example.com"
ENV REFRESHED_AT 2016-06-01
RUN yum -y install ruby rubygems rubygem-rake
RUN gem install --no-rdoc --no-ri rspec ci_reporter_rspec
```

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.

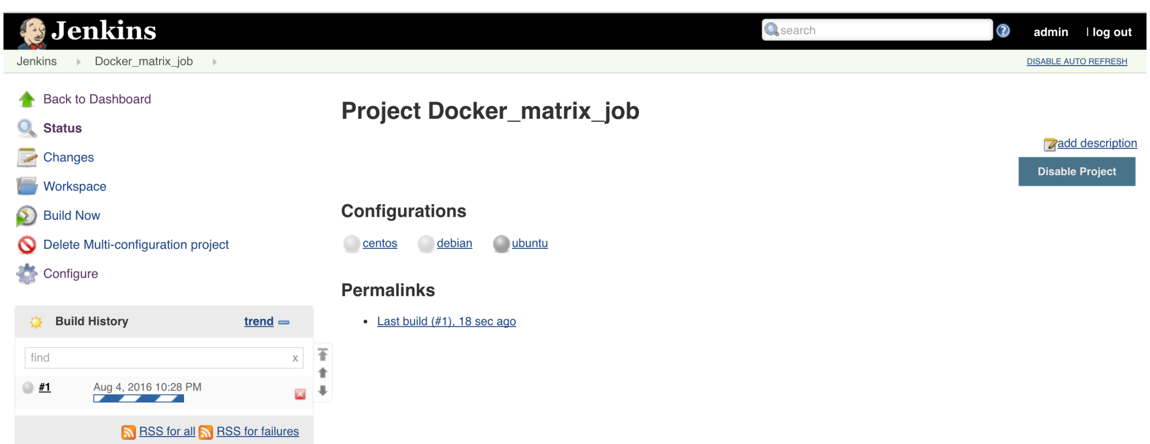


Figure 5.13: Our Jenkins multi-configuration job

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 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.

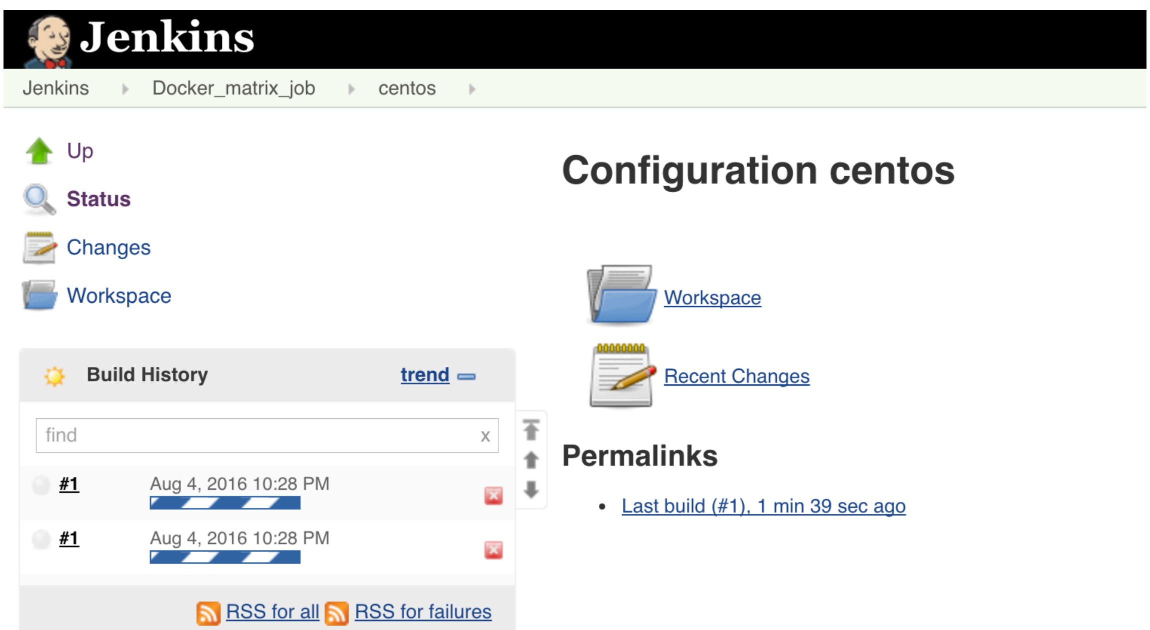


Figure 5.14: The centos sub-job.

We 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`.

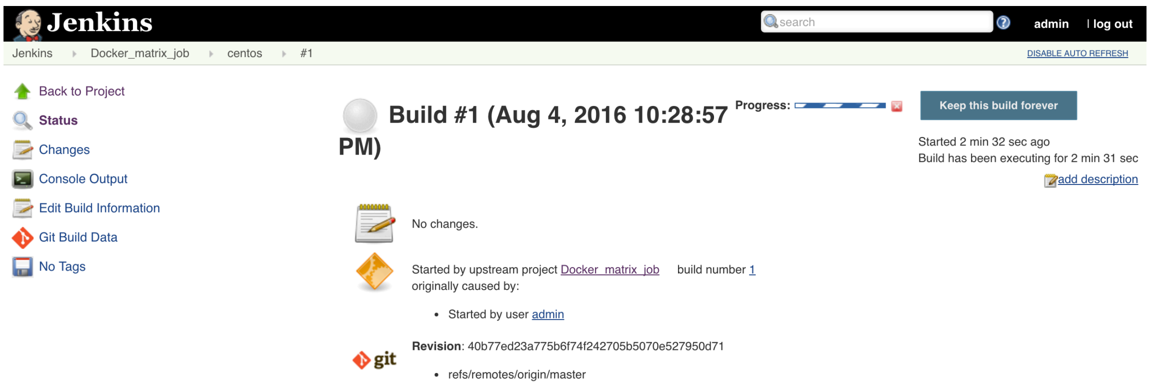


Figure 5.15: The centos sub-job details.

Here we see some more details of the executed `centos` job. We 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.

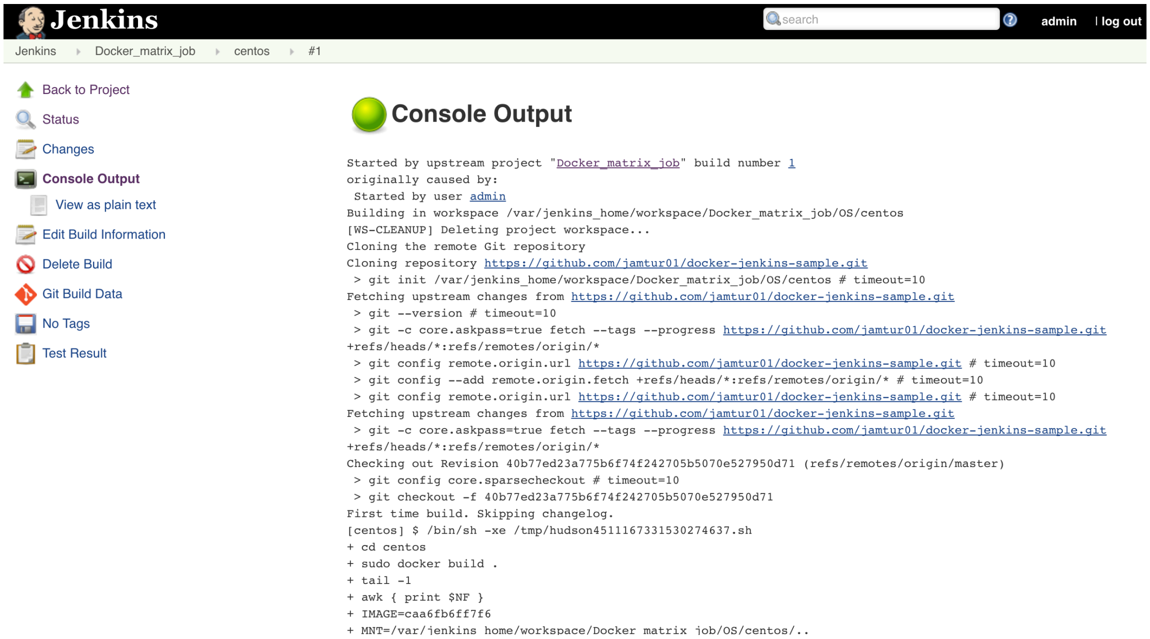


Figure 5.16: The centos sub-job console output.

We 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 simple, but powerful example of a multi-platform testing job for an application.

Summary of our multi-configuration Jenkins

These examples show 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 networking 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 simple, but you will see how we 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:18.04
LABEL maintainer="james@example.com"
ENV REFRESHED_AT 2016-06-01

RUN apt-get -yqq update
RUN apt-get -yqq install ruby ruby-dev build-essential nodejs
RUN gem install jekyll -v 2.5.3

VOLUME /data
VOLUME /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 16.04 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 will now build an image from which we will 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:18.04
 ---> 99ec81b80c55
Step 1 : LABEL maintainer="james@example.com"
. . .
Step 7 : ENTRYPOINT [ "jekyll", "build" "--destination=/var/www/
   html" ]
 ---> Running in 542e2de2029d
 ---> 79009691f408
Removing intermediate container 542e2de2029d
Successfully built 79009691f408
```

We see that we've built a new image with an ID of `79009691f408` named `jamtur01/jekyll` that is our new Jekyll image. We 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:18.04
LABEL maintainer="james@example.com"
ENV REFRESHED_AT 2016-06-01

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/apachectl" ]
CMD ["-D", "FOREGROUND"]
```

This final image is again based on Ubuntu 16.04 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 will now build an image from which we will 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:18.04
 ---> 99ec81b80c55
Step 1 : LABEL maintainer="james@example.com"
 ---> Using cache
 ---> c444e8ee0058
. . .
Step 11 : CMD ["-D", "FOREGROUND"]
 ---> Running in 7aa5c127b41e
 ---> fc8e9135212d
Removing intermediate container 7aa5c127b41e
Successfully built fc8e9135212d
```

We see that we've built a new image with an ID of `fc8e9135212d` named `jamtur01/apache` that is our new Apache image. We 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/turnbullpress/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: none
            Source: /data
       Destination: /var/www/html
      Generating...
                    done.
 Auto-regeneration: disabled. Use --watch to enable.
```

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 even when 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 "{{ range .Mounts }}{{.}}{{end}}" james_blog
```

TIP In Docker 1.9 volumes have been expanded to also support third-party storage systems like Ceph, Flocker and EMC via plugins. You can read about them in the volume plugins documentation and the docker volume create command documentation.

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.

NOTE Even if you delete the last container that uses a volume, the volume will still persist.

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.

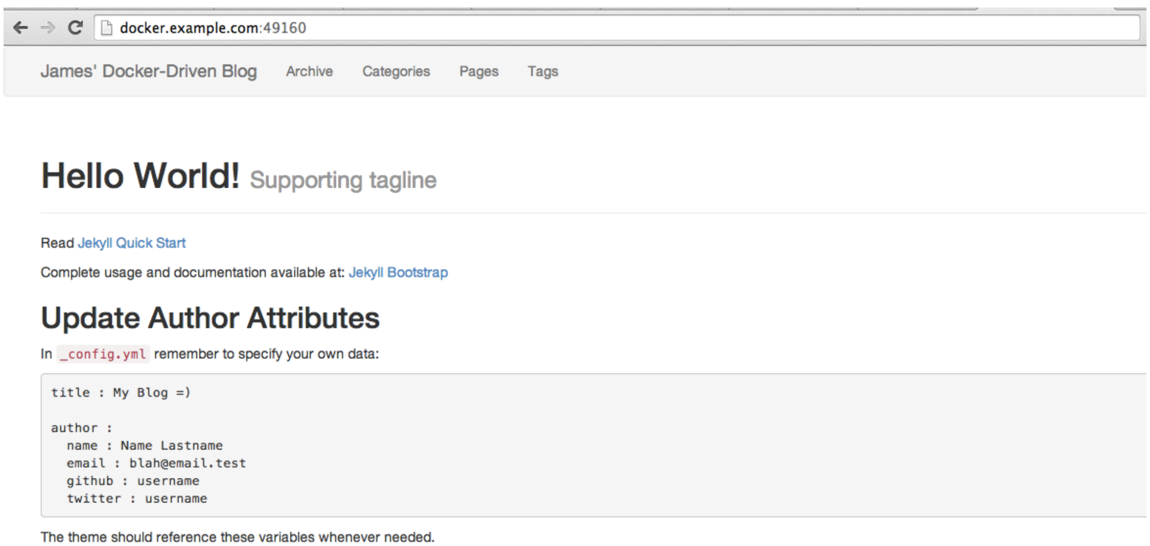


Figure 6.1: Our Jekyll website.

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 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.

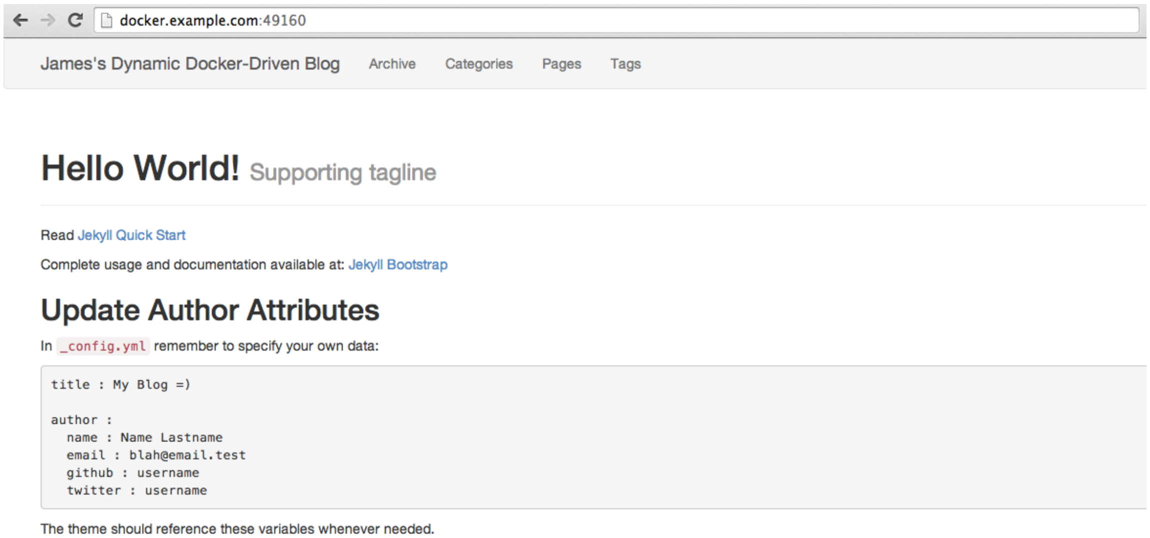


Figure 6.2: Our updated Jekyll website.

This all happened without having to update or restart our Apache container, because 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 prettily 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 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 a 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/jeykll` 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 own variant of 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:18.04
LABEL maintainer="james@example.com"
ENV REFRESHED_AT 2016-06-01

RUN apt-get -yqq update
RUN apt-get -yqq install wget

VOLUME [ "/var/lib/tomcat8/webapps/" ]
WORKDIR /var/lib/tomcat8/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/tomcat8/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 https://tomcat.apache.org/tomcat-7.0-doc/appdev/sample/.

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 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/tomcat8/webapps/` directory.

Our WAR file is 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 "{{ range .Mounts }}{{.}}{{end}}" sample
{c20a0567145677ed46938825f285402566e821462632e1842e82bc51b47fe4dc
    /var/lib/docker/volumes/
   c20a0567145677ed46938825f285402566e821462632e1842e82bc51b47fe4dc
   /_data /var/lib/tomcat8/webapps local  true}
```

We then list this directory.

Listing 6.23: Listing the volume directory

```
$ sudo ls -l /var/lib/docker/volumes/
   c20a0567145677ed46938825f285402566e821462632e1842e82bc51b47fe4dc
   /_data
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 tomcat8
$ cd tomcat8
$ touch Dockerfile
```

Now let's populate our `Dockerfile`.

Listing 6.25: Our Tomcat 7 Application server

```
FROM ubuntu:18.04
LABEL maintainer="james@example.com"
ENV REFRESHED_AT 2016-06-01

RUN apt-get -yqq update
RUN apt-get -yqq install tomcat8 default-jdk

ENV CATALINA_HOME /usr/share/tomcat8
ENV CATALINA_BASE /var/lib/tomcat8
ENV CATALINA_PID /var/run/tomcat8.pid
ENV CATALINA_SH /usr/share/tomcat8/bin/catalina.sh
ENV CATALINA_TMPDIR /tmp/tomcat8-tomcat8-tmp

RUN mkdir -p $CATALINA_TMPDIR

VOLUME [ "/var/lib/tomcat8/webapps/" ]

EXPOSE 8080

ENTRYPOINT [ "/usr/share/tomcat8/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/tomcat8/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/tomcat8 .
```

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/tomcat8
```

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/tomcat8/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.

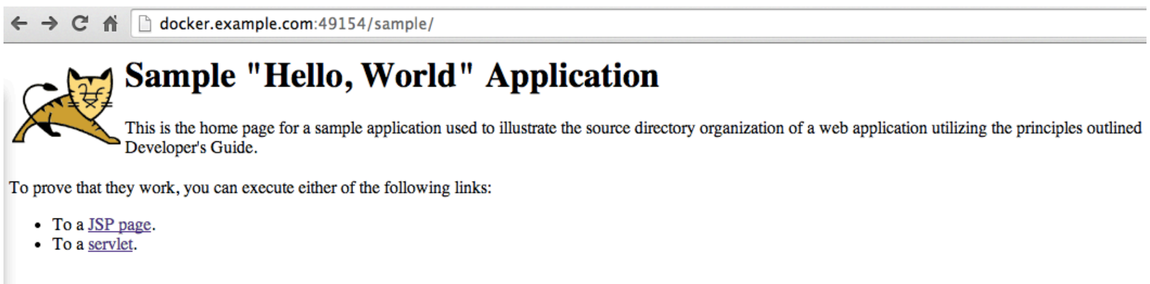


Figure 6.3: Our Tomcat sample application.

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 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.

Listing 6.29: Installing Ruby

```
$ sudo apt-get -qqy install ruby make ruby-dev build-essential
```

We 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-0.0.6
```

This will install the TProv application and some supporting gems.

We 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
   ]
== Sinatra/1.4.5 has taken the stage on 4567 for development with
    backup from WEBrick
[2014-06-21 16:17:24] INFO  WEBrick::HTTPServer#start: pid=14209
   port=4567
```

This command has launched our application; now we can browse to the TProv website on port 4567 of the Docker host.

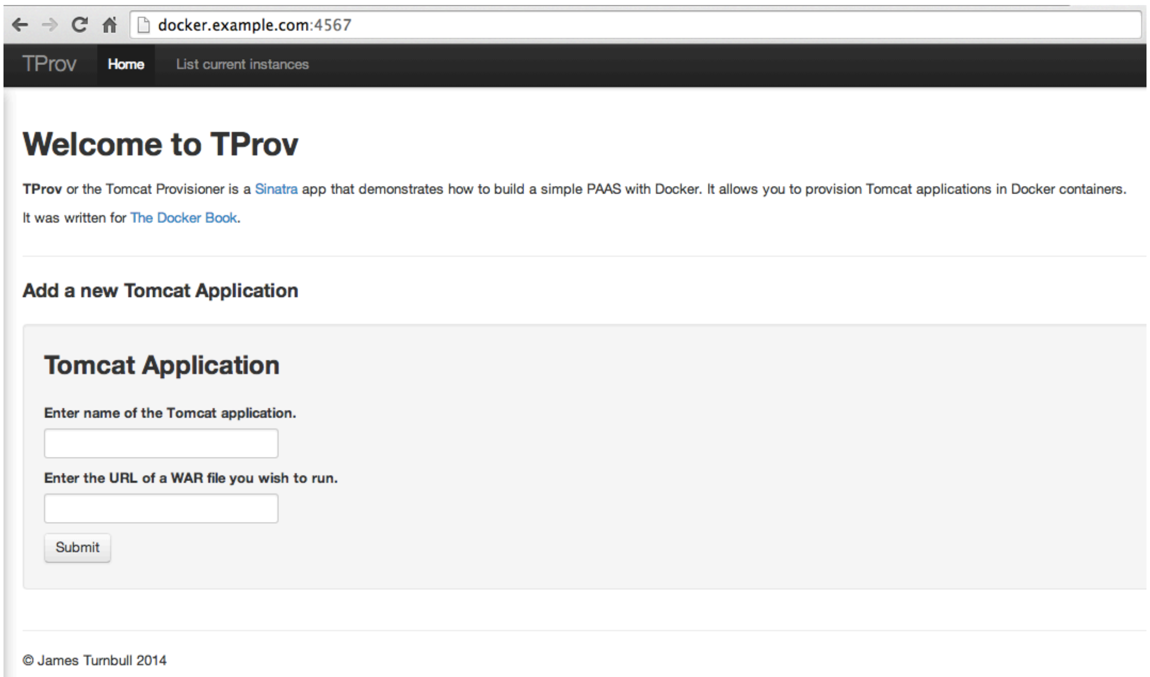


Figure 6.4: Our TProv web application.

We specify a Tomcat application name and the URL to a Tomcat WAR file. Let's download a sample calendar application from:

https://storage.googleapis.com/google-code-archive-downloads/v2/code.google.com/gwt-examples/Calendar.war

And call it Calendar.

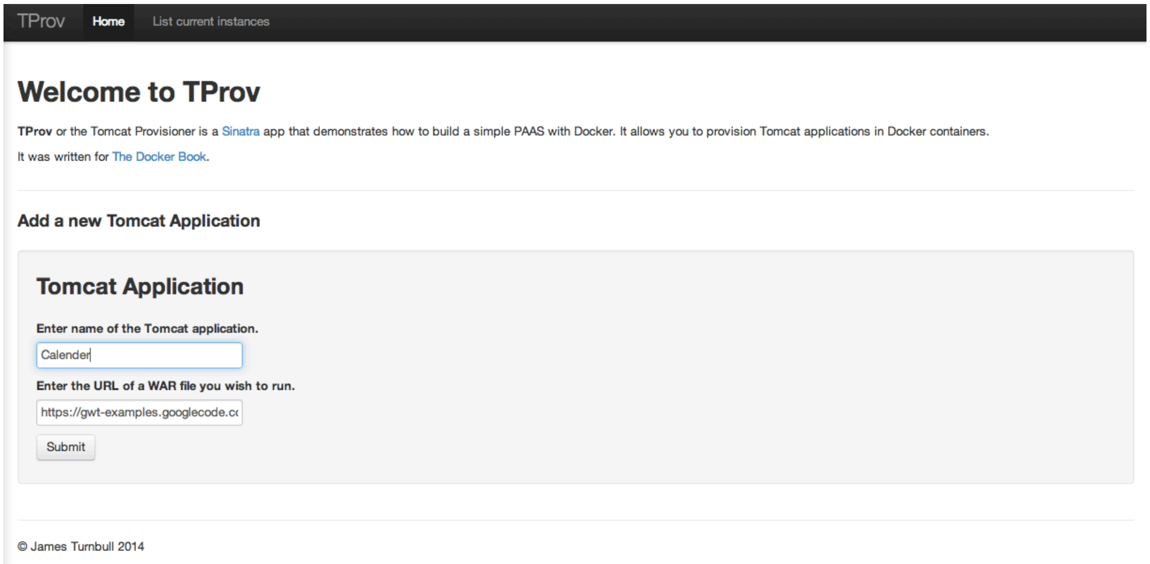


Figure 6.5: Downloading a sample application.

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 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.

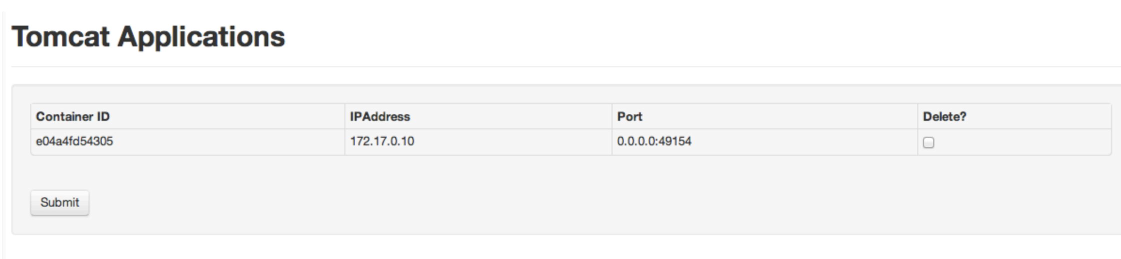


Figure 6.6: Listing the Tomcat instances.

Using this information, we 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's simple code, 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 networking 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.

[1] Really 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 -p nodejs/nodeapp
$ cd nodejs/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 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 on GitHub here.

Finally, we've changed back to the `nodejs` directory and now we populate our `Dockerfile`.

Listing 6.33: Our Node.js image

```
FROM ubuntu:18.04
LABEL maintainer="james@example.com"
ENV REFRESHED_AT 2016-06-01

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 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 on GitHub here.

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:18.04
LABEL maintainer="james@example.com"
ENV REFRESHED_AT 2017-06-01

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 `VOLUME`s (`/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
LABEL maintainer="james@example.com"
ENV REFRESHED_AT 2016-06-01

ENTRYPOINT [ "redis-server", "--protected-mode no", "--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
LABEL maintainer="james@example.com"
ENV REFRESHED_AT 2016-06-01

ENTRYPOINT [ "redis-server", "--protected-mode no", "--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 build our own Redis replication environment. Let's start by creating a network to hold our Express application. We'll call it `express`.

Listing 6.45: Creating the express network

```
$ sudo docker network create express
dfe9fe7ee5c9bfa035b7cf10266f29a701634442903ed9732dfdba2b509680c2
```

Now let's run the Redis primary container inside this network.

Listing 6.46: Running the Redis primary container

```
$ sudo docker run -d -h redis_primary \
--net express --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 specified the `--name` flag to ensure that our container's name is `redis_primary` and we've specified the `--net` flag to run the container in the `express` network. We're going to use this network for our container connectivity, as we'll see shortly.

Let's see what the `docker logs` command can tell us about our Redis primary container.

Listing 6.47: 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 use the `/var/log/redis` volume we created earlier. Let's use this volume and read some log files now.

Listing 6.48: Reading our Redis primary logs

```
$ sudo docker run -ti --rm --volumes-from redis_primary \
ubuntu cat /var/log/redis/redis-server.log
. . .
1:M 05 Aug 15:22:21.697 # Server started, Redis version 3.2.9
. . .
1:M 05 Aug 15:22:21.698 * 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 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 6.49: Running our first Redis replica container

```
$ sudo docker run -d -h redis_replica1 \
--name redis_replica1 \
--net express \
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) of `redis_replica1` and a name (with `--name`) of `redis_replica1`. We've also used the `--net` flag to run our Redis replica container inside the `express` network.

Let's check this new container's logs.

Listing 6.50: Reading our Redis replica logs

```
$ sudo docker run -ti --rm --volumes-from redis_replica1 \
ubuntu cat /var/log/redis/redis-replica.log
...
1:S 05 Aug 15:23:57.733 # Server started, Redis version 3.2.9
1:S 05 Aug 15:23:57.733 * The server is now ready to accept
    connections on port 6379
1:S 05 Aug 15:23:57.733 * Connecting to MASTER redis_primary:6379
1:S 05 Aug 15:23:57.743 * MASTER <-> SLAVE sync started
1:S 05 Aug 15:23:57.743 * Non blocking connect for SYNC fired the
     event.
1:S 05 Aug 15:23:57.743 * Master replied to PING, replication can
     continue...
1:S 05 Aug 15:23:57.744 * Partial resynchronization not possible
    (no cached master)
1:S 05 Aug 15:23:57.751 * Full resync from master: 692
    b4d19978a2d6add881944a079ab8b8dae6653:1
1:S 05 Aug 15:23:57.841 * MASTER <-> SLAVE sync: receiving 18
    bytes from master
1:S 05 Aug 15:23:57.841 * MASTER <-> SLAVE sync: Flushing old
    data
1:S 05 Aug 15:23:57.841 * MASTER <-> SLAVE sync: Loading DB in
    memory
1:S 05 Aug 15:23:57.841 * 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 `/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 6.51: Running our second Redis replica container

```
$ sudo docker run -d -h redis_replica2 \
--name redis_replica2 \
--net express \
jamtur01/redis_replica
72267cd74c412c7b168d87bba70f3aaa3b96d17d6e9682663095a492bc260357
```

Let's see a sampling of the logs from our new container.

Listing 6.52: Our Redis replica2 logs

```
$ sudo docker run -ti --rm --volumes-from redis_replica2 ubuntu \
cat /var/log/redis/redis-replica.log
. . .
1:S 05 Aug 15:27:38.355 # Server started, Redis version 3.2.9
1:S 05 Aug 15:27:38.355 * The server is now ready to accept
   connections on port 6379
1:S 05 Aug 15:27:38.355 * Connecting to MASTER redis_primary:6379
1:S 05 Aug 15:27:38.366 * MASTER <-> SLAVE sync started
1:S 05 Aug 15:27:38.366 * Non blocking connect for SYNC fired the
    event.
1:S 05 Aug 15:27:38.366 * Master replied to PING, replication can
    continue...
1:S 05 Aug 15:27:38.366 * Partial resynchronization not possible
   (no cached master)
1:S 05 Aug 15:27:38.372 * Full resync from master: 692
   b4d19978a2d6add881944a079ab8b8dae6653:309
1:S 05 Aug 15:27:38.465 * MASTER <-> SLAVE sync: receiving 18
   bytes from master
1:S 05 Aug 15:27:38.465 * MASTER <-> SLAVE sync: Flushing old
   data
1:S 05 Aug 15:27:38.465 * MASTER <-> SLAVE sync: Loading DB in
   memory
1:S 05 Aug 15:27:38.465 * MASTER <-> SLAVE sync: Finished with
   success
```

And again, we're off and away replicating!

Creating our Node container

Now that we've got our Redis cluster running, we launch a container for our Node.js application.

Listing 6.53: Running our Node.js container

```
$ sudo docker run -d \
--name nodeapp -p 3000:3000 \
--net express \
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 run our new `nodeapp` container in the `express` network.

We use the `docker logs` command to see what's going on in our `nodeapp` container.

Listing 6.54: The nodeapp console log

```
$ sudo docker logs nodeapp
Listening on port 3000
```

Here we 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.

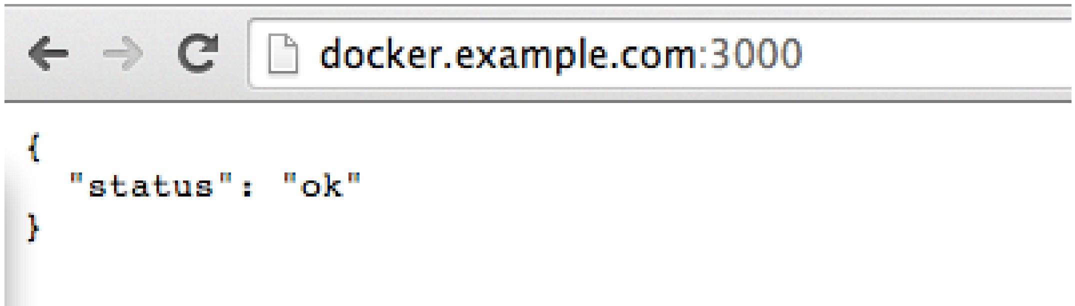


Figure 6.7: Our Node application.

We see that our simple Node application returns an `OK` status.

Listing 6.55: 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 6.56: Creating our Logstash Dockerfile

```
$ mkdir logstash
$ cd logstash
$ touch Dockerfile
```

Now let's populate our `Dockerfile`.

Listing 6.57: Our Logstash image

```
FROM ubuntu:18.04
LABEL maintainer="james@example.com"
ENV REFRESHED_AT 2016-06-01
RUN apt-get -qq update
RUN apt-get -qq install wget gnupg2 openjdk-8-jdk
RUN wget -qO - https://artifacts.elastic.co/GPG-KEY-elasticsearch
    | apt-key add -
RUN echo "deb https://artifacts.elastic.co/packages/5.x/apt
   stable main" | tee -a /etc/apt/sources.list.d/elastic-5.x.li
st
RUN apt-get -qq update
RUN apt-get -qq install logstash
WORKDIR /usr/share/logstash
ADD logstash.conf /usr/share/logstash/
ENTRYPOINT [ "bin/logstash" ]
CMD [ "-f", "logstash.conf", "--config.reload.automatic" ]
```

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 create this file in the `logstash` directory. Add a file called `logstash.conf` and populate it like so:

Listing 6.58: 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 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 6.59: Building our Logstash image

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

Now that we've built our Logstash image, we launch a container from it.

Listing 6.60: 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 browse to our web application again and refresh it to generate an event. We should see that event reflected in the `logstash` container's `docker logs` output.

Listing 6.61: A Node event in Logstash

```
{
       "message" => "::ffff:198.179.69.250 - - [Fri, 05 Aug 2016
    16:39:25 GMT] \"GET / HTTP/1.1\" 200 20 \"-\" \"Mozilla/5.0 (
    Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (KHTML,
     like Gecko) Chrome/51.0.2704.103 Safari/537.36\"",
      "@version" => "1",
    "@timestamp" => "2016-08-05T16:39:25.945Z",
          "host" => "1bbc26b1ed7d",
          "path" => "/var/log/nodeapp/nodeapp.log",
          "type" => "syslog"
}
```

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 networking to connect our application together and Docker volumes to help manage a variety of aspects of our application. We can 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 will use volumes or networking to perform a lot of the same actions. For example, if our service is managed via a network interface, we expose that on a container; if our service is managed through a Unix socket, we expose that with a volume. If we need to send a signal to a Docker container, we 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 the `docker exec` command

NOTE The `docker exec` command introduced in Docker 1.3 replaces the previous tool, `nsenter`.

Listing 6.63: Running docker exec

```
$ sudo docker exec -ti nodeapp /bin/bash
```

This will launch an interactive Bash shell inside our `nodeapp` 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 build multi-container services and manage those stacks. We've combined features like Docker networking 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 Docker Compose, Docker Swarm 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 its own book but we've focused on a few useful tools that give you some insight into what you can achieve when orchestrating containers. They provide some useful building blocks upon which you can grow your Docker-enabled environment.

In this chapter we will focus on three areas:

- Simple container orchestration. Here we'll look at Docker Compose. Docker Compose (previously 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'll introduce 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.
- Orchestration and clustering of Docker. Here we're looking at Swarm. Swarm is open source, licensed with the Apache 2.0 license. It's written in Go and developed by the Docker Inc team. As of Docker 1.12 the Docker Engine now has a Swarm-mode built in and we'll be covering that later in this chapter.

TIP We'll also talk about many of the other orchestration tools available to you later in this chapter.

Docker Compose

Now let's get familiar with Docker Compose. With Docker Compose, we define a set of containers to boot up, and their runtime properties, all defined in a YAML file. Docker Compose 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 Docker Compose and then using it to build a simple, multi-container application stack.

Installing Docker Compose

We start by installing Docker Compose. Docker Compose is currently available for Linux, Windows, and OS X. It can be installed directly as a binary and via Docker for Mac or Windows.

To install Docker Compose on Linux we can grab the Docker Compose binary from GitHub and make it executable. Like Docker, Docker Compose is currently only supported on 64-bit Linux installations. We'll need the `curl` command available to do this.

Listing 7.1: Installing Docker Compose on Linux

```
$ sudo curl -L "https://github.com/docker/compose/releases/
    download/$(curl -sL https://api.github.com/repos/docker/
    compose/releases/latest | grep tag_name | cut -d'"' -f 4)/
    docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/
    docker-compose
$ sudo chmod +x /usr/local/bin/docker-compose
```

This will download the `docker-compose` binary from GitHub and install it into the `/usr/local/bin` directory. We've also used the `chmod` command to make the `docker-compose` binary executable so we can run it.

If we're on OS X Docker Compose comes bundled with Docker for Mac or we can install it like so:

Listing 7.2: Installing Docker Compose on OS X

```
$ sudo bash -c "curl -L https://github.com/docker/compose/
    releases/download/1.17.1/docker-compose-Darwin-x86_64 > /usr/
    local/bin/docker-compose"
$ sudo chmod +x /usr/local/bin/docker-compose
```

TIP Replace the 1.17.1 with the release number of the current Docker Compose release.

If we're on Windows Docker Compose comes bundled inside Docker for Windows.

Once you have installed the `docker-compose` binary you can test it's working using the `docker-compose` command with the `--version` flag:

Listing 7.3: Testing Docker Compose is working

```
$ docker-compose --version
docker-compose version 1.17.1, build f3628c7
```

NOTE If you're upgrading from a pre-1.3.0 release you'll need to migrate any existing container to the new 1.3.0 format using the `docker-compose migrate-to-labels` command.

Getting our sample application

To demonstrate how Compose works we're going to use a sample Python Flask application that combines two containers:

- An application container running our sample Python application.
- A container running the Redis database.

Let's start with building our sample application. Firstly, we create a directory and a `Dockerfile`.

Listing 7.4: Creating the composeapp directory

```
$ mkdir composeapp
$ cd composeapp
```

Here we've created a directory to hold our sample application, which we're calling `composeapp`.

Next, we need to add our application code. Let's create a file called `app.py` in the `composeapp` directory and add the following Python code to it.

Listing 7.5: The app.py file

```
from flask import Flask
from redis import Redis
import os

app = Flask(__name__)
redis = Redis(host="redis", 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 GitHub.

This simple Flask 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 7.6: The requirements.txt file

```
flask
redis
```

Now let's populate our Compose `Dockerfile`.

Listing 7.7: The composeapp Dockerfile

```
# Compose Sample application image
FROM python:2.7
LABEL maintainer="james@example.com"
ENV REFRESHED_AT 2016-06-01

ADD . /composeapp

WORKDIR /composeapp

RUN pip install -r requirements.txt
```

Our `Dockerfile` is 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 `/composeapp`. The `Dockerfile` then sets the working directory to `/composeapp` 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 7.8: Building the composeapp application

```
$ sudo docker build -t jamtur01/composeapp .
Sending build context to Docker daemon  16.9 kB
Sending build context to Docker daemon
Step 0 : FROM python:2.7
 ---> 1c8df2f0c10b
Step 1 : LABEL maintainer="james@example.com"
 ---> Using cache
 ---> aa564fe8be5a
Step 2 : ADD . /composeapp
 ---> c33aa147e19f
Removing intermediate container 0097bc79d37b
Step 3 : WORKDIR /composeapp
 ---> 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/composeapp` containing our sample application and its required dependencies. We can now use Compose 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 `docker-compose.yml` file

Now we've got our application image built we can configure Compose to create both the services we require. With Compose, we define a set of services (in the form of Docker containers) to launch. 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 `docker-compose up` command. Compose launches the containers, executes the appropriate runtime configuration, and multiplexes the log output together for us.

Let's create a `docker-compose.yml` file for our application inside our `composeapp` directory.

Listing 7.9: Creating the docker-compose.yml file

```
$ touch docker-compose.yml
```

Let's populate our `docker-compose.yml` file. The `docker-compose.yml` file is a YAML file that contains instructions for running one or more Docker containers. Let's look at the instructions for our example application.

Listing 7.10: The docker-compose.yml file

```
version: '3'
services:
  web:
    image: jamtur01/composeapp
    command: python app.py
    ports:
     - "5000:5000"
    volumes:
     - .:/composeapp
  redis:
    image: redis
```

Each service we wish to launch is specified as a YAML hash inside a hash called `services`. Here our two services are: `web` and `redis`.

TIP The `version` tag tells Docker Compose what configuration version of use. The Docker Compose API has evolved over the years and each change has been marked by incrementing the version.

For our `web` service we've specified some runtime options. Firstly, we've specified the `image` we're using: the `jamtur01/composeapp` image. Compose can also build Docker images. You can use the `build` instruction and provide the path to a `Dockerfile` to have Compose build an image and then create services from it.

Listing 7.11: An example of the build instruction

```
web:
  build: /home/james/composeapp
. . .
```

This `build` instruction would build a Docker image from a `Dockerfile` found in the `/home/james/composeapp` 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 `/composeapp` as a volume.

If we were executing the same configuration on the command line using `docker run` we'd do it like so:

Listing 7.12: The docker run equivalent command

```
$ sudo docker run -d -p 5000:5000 -v .:/composeapp \
--name jamtur01/composeapp 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 `docker-compose.yml` file in the Docker Compose documentation.

Running Compose

Once we've specified our services in `docker-compose.yml` we use the `docker-compose up` command to execute them both.

Listing 7.13: Running docker-compose up with our sample application

```
$ cd composeapp
$ sudo docker-compose up
Creating network "composeapp_default" with the default driver
Recreating composeapp_web_1 ...
Recreating composeapp_web_1
Recreating composeapp_redis_1 ...
Recreating composeapp_web_1 ... done
Attaching to composeapp_redis_1, composeapp_web_1
web_1    |  * Running on http://0.0.0.0:5000/ (Press CTRL+C to
   quit)
. . .
```

TIP You must be inside the directory with the `docker-compose.yml` file in order to execute most Compose commands.

Compose has created two new services: `composeapp_redis_1` and `composeapp_web_1`. So where did these names come from? Well, to ensure our services are unique, Compose has prefixed and suffixed the names specified in the `docker-compose.yml` file with the directory and a number respectively.

Compose 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 7.14: Compose service log output

```
redis_1  | 1:M 05 Aug 17:49:17.839 * The server is now ready to
    accept connections on port 6379
```

The services (and Compose) are being run interactively. That means if you use `Ctrl-C` or the like to cancel Compose then it'll stop the running services. We could also run Compose with `-d` flag to run our services daemonized (similar to the `docker run -d` flag).

Listing 7.15: Running Compose daemonized

```
$ sudo docker-compose 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`.

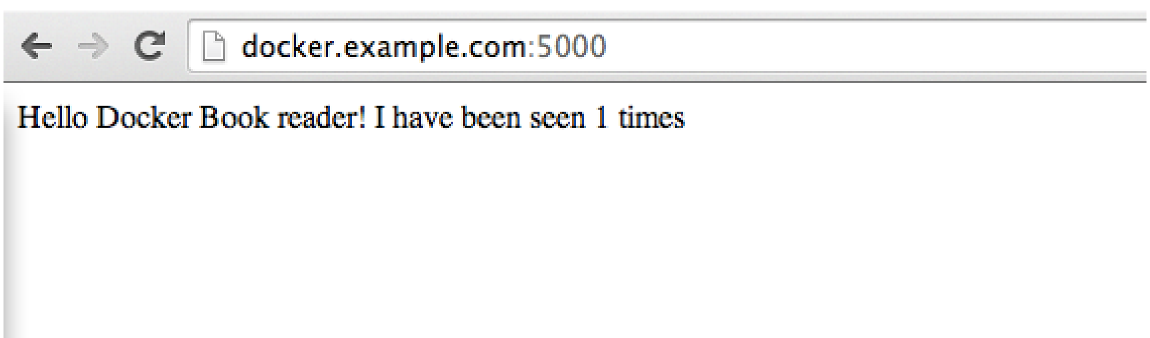


Figure 7.1: Sample Compose application.

We 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 Compose.

TIP By default, Compose 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 Compose

Now let's explore some of Compose'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 `composeapp` directory and then re-run the `docker-compose up` command, this time with the `-d` flag.

Listing 7.16: Restarting Compose as daemonized

```
$ sudo docker-compose up -d
Starting composeapp_web_1 ...
Starting composeapp_redis_1 ...
Starting composeapp_redis_1
Starting composeapp_web_1 ... done
$ . . .
```

We see that Compose has recreated our services, launched them and returned to the command line.

Our Compose-managed services are now running daemonized on the host. Let's look at them now using the `docker-compose ps` command; a close cousin of the `docker ps` command.

TIP You can get help on Compose commands by running `docker-compose help` and the command you wish to get help on, for example `docker-compose help ps`.

The `docker-compose ps` command lists all of the currently running services from our local `docker-compose.yml` file.

Listing 7.17: Running the docker-compose ps command

```
$ cd composeapp
$ sudo docker-compose ps
Name                 Command                      State Ports
-----------------------------------------------------------------
   -
composeapp_redis_1 docker-entrypoint.sh redis  Up    6379/tcp
composeapp_web_1   python app.py               Up    0.0.0.0:5000
   ->5000/tcp
```

This shows some basic information about our running Compose 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 `docker-compose logs` command to show us the log events from our services.

Listing 7.18: Showing a Compose services logs

```
$ sudo docker-compose logs
docker-compose logs
Attaching to composeapp_redis_1, composeapp_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 `docker-compose stop` command.

Listing 7.19: Stopping running services

```
$ sudo docker-compose stop
Stopping composeapp_web_1...
Stopping composeapp_redis_1...
```

This will stop both services. If the services don't stop you can use the `docker-compose kill` command to force kill the services.

We can verify this with the `docker-compose ps` command again.

Listing 7.20: Verifying our Compose services have been stopped

```
$ sudo docker-compose ps
Name                 Command          State    Ports
-------------------------------------------------
composeapp_redis_1 redis-server    Exit 0
composeapp_web_1   python app.py   Exit 0
```

If you've stopped services using `docker-compose stop` or `docker-compose kill` you can also restart them again with the `docker-compose start` command. This is much like using the `docker start` command and will restart these services.

Finally, we can remove services using the `docker-compose rm` command.

Listing 7.21: Removing Compose services

```
$ sudo docker-compose rm
Going to remove composeapp_redis_1, composeapp_web_1
Are you sure? [yN] y
Removing composeapp_redis_1...
Removing composeapp_web_1...
```

You'll be prompted to confirm you wish to remove the services and then both services will be deleted. The `docker-compose ps` command will now show no running or stopped services.

Listing 7.22: Showing no Compose services

```
$ sudo docker-compose ps
Name   Command   State   Ports
------------------------------
```

Compose 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. It's especially a great tool for building local development stacks. This, however, just scratches the surface of what you can do with Compose. There are some more examples using Rails, Django and Wordpress on the Compose website that introduce some more advanced concepts.

TIP You can see a full command line reference in the Docker Compose Reference documentation.

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 centers or geographic 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-oriented 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 an 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 in their documentation.

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 7.23: Creating a Consul Dockerfile directory

```
$ mkdir consul
$ cd consul
$ touch Dockerfile
```

Now let's look at the `Dockerfile` for our Consul image.

Listing 7.24: The Consul Dockerfile

```
FROM ubuntu:18.04
LABEL maintainer="james@example.com"
ENV REFRESHED_AT 2014-08-01

RUN apt-get -qqy update
RUN apt-get -qqy install curl unzip

ADD https://releases.hashicorp.com/consul/0.6.4/consul_0.6.4
   _linux_amd64.zip /tmp/consul.zip
RUN cd /usr/sbin; unzip /tmp/consul.zip; chmod +x /usr/sbin/
   consul; rm /tmp/consul.zip

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 16.04 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 then add a configuration file for Consul, `consul.json`, to the `/config` directory. Let's create and look at that file now.

Listing 7.25: The consul.json configuration file

```
{
  "data_dir": "/data",
  "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 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 in the Consul documentation.

Back in our `Dockerfile` we've used 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
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 in the Consul documentation.

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 7.26: Building our Consul image

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

NOTE You can get our Consul `Dockerfile` and configuration file on GitHub. If you don't want to use a home grown image there is also an officially sanctioned Consul image on the Docker Hub.

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 7.27: Running a local Consul node

```
$ sudo docker run -p 8500:8500 -p 53:53/udp \
-h node1 jamtur01/consul -server -bootstrap
==> WARNING: Bootstrap mode enabled! Do not enable unless
    necessary
==> Starting Consul agent...
==> Starting Consul agent RPC...
==> Consul agent running!
   Node name: 'node1'
   Datacenter: 'dc1'
   Server: true (bootstrap: true)
   Client Addr: 0.0.0.0 (HTTP: 8500, HTTPS: -1, DNS: 53, RPC:
    8400)
   Cluster Addr: 172.17.0.8 (LAN: 8301, WAN: 8302)
   Gossip encrypt: false, RPC-TLS: false, TLS-Incoming: false
   Atlas: <disabled>

==> Log data will now stream in as it occurs:

. . .

2016/08/05 17:59:38 [INFO] consul: cluster leadership acquired
2016/08/05 17:59:38 [INFO] consul: New leader elected: node1
2016/08/05 17:59:38 [INFO] raft: Disabling EnableSingleNode (
    bootstrap)
2016/08/05 17:59:38 [INFO] consul: member 'node1' joined, marking
     health alive
2016/08/05 17:59:40 [INFO] agent: Synced service 'consul'
```

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 container 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 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`.

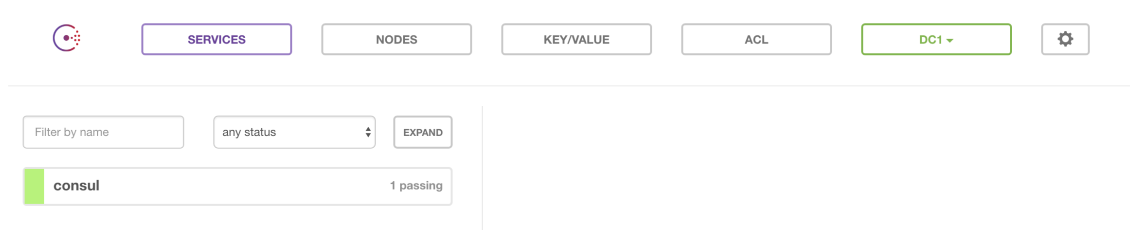


Figure 7.2: The Consul web interface.

Running a Consul cluster in Docker

As Consul is distributed we'd normally create three (or more) hosts to run in separate data centers, 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 new hosts each with a Docker daemon to run Consul. We will create three new Ubuntu 16.04 hosts: `larry`, `curly`, and `moe`. On each host we'll install a Docker daemon. We'll also pull down the `jamtur01/consul` image.

TIP Create the hosts using whatever means you run up new hosts and to install Docker you can use the installation instructions in Chapter 2.

Listing 7.28: 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 to which we're going to bind each Consul agent.

Let's grab that now on `larry` and assign it to an environment variable, `$PUBLIC_IP`.

Listing 7.29: Getting public IP on larry

```
larry$ PUBLIC_IP="$(ifconfig eth0 | awk -F ' *|:' '/inet addr/{
   print $4}')"
larry$ echo $PUBLIC_IP
162.243.167.159
```

And then create the same `$PUBLIC_IP` variable on `curly` and `moe` too.

Listing 7.30: Assigning public IP on curly and moe

```
curly$ PUBLIC_IP="$(ifconfig eth0 | awk -F ' *|:' '/inet addr/{
    print $4}')"
curly$ echo $PUBLIC_IP
162.243.170.66
moe$ PUBLIC_IP="$(ifconfig eth0 | awk -F ' *|:' '/inet addr/{
    print $4}')"
moe$ echo $PUBLIC_IP
159.203.191.16
```

We 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	162.243.167.159
curly	162.243.170.66
moe	159.203.191.16

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 `162.243.167.159` to `curly` and `moe` as the environment variable, `$JOIN_IP`.

Listing 7.31: Adding the cluster IP address

```
curly$ JOIN_IP=162.243.167.159
moe$ JOIN_IP=162.243.167.159
```

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 7.32: 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'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 7.33: 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 7.34: Starting bootstrap Consul node

```
larry$ sudo docker logs larry_agent
==> WARNING: Expect Mode enabled, expecting 3 servers
==> 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, HTTPS: -1, DNS: 53, RPC:
    8400)
      Cluster Addr: 162.243.167.159 (LAN: 8301, WAN: 8302)
    Gossip encrypt: false, RPC-TLS: false, TLS-Incoming: false
             Atlas: <disabled>

==> Log data will now stream in as it occurs:

. . .

2016/08/06 12:35:11 [INFO] serf: EventMemberJoin: larry.dc1
   162.243.167.159
2016/08/06 12:35:11 [INFO] consul: adding LAN server larry (Addr:
    162.243.167.159:8300) (DC: dc1)
2016/08/06 12:35:11 [INFO] consul: adding WAN server larry.dc1 (
   Addr: 162.243.167.159:8300) (DC: dc1)
2016/08/06 12:35:11 [ERR] agent: failed to sync remote state: No
   cluster leader
2016/08/06 12:35:12 [WARN] raft: EnableSingleNode disabled, and
   no known peers. Aborting election.
```

We see that the agent on `larry` is started but because we don't have any more nodes yet no election has taken place. We know this from the only error returned.

Listing 7.35: 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 7.36: 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 similar to our bootstrapped node on `larry` with the exception of the command we're passing to the Consul agent.

Listing 7.37: 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 7.38: 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, HTTPS: -1, DNS: 53, RPC:
    8400)
      Cluster Addr: 162.243.170.66 (LAN: 8301, WAN: 8302)
    Gossip encrypt: false, RPC-TLS: false, TLS-Incoming: false
             Atlas: <disabled>

==> Log data will now stream in as it occurs:

. . .

2016/08/06 12:37:17 [INFO] consul: adding LAN server curly (Addr:
    162.243.170.66:8300) (DC: dc1)
2016/08/06 12:37:17 [INFO] consul: adding WAN server curly.dc1 (
   Addr: 162.243.170.66:8300) (DC: dc1)
2016/08/06 12:37:17 [INFO] agent: (LAN) joining:
   [162.243.167.159]
2016/08/06 12:37:17 [INFO] serf: EventMemberJoin: larry
   162.243.167.159
2016/08/06 12:37:17 [INFO] agent: (LAN) joined: 1 Err: <nil>
2016/08/06 12:37:17 [ERR] agent: failed to sync remote state: No
   cluster leader
2016/08/06 12:37:17 [INFO] consul: adding LAN server larry (Addr:
    162.243.167.159:8300) (DC: dc1)
2016/08/06 12:37:18 [WARN] raft: EnableSingleNode disabled, and
   no known peers. Aborting election.
```

We see `curly` has joined `larry`, indeed on `larry` we should see something like the following:

Listing 7.39: Curly joining Larry

```
2016/08/06 12:37:17 [INFO] serf: EventMemberJoin: curly
   162.243.170.66
2016/08/06 12:37:17 [INFO] consul: adding LAN server curly (Addr:
    162.243.170.66:8300) (DC: dc1)
```

But we've still not got a quorum in our cluster, remember we told `-bootstrap-expect` to expect 3 nodes. So let's start our final agent on `moe`.

Listing 7.40: Starting the agent on moe

```
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 7.41: 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, HTTPS: -1, DNS: 53, RPC:
    8400)
      Cluster Addr: 159.203.191.16 (LAN: 8301, WAN: 8302)
    Gossip encrypt: false, RPC-TLS: false, TLS-Incoming: false
             Atlas: <disabled>

==> Log data will now stream in as it occurs:

. . .

2016/08/06 12:39:14 [ERR] agent: failed to sync remote state: No
   cluster leader
2016/08/06 12:39:15 [INFO] consul: New leader elected: larry
2016/08/06 12:39:16 [INFO] agent: Synced service 'consul'
```

We 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 see the result of this final agent joining in the Consul logs on `larry` too.

Listing 7.42: Consul leader election on larry

```
2016/08/06 12:39:14 [INFO] consul: Attempting bootstrap with
   nodes: [162.243.170.66:8300 159.203.191.16:8300
   162.243.167.159:8300]
2016/08/06 12:39:15 [WARN] raft: Heartbeat timeout reached,
   starting election
2016/08/06 12:39:15 [INFO] raft: Node at 162.243.170.66:8300 [
   Candidate] entering Candidate state
2016/08/06 12:39:15 [WARN] raft: Remote peer 159.203.191.16:8300
   does not have local node 162.243.167.159:8300 as a peer
2016/08/06 12:39:15 [INFO] raft: Election won. Tally: 2
2016/08/06 12:39:15 [INFO] raft: Node at 162.243.170.66:8300 [
   Leader] entering Leader state
2016/08/06 12:39:15 [INFO] consul: cluster leadership acquired
2016/08/06 12:39:15 [INFO] consul: New leader elected: larry
2016/08/06 12:39:15 [INFO] raft: pipelining replication to peer
   159.203.191.16:8300
2016/08/06 12:39:15 [INFO] consul: member 'larry' joined, marking
    health alive
2016/08/06 12:39:15 [INFO] consul: member 'curly' joined, marking
    health alive
2016/08/06 12:39:15 [INFO] raft: pipelining replication to peer
   162.243.170.66:8300
2016/08/06 12:39:15 [INFO] consul: member 'moe' joined, marking
   health alive
```

We can also browse to the Consul web interface on `larry` on port `8500` and select the `Consul` service to see the current state

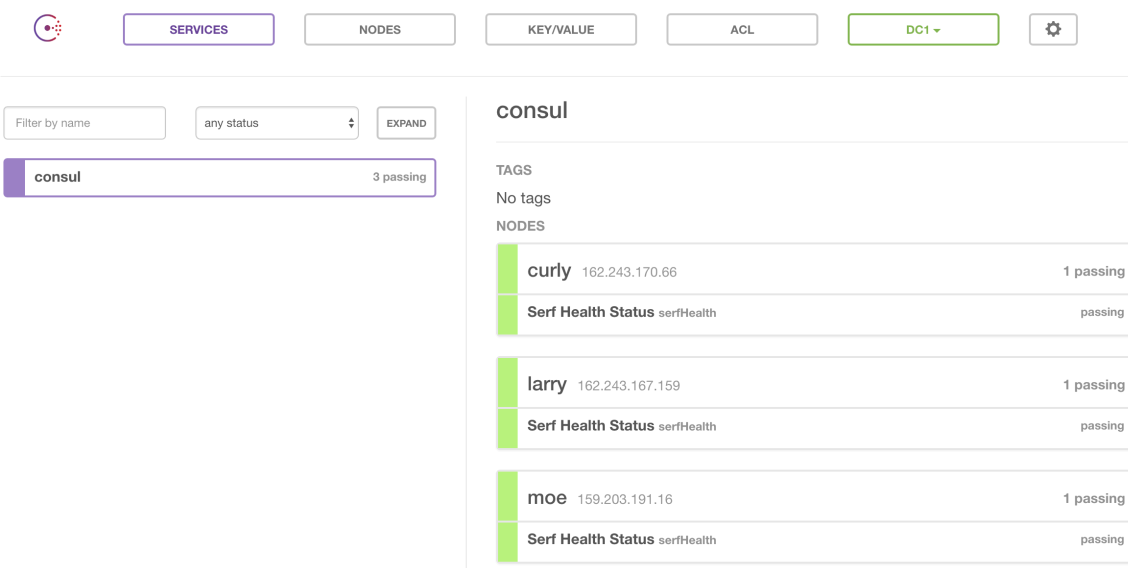


Figure 7.3: The Consul service in the web interface.

Finally, we can test the DNS is working using the `dig` command. We specify our local Docker bridge IP as the DNS server. That's the IP address of the Docker interface: `docker0`.

Listing 7.43: 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 172.17.0.1/16 scope global docker0
        valid_lft forever preferred_lft forever
     inet6 fe80::5484:7aff:fefe:9799/64 scope link
        valid_lft forever preferred_lft forever
```

We see the interface has an IP of `172.17.0.1`. We then use this with the `dig` command.

Listing 7.44: Testing the Consul DNS

```
larry$ dig @172.17.0.1 consul.service.consul
; <<>> DiG 9.10.3-P4-Ubuntu <<>> @172.17.0.1 consul.service.
    consul
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 42298
;; 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   162.243.170.66
consul.service.consul.  0   IN  A   159.203.191.16
consul.service.consul.  0   IN  A   162.243.167.159

;; Query time: 1 msec
;; SERVER: 172.17.0.1#53(172.17.0.1)
;; WHEN: Sat Aug 06 12:54:18 UTC 2016
;; 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 7.45: Querying another Consul service via DNS

```
larry$ dig @172.17.0.1 webservice.service.consul
```

Would return all DNS A records for the service `webservice`. We can also query individual nodes.

Listing 7.46: Querying another Consul service via DNS

```
larry$ dig @172.17.0.1 curly.node.consul +noall +answer

; <<>> DiG 9.10.3-P4-Ubuntu <<>> @172.17.0.1 curly.node.consul +
    noall +answer
; (1 server found)
;; global options: +cmd
curly.node.consul.  0   IN  A   162.243.170.66
```

TIP You can see more details on Consul's DNS interface in the Consul documentation.

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 7.47: 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 7.48: The distributed_app Dockerfile

```
FROM ubuntu:18.04
LABEL maintainer="james@example.com"
ENV REFRESHED_AT 2016-06-01

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 mkdir -p /opt/distributed_app
WORKDIR /opt/distributed_app

RUN uwsgi --build-plugin https://github.com/unbit/uwsgi-consul

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 7.49: 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 7.50: 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 7.51: The Consul plugin URL

```
url=http://%h.node.consul:8500...
```

Lastly, we've configured an `ENTRYPOINT` instruction to automatically run our web application workers.

Let's build our image now.

Listing 7.52: 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 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 7.53: 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 7.54: The distributed_client Dockerfile

```
FROM ubuntu:18.04
LABEL maintainer="james@example.com"
ENV REFRESHED_AT 2016-06-01

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 7.55: 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 7.56: 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 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 7.57: Starting distributed_app on larry

```
larry$ sudo docker run --dns=172.17.0.1 -h $HOSTNAME -d --name
   larry_distributed \
jamtur01/distributed_app
```

Here we've launched the `jamtur01/distributed_app` image and specified the `--dns` flag to add a DNS lookup from the Docker server, here represented by the `docker0` interface bridge IP address of `172.17.0.1`. As we've bound Consul's DNS lookup when we ran the Consul server this will allow the application to lookup nodes and services in Consul. You should replace this with the IP address of your own `docker0` interface.

We've also specified `-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 7.58: The distributed_app log output

```
larry$ sudo docker logs larry_distributed
*** Starting uWSGI 2.0.13.1 (64bit) on [Sat Aug  6 13:44:26 2016]
    ***
compiled with version: 5.4.0 20160609 on 06 August 2016 12:58:54
os: Linux-4.4.0-31-generic #50-Ubuntu SMP Wed Jul 13 00:07:12 UTC
    2016

. . .

Sat Aug  6 13:44:26 2016 - [consul] workers ready, let's register
    the service to the agent
spawned uWSGI mule 2 (pid: 13)
[consul] service distributed_app registered successfully
Sat Aug  6 13:44:27 2016 - [consul] workers ready, let's register
    the service to the agent
[consul] service distributed_app registered successfully
```

We see a subset of the logs here and 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.

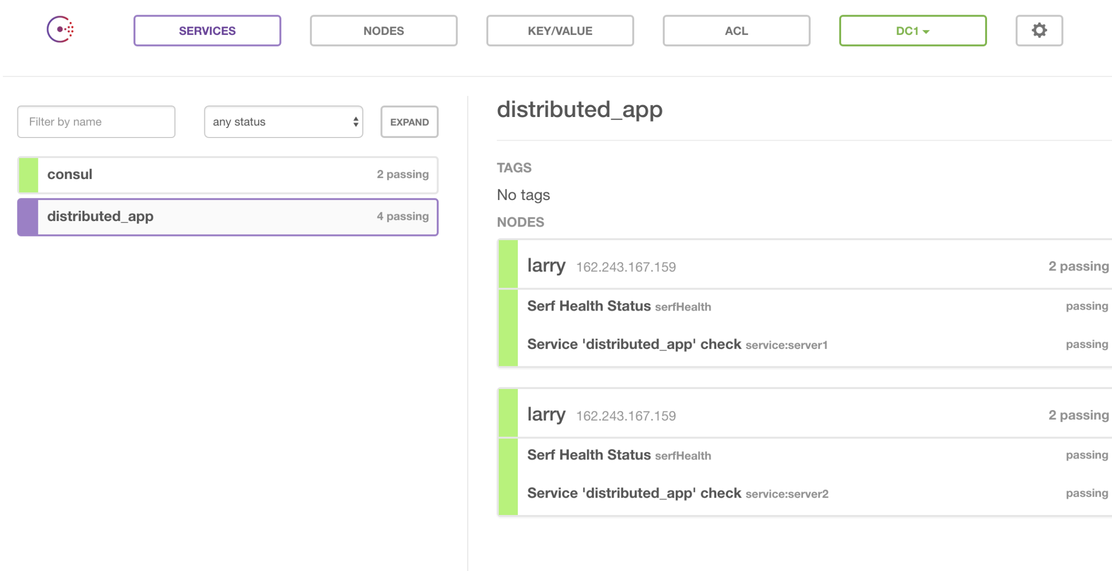


Figure 7.4: The distributed_app service in the Consul web interface.

Let's start some more web application workers on `curly` now.

Listing 7.59: Starting distributed_app on curly

```
curly$ sudo docker run --dns=172.17.0.1 -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.

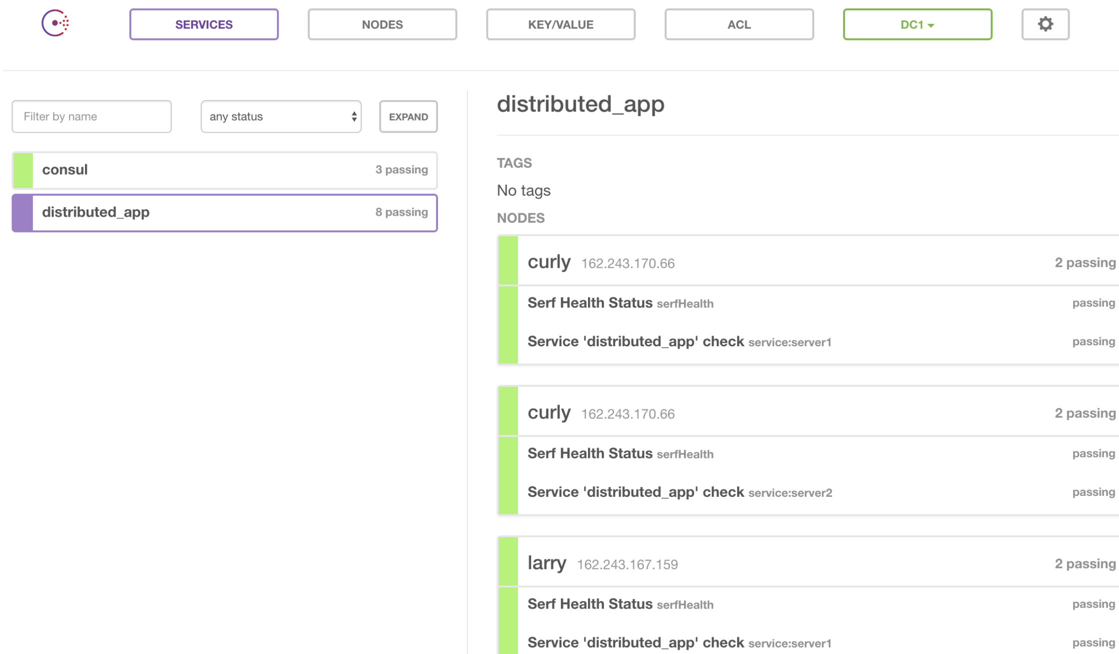


Figure 7.5: More distributed_app services in the Consul web interface.

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 7.60: Starting distributed_client on moe

```
moe$ sudo docker run -ti --dns=172.17.0.1 --name
   moe_distributed_client jamtur01/distributed_client
```

This time we've run the `jamtur01/distributed_client` image on `moe` and created an interactive container called `moe_distributed_client`. It should start emitting log output like so:

Listing 7.61: The distributed_client logs on moe

```
Application distributed_app with element server1 on port 2001
   found on node curly (162.243.170.66).
We can also resolve DNS - distributed_app resolves to
   162.243.167.159 and 162.243.170.66.
Application distributed_app with element server2 on port 2002
   found on node curly (162.243.170.66).
We can also resolve DNS - distributed_app resolves to
   162.243.167.159 and 162.243.170.66.
Application distributed_app with element server1 on port 2001
   found on node larry (162.243.167.159).
We can also resolve DNS - distributed_app resolves to
   162.243.170.66 and 162.243.167.159.
Application distributed_app with element server2 on port 2002
   found on node larry (162.243.167.159).
We can also resolve DNS - distributed_app resolves to
   162.243.167.159 and 162.243.170.66.
```

We 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, `162.243.167.159` and `162.243.170.66`.

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.

Docker Swarm

Docker Swarm is native clustering for Docker. It turns a pool of Docker hosts into a single virtual Docker host. Swarm has a simple architecture. It clusters together multiple Docker hosts and serves the standard Docker API on top of that cluster. This is incredibly powerful because it moves up the abstraction of Docker containers to the cluster level without you having to learn a new API. This makes integration with tools that already support the Docker API easy, including the standard Docker client. To a Docker client a Swarm cluster is just another Docker host.

Swarm, like many other Docker tools, follows a design principle of "batteries included but removable". This means it ships with tooling and backend integration for simple use cases and provides an API for integration with more complex tools and use cases. Swarm is shipped integrated into Docker since Docker 1.12. Prior to that it was a standalone application licensed with the Apache 2 license.

Understanding the Swarm

A swarm is a cluster of Docker hosts onto which you can deploy services. Since Docker 1.12 the Docker command line tool has included a `swarm` mode. This allows the `docker` binary to create and manage swarms as well as run local containers.

A swarm is made up of manager and worker nodes. Manager do the dispatching and organizing of work on the swarm. Each unit of work is called a task. Managers also handle all the cluster management functions that keep the swarm healthy and active. You can have many manager nodes, if there is more than one then the manager node will conduct an election for a leader.

Worker nodes run the tasks dispatched from manager nodes. Out of the box, every node, managers and workers, will run tasks. You can instead configure a swarm manager node to only perform management activities and not run tasks.

As a task is a pretty atomic unit swarms use a bigger abstraction, called a service as a building block. Services defined which tasks are executed on your nodes. Each service consists of a container image and a series of commands to execute inside one or more containers on the nodes. You can run services in a number of modes:

- Replicated services - a swarm manager distributes replica tasks amongst workers according to a scale you specify.
- Global services - a swarm manager dispatches one task for the service on every available worker.

The swarm also manages load balancing and DNS much like a local Docker host. Each swarm can expose ports, much like Docker containers publish ports. Like container ports, These can be automatically or manually defined. The swarm handles internal DNS much like a Docker host allowing services and workers to be discoverable inside the swarm.

Installing Swarm

The easiest way to install Swarm is to use Docker itself. As a result, Swarm doesn't have anymore prerequisites than what we saw in Chapter 2. These instructions assume you've installed Docker in accordance with those instructions.

TIP Prior to Docker 1.12, when Swarm was integrated into Docker, you can use Swarm via a Docker image provided by the Docker Inc team called `swarm`. Instructions for installation and usage are available on the Docker Swarm documentation site.

We're going to reuse our `larry`, `curly` and `moe` hosts to demonstrate Swarm.

The latest Docker release is already installed on these hosts and we're going to turn them into nodes of a Swarm cluster.

Setting up a Swarm

Now let's create a Swarm cluster. Each node in our cluster runs a Swarm node agent. Each agent registers its related Docker daemon with the cluster. Also available is the Swarm manager that we'll use to manage our cluster. We're going to create two cluster workers and a manager on our three hosts.

Table 7.3: Swarm addresses and roles

Host	IP Address	Role
larry	162.243.167.159	Manager
curly	162.243.170.66	Worker
moe	159.203.191.16	Worker

We also need to make sure some ports are open between all our nodes. We need to consider the following access:

Table 7.4: Docker Swarm default ports.

Port	Purpose
2377	Cluster Management
7946 + udp	Node communication
4789 + udp	Overlay network

We're going to start with registering a Swarm on our `larry` node and use this host as our Swarm manager. We're again going to need `larry`'s public IP address. Let's make sure it's still assigned to an environment variable.

Listing 7.62: Getting public IP on larry again

```
larry$ PUBLIC_IP="$(ifconfig eth0 | awk -F ' *|:' '/inet addr/{
    print $4}')"
larry$ echo $PUBLIC_IP
162.243.167.159
```

Now let's initialize a swarm on `larry` using this address.

Listing 7.63: Initializing a swarm on larry

```
$ sudo docker swarm init --advertise-addr $PUBLIC_IP
Swarm initialized: current node (bu84wfix0h0x31aut8qlpbi9x) is
   now a manager.

To add a worker to this swarm, run the following command:
    docker swarm join \
    --token SWMTKN-1-2
   mk0wnb9m9cdwhheoysr3pt8orxku8c7k3x3kjjsxatc5ua72v-776
   lg9r60gigwb32q329m0dli \
    162.243.167.159:2377

To add a manager to this swarm, run the following command:
    docker swarm join \
    --token SWMTKN-1-2
   mk0wnb9m9cdwhheoysr3pt8orxku8c7k3x3kjjsxatc5ua72v-78
   bsc54abf35rhpr3ntbh98t8 \
    162.243.167.159:2377
```

You can see we've run a `docker` command: `swarm`. We've then used the `init` option to initialize a swarm and the `--advertise-addr` flag to specify the management IP of the new swarm.

We can see the swarm has been started, assigning `larry` as the swarm manager. Each swarm has two registration tokens initialized when the swarm begins. One token for a manager and another for worker nodes. Each type of node can use this token to join the swarm. We can see one of our tokens:

`SWMTKN-1-2mk0wnb9m9cdwhheoysr3pt8orxku8c7k3x3kjjsxatc5ua72v-776 lg9r60gigwb32q329m0dli`

You can see that the output from initializing the swarm has also provided sample commands for adding workers and managers to the new swarm.

TIP If you ever need to get this token back again then you can run the `docker swarm join-token worker` command on the Swarm manager to retrieve it.

Let's look at the state of our Swarm by running the `docker info` command.

Listing 7.64: The Docker

```
larry$ sudo docker info
. . .
Swarm: active
 NodeID: bu84wfix0h0x31aut8qlpbi9x
 Is Manager: true
 ClusterID: 0qtrjqv37gs3yc5f7ywt8nwfq
 Managers: 1
 Nodes: 1
 Orchestration:
  Task History Retention Limit: 5
 Raft:
  Snapshot interval: 10000
  Heartbeat tick: 1
  Election tick: 3
 Dispatcher:
  Heartbeat period: 5 seconds
 CA configuration:
  Expiry duration: 3 months
 Node Address: 162.243.167.159
. . .
```

By enabling a swarm you'll see a new section in the `docker info` output.

We can also view information on the nodes inside the swarm using the `docker node` command.

Listing 7.65: The docker node command

```
larry$ sudo docker node ls
ID                            HOSTNAME  STATUS  AVAILABILITY
   MANAGER STATUS
bu84wfix0h0x31aut8qlpbi9x *   larry     Ready   Active
   Leader
```

The `docker node` command with the `ls` flag shows the list of nodes in the swarm. Currently we only have one node `larry` which is active and shows its role as `Leader` of the manager nodes.

Let's add our `curly` and `moe` hosts to the swarm as workers. We can use the command emitted when we initialized the swarm.

Listing 7.66: Adding worker nodes to the cluster

```
curly$ sudo docker swarm join \
--token SWMTKN-1-2
   mk0wnb9m9cdwhheoysr3pt8orxku8c7k3x3kjjsxatc5ua72v-776
   lg9r60gigwb32q329m0dli \
162.243.167.159:2377
This node joined a swarm as a worker.
```

The `docker swarm join` command takes a token, in our case the worker token, and the IP address and port of a Swarm manager node and adds that Docker host to the swarm.

And then again with the same command on the `moe` node. Now let's look at our node list again on the `larry` host.

Listing 7.67: Running the docker node command again

```
larry$ sudo docker node ls
ID                           HOSTNAME  STATUS  AVAILABILITY
    MANAGER STATUS
bu84wfix0h0x31aut8qlpbi9x *  larry     Ready   Active
    Leader
c6viix7oja1twnyuc8ez7txhd    curly     Ready   Active
dzxrvk6awnegjtj5aixnojetf    moe       Ready   Active
```

Now we can see two more nodes added to our swarm as workers.

Running a service on your Swarm

With the swarm running, we can now start to run services on it. Remember services are a container image and commands that will be executed on our swarm nodes. Let's create a simple replica service now. Remember that replica services run the number of tasks you specify.

Listing 7.68: Creating a swarm service

```
$ sudo docker service create --replicas 2 --name heyworld ubuntu
    /bin/sh -c "while true; do echo hey world; sleep 1; done"
8bl7yw1z3gzir0rmcvnrktqol
```

TIP You can find the full list of `docker service create` flags on the Docker documentation site.

We've used the `docker service` command with the `create` keyword. This creates services on our swarm. We've used the `--name` flag to call the service: `heyworld`. The `heyworld` runs the `ubuntu` image and a `while` loop that echoes `hey world`. The `--replicas` flag controls how many tasks are run on the swarm. In this case we're running two tasks.

Let's look at our service using the `docker service ls` command.

Listing 7.69: Listing the services

```
$ sudo docker service ls
ID            NAME      REPLICAS  IMAGE   COMMAND
8bl7yw1z3gzi  heyworld  2/2       ubuntu  /bin/sh -c while true;
   do echo hey world; sleep 1; done
```

This command lists all services in the swarm. We can see that our `heyworld` service is running on two replicas. We can inspect the service in further detail using the `docker service inspect` command. We've also passed in the `--pretty` flag to return the output in an elegant form.

Listing 7.70: Inspecting the heyworld service

```
$ sudo docker service inspect --pretty heyworld
ID:     8bl7yw1z3gzir0rmcvnrktqol
Name:       heyworld
Mode:       Replicated
 Replicas:  2
Placement:
UpdateConfig:
 Parallelism:   1
 On failure:    pause
ContainerSpec:
 Image:     ubuntu
 Args:      /bin/sh -c while true; do echo hey world; sleep 1;
   done
Resources:
```

But we still don't know where the service running. Let's look at another command: `docker service ps`.

Listing 7.71: Checking the heyworld service process

```
$ sudo docker service ps heyworld
ID      NAME        IMAGE  NODE  DESIRED STATE CURRENT STATE
103q... heyworld.1 ubuntu larry Running       Running about a
   minute ago
6ztf... heyworld.2 ubuntu moe   Running       Running about a
   minute ago
```

We can see each task, suffixed with the task number, and the node it is running on.

Now, let's say we wanted to add another task to the service, scaling it up. To do this we use the `docker service scale` command.

Listing 7.72: Scaling the heyworld service

```
$ sudo docker service scale heyworld=3
heyworld scaled to 3
```

We specify the service we want to scale and then the new number of tasks we want run, here `3`. The swarm has then let us know it has scaled. Let's again check the running processes.

Listing 7.73: Checking the heyworld service process

```
$ sudo docker service ps heyworld
ID      NAME       IMAGE  NODE  DESIRED STATE CURRENT STATE
103q... heyworld.1 ubuntu larry Running       Running 5 minutes
   ago
6ztf... heyworld.2 ubuntu moe   Running       Running 5 minutes
   ago
1gib... heyworld.3 ubuntu curly Running       Running about a
   minute ago
```

We can see that our service is now running on a third node.

In addition to running replica services we can also run global services. Rather than running as many replicas as you specify, global services run on every worker in the swarm.

Listing 7.74: Running a global service

```
$ sudo docker service create --name heyworld_global --mode global
    ubuntu /bin/sh -c "while true; do echo hey world; sleep 1;
   done"
```

Here we've started a global service called `heyworld_global`. We've specified the `--mode` flag with a value of `global` and run the same `ubuntu` image and the same command we ran above.

Let's see the processes for the `heyworld_global` service using the `docker service ps` command.

Listing 7.75: The heyworld_global process

```
$ sudo docker service ps heyworld_global
ID       NAME                IMAGE  NODE  DESIRED STATE CURRENT
   STATE
c8c1... heyworld_global     ubuntu moe   Running       Running 30
    seconds ago
48wm...  \_ heyworld_global ubuntu curly Running       Running 30
    seconds ago
8b8u...  \_ heyworld_global ubuntu larry Running       Running 29
    seconds ago
```

We can see that the `heyworld_global` service is running on every one of our nodes.

If we want to stop a service we can run the `docker service rm` command.

Listing 7.76: Deleting the heyworld service

```
$ sudo docker service rm heyworld
heyworld
```

We can now list the running services.

Listing 7.77: Listing the remaining services

```
$ sudo docker service ls
ID       NAME            REPLICAS IMAGE  COMMAND
5k3t...  heyworld_global global   ubuntu /bin/sh -c...
```

And we can see that only the `heyworld_global` service remains running.

TIP Swarm mode also allows for scaling, draining and staged upgrades. You can find some examples of this in Docker Swarm tutorial.

Orchestration alternatives and components

As we mentioned earlier, Compose 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 containers. Think of it as an extension of systemd that operates at the cluster level instead of the machine level.

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.

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 focused 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.

Summary

In this chapter we've introduced you to orchestration with Compose. We've shown you how to add a Compose configuration file to create simple application stacks. We've shown you how to run Compose and build those stacks and how to perform basic management tasks on them.

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.

We also took a look at Docker Swarm as a Docker clustering and scheduling tool. We saw how to install Swarm, how to manage it and how to schedule workloads across it.

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 an 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 to 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 Engine API - provides integration with the Docker daemon.

All three APIs are broadly RESTful. In this chapter, we'll focus on the Engine API because it is key to any programmatic integration and interaction with Docker.

First steps with the Engine API

Let's explore the Docker Engine API and see its capabilities. Firstly, we need to remember the Engine 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.

If you want to use the Docker API locally we use the `curl` command to query it, like so:

Listing 8.1: Querying the Docker API locally

```
$ curl --unix-socket /var/run/docker.sock http:/info
{"ID":"PH4R:BT7H:44F6:GQGP:FS2O:7OZO:HQ2P:NSVF:MK27:NBGZ:N3VP:
   K2O5","Containers":3,"ContainersRunning":3,"ContainersPaused"
   :0,"ContainersStopped":0,"Images":3,"
. . .
}
```

On most distributions, we can bind the Docker daemon to a network interface by editing the daemon's startup configuration files. For older Ubuntu or Debian releases, this would be the `/etc/default/docker` file; for those releases with Upstart, it would be the `/etc/init/docker.conf` file; for systemd releases it'll be `/lib/systemd/system/docker.service`. 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 8.2: Default systemd daemon start options

```
ExecStart=/usr/bin/dockerd --selinux-enabled
```

To:

Listing 8.3: Network binding systemd daemon start options

```
ExecStart=/usr/bin/dockerd --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 8.4: 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 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 8.5: 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 8.6: 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 add authentication to this connection.

Testing the Docker Engine 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 8.7: Using the info API endpoint

```
$ curl http://docker.example.com:2375/info | python3 -mjson.tool
{
    "ID": "PH4R:BT7H:44F6:GQGP:FS2O:7OZO:HQ2P:NSVF:MK27:NBGZ:N3VP
   :K2O5",
    "Containers": 7,
    "ContainersRunning": 1,
    "ContainersPaused": 0,
    "ContainersStopped": 6,
    "Images": 3,
    "Driver": "aufs",
    "DriverStatus": [
        [
. . .
```

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.example.com` on port `2375` with endpoint `info`.

We 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. We've passed the JSON through python's JSON tool to prettify it.

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 8.8: Getting a list of images via API

```
$ curl http://docker.example.com:2375/images/json | python3 -
   mjson.tool
[
    {
        "Id": "sha256:
   b608dbb10e2564f5bd0eef045bf297e56b1149edc70bece54fef4b217261a473
   ",
        "ParentId": "",
        "RepoTags": [
            "jamtur01/distributed_app:latest"
        ],
        "RepoDigests": [
            "jamtur01/distributed_app@sha256:
   ecc6b617e9c776d8bd7ed281a55b02e9214d701cad72b9628f5668edfbb86a26
   "
        ],
        "Created": 1470488372,
        "Size": 469434429,
        "VirtualSize": 469434429,
        "Labels": {}
    },
. . .
]
```

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 8.9: Getting a specific image

```
$ curl http://docker.example.com:2375/images/
    b608dbb10e2564f5bd0eef045bf297e56b1149edc70bece54fef4b217261a473
    /json | python3 -mjson.tool
{
     "Id": "sha256:
    b608dbb10e2564f5bd0eef045bf297e56b1149edc70bece54fef4b217261a473
    ",
     "RepoTags": [
         "jamtur01/distributed_app:latest"
     ],
     "RepoDigests": [
         "jamtur01/distributed_app@sha256:
    ecc6b617e9c776d8bd7ed281a55b02e9214d701cad72b9628f5668edfbb86a26
    "
     ],
     "Parent": "",
     "Comment": "",
     "Created": "2016-08-06T12:59:32.957396238Z",
. . .
    }
}
```

Here we see a subset of the output of inspecting our `jamtur01/distributed_app` image. And finally, like the command line, we can search for images on the Docker Hub.

Listing 8.10: Searching for images with the API

```
$ curl "http://docker.example.com:2375/images/search?term=
   jamtur01" | python3 -mjson.tool
[
    {
        "star_count": 0,
        "is_official": false,
        "name": "jamtur01/docker-jenkins-sample",
        "is_automated": true,
        "description": ""
    },
    {
        "star_count": 5,
        "is_official": false,
        "name": "jamtur01/docker-presentation",
        "is_automated": true,
        "description": ""
    },
. . .
]
```

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 Engine 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 8.11: Listing running containers

```
$ curl -s "http://docker.example.com:2375/containers/json" |
    python3 -mjson.tool
[
    {
        "Id": "
    d580b605fa1bcd210af0d2fe28e50a018f9ea546b56e8b28806d8dc16596340e
    ",
        "Names": [
            "/heyworld_global.0.bbctscdrhkro371mkieb0roid"
        ],
        "Image": "ubuntu:latest",
        "ImageID": "sha256:42118
    e3df429f09ca581a9deb3df274601930e428e452f7e4e9f1833c56a100a",
        "Command": "/bin/sh -c 'while true; do echo hey world;
    sleep 1; done'",
        "Created": 1470676972,
        "Ports": [],
        "Labels": {
            "com.docker.swarm.node.id": "
    c6viix7oja1twnyuc8ez7txhd",
            "com.docker.swarm.service.id": "5
    k3tw55i050qqh16ob9651pqx",
            "com.docker.swarm.service.name": "heyworld_global",
            "com.docker.swarm.task": "",
            "com.docker.swarm.task.id": "
    bbctscdrhkro371mkieb0roid",
            "com.docker.swarm.task.name": "heyworld_global.0"
        },
        "State": "running",
        "Status": "Up 11 hours",
        "HostConfig": {
            "NetworkMode": "default"
        }
. . .
    }
]
```

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 8.12: 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 8.13: 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 8.14: 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 8.15: 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 8.16: API equivalent for docker run command

```
$ sudo docker run jamtur01/jekyll
```

We can also inspect the resulting container via the `/containers/` endpoint.

Listing 8.17: Listing all containers via the API

```
$ curl http://docker.example.com:2375/containers/591
   ba02d8d149e5ae5ec2ea30ffe85ed47558b9a40b7405e3b71553d9e59bed3
   /json | python3 -mjson.tool
{
    "Args": [
        "build",
        "--destination=/var/www/html"
    ],
. . .
        "Hostname": "591ba02d8d14",
        "Image": "jamtur01/jekyll",
. . .
    "Id": "591
   ba02d8d149e5ae5ec2ea30ffe85ed47558b9a40b7405e3b71553d9e59bed3
   ",
    "Image": "29
   d4355e575cff59d7b7ad837055f231970296846ab58a037dd84be520d1cc31
   ",
. . .
    "Name": "/hopeful_davinci",
. . .
}
```

Here we see we've queried our container using the container ID and shown a sampling of the data available to us.

Improving the TProv application

Now let's look at one of 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 8.18: The legacy TProv container launch methods

```
def create\_instance(name)
  cid = `docker run -P --volumes-from #{name} -d -t jamtur01/
   tomcat8 2>&1`.chop
  [\$?.exitstatus == 0, cid]
end
```

NOTE You can see the previous TProv code on GitHub.

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 8.19: The Docker Ruby client

```
require 'docker'
. . .

module TProvAPI
  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 8.20: 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 8.21: 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"=>"3.10.0-33-generic",
 "Os"=>"linux",
 "Version"=>"1.0.1"}
. . .
```

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 8.22: 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/
   tomcat8')
  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/tomcat8` 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.

You can install the API-enabled version of the TProv application via gem to see it in action.

Listing 8.23: Installing TProvAPI

```
$ sudo gem install tprov-api
```

NOTE You can see the updated TProv code on GitHub.

Authenticating the Docker Engine API

Whilst we've shown that we can connect to the Docker Engine API, that means that anyone else can also connect to the API. That poses a bit of a security issue. The Engine 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 8.24: 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 8.25: Create a CA directory

```
$ sudo mkdir /etc/docker
```

Now let's create our CA.

We first generate a private key.

Listing 8.26: 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 8.27: 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 8.28: 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 is 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 8.29: 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.

We also know folks connect to our host via IP address so we need to configure for that too.

Listing 8.30: Connect via IP address

```
$ echo subjectAltName = IP:x.x.x.x,IP:127.0.0.1 > extfile.cnf
```

Replacing `x.x.x.x` with the IP address(es) of your Docker daemon.

Now let's sign our CSR and generate our server certificate.

Listing 8.31: Signing our CSR

```
$ sudo openssl x509 -req -days 365 -in server.csr -CA ca.pem \
-CAkey ca-key.pem -out server-cert.pem -extfile extfile.cnf
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 8.32: 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 8.33: 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 8.34: 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 8.35: 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 8.36: 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 8.37: 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) []: Docker daemon
    host FQDN
Email Address []:

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

Replace the `Docker daemon host FQDN` with the fully-qualified domain name of your Docker daemon host.

We next need to enable client authentication for our key by adding some extended SSL attributes.

Listing 8.38: Adding Client Authentication attributes

```
$ echo extendedKeyUsage = clientAuth > extfile.cnf
```

Now let's sign our CSR with our CA.

Listing 8.39: 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 8.40: 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 8.41: 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 8.42: 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: 3.10.0-33-generic
Username: jamtur01
Registry: [https://index.docker.io/v1/]
WARNING: No swap limit support
```

We 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 8.43: 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 Engine API. We've also seen how to secure the Docker Engine 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. Most Docker folks congregate in three places:

NOTE Docker, Inc. also sells enterprise support for Docker. You can find the information on the Support page.

The Docker forums

There is a Docker forum available.

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` at `https://botbot.me/freenode/docker/` and for `#docker-dev` at `https://botbot.me/freenode/docker-dev/`.

Docker on GitHub

Docker (and most of its components and ecosystem) is hosted on GitHub. The principal repository for Docker itself is `https://github.com/docker/docker/`.

Other repositories of note are:

- distribution - The stand-alone Docker registry and distribution tools.
- runc - The Docker container format and CLI tools.

- Docker Swarm - Docker's orchestration framework.
- Docker Compose - The Docker Compose tool.

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 16.04).

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. The source code is stored in a repository called Moby, which is the codename for the broader ecosystem of Docker source code. It's named this to reflect that components of Docker are used in a variety of different platforms, not just the Docker Engine.

Listing 9.3: Check out the Docker source code

```
$ git clone https://github.com/moby/moby
$ cd moby
```

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 Docs repo on GitHub.

You can find specific guidelines and a basic style guide for the documentation at:

- `https://github.com/docker/docker.github.io/blob/master/README.md`.

You can build the documentation locally using Docker itself.

Make any changes you want to the documentation, and then you can stage the documentation locally to review your changes.

Build the environment

If you want to contribute to Docker Engine, 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 9.4: Building the Docker environment

```
$ cd moby
$ sudo make build
```

TIP This command will take some time to complete when you first execute it. It might require a host with at least 2Gb of RAM to also run the development build.

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 also create a Docker binary to test any fixes or features. We do this using the `make` tool again.

Listing 9.5: Building the Docker binary

```
$ sudo make binary
```

This command will create a `docker` and `dockerd` binary in a volume at `./bundles/<version>-dev/binary-client/` and `./bundles/<version>-dev/binary-daemon/` respectively. For example, we would create a client binary and associated checksums like so:

Listing 9.6: The Docker dev client binary

```
$ ls -l bundles/1.13.0-dev/binary-client/
total 15344
lrwxrwxrwx 1 root root       17 Aug  9 13:59 docker -> docker-
   1.13.0-dev
-rwxr-xr-x 1 root root 15700192 Aug  9 13:59 docker-1.13.0-dev
-rw-r--r-- 1 root root       52 Aug  9 13:59 docker-1.13.0-dev.
   md5
-rw-r--r-- 1 root root       84 Aug  9 13:59 docker-1.13.0-dev.
   sha256
```

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 9.7: Using the development daemon

```
$ sudo service docker stop
$ ~/moby/bundles/1.13.0-dev/binary-daemon/dockerd
```

This will run the development Docker daemon interactively. You can also background the daemon if you wish.

We can then test the `docker` binary by running it against this daemon.

Listing 9.8: Using the development binary

```
$ ~/moby/bundles/1.13.0-dev/binary-client/docker version
Client:
 Version:      1.13.0-dev
 API version:  1.25
 Go version:   go1.6.3
 Git commit:   04e021d
 Built:        Tue Aug  9 13:58:52 2016
 OS/Arch:      linux/amd64

Server:
 Version:      1.13.0-dev
 API version:  1.25
 Go version:   go1.6.3
 Git commit:   04e021d
 Built:        Tue Aug  9 13:58:52 2016
 OS/Arch:      linux/amd64
```

You can see that we're running a `1.13.0-dev` client, this binary, against the `1.13.0-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 9.9: 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 9.10: 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 9.11: 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 at http://blog.docker.com/2014/01/docker-code-contributions-require-developer-certificate-of-origin/. The certificate is easy to apply. All you need to do is add a line to each Git commit message.

Listing 9.12: 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 Documentation repository.
- Your patch fixes Markdown formatting or syntax errors in the documentation contained in the Documentation repository.

It's also pretty easy to automate the signing of your Git commits using the `git commit -s` command.

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.

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 (or for documentation - a minimum of two maintainers). If a change affects Documentation and `registry/`, then it needs two maintainers of Documentation 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.

List of Figures

Listings

Index

Thanks! I hope you enjoyed the book.

ISBN 978-0-9888202-0-3