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.

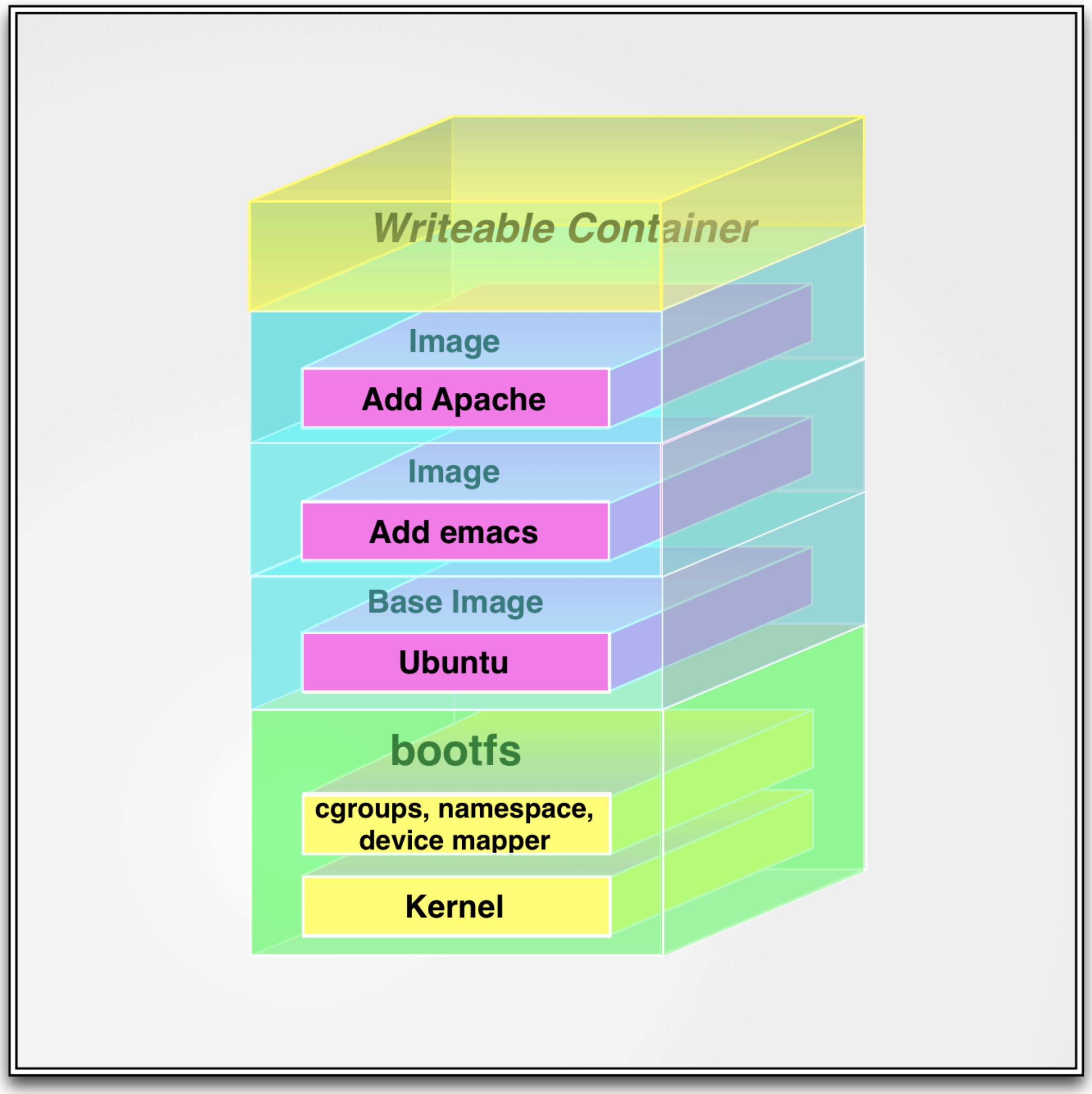


Figure 4.1: The Docker filesystem layers

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

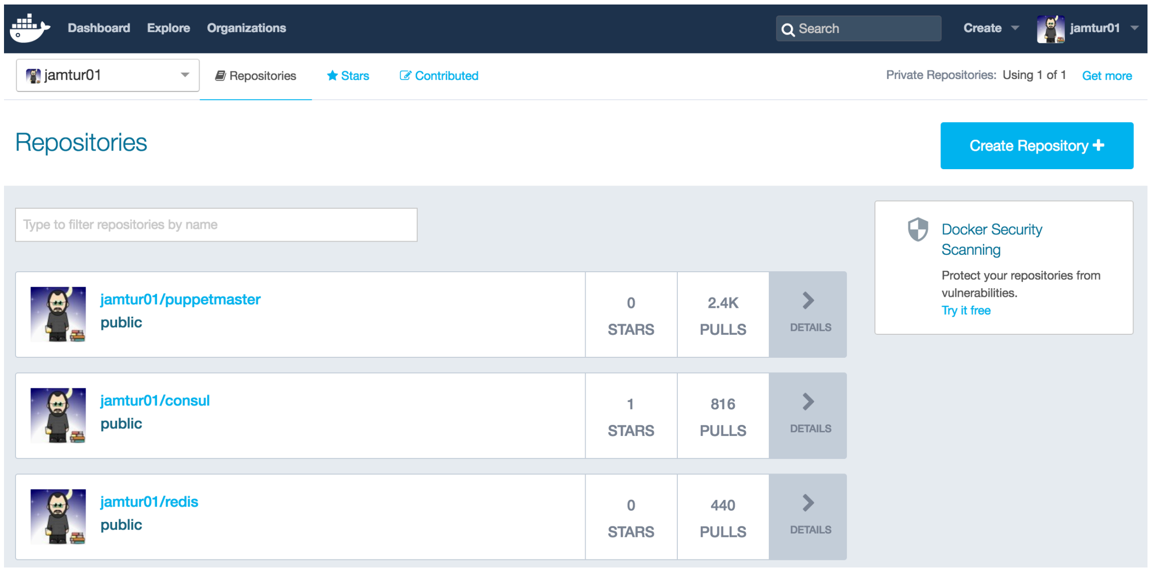
```
$ sudo docker images
REPOSITORY TAG     IMAGE ID     CREATED     VIRTUAL SIZE
ubuntu     latest  c4ff7513909d 6 days ago  225.4 MB
```

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.



Protect your repositories from vulnerabilities.

jamtur01/puppetmaster public	0 STARS	2.4K PULLS
jamtur01/consul public	1 STARS	816 PULLS
jamtur01/redis public	0 STARS	440 PULLS

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, 18.04). Let's get another image from the `ubuntu` repository now.

Listing 4.3: Pulling the Ubuntu 18.04 image

```
$ sudo docker pull ubuntu:18.04
18.04: Pulling from library/ubuntu
Digest: sha256:
   c6674c44c6439673bf56536c1a15916639c47ea04c3d6296c5df938add67b54b

Status: Downloaded newer image for ubuntu:18.04
```

Here we've used the `docker pull` command to pull down the Ubuntu 18.04 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     18.04   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 `18.04` 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 `18.04` tag collects together all the layers of the Ubuntu 18.04 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:18.04 /bin/
   bash
root@79e36bff89b4:/#
```

This launches a container from the `ubuntu:18.04` image, which is an Ubuntu 18.04 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 16.04 and 18.04, so it would be useful to specifically state that we're using `ubuntu:18.04` 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.

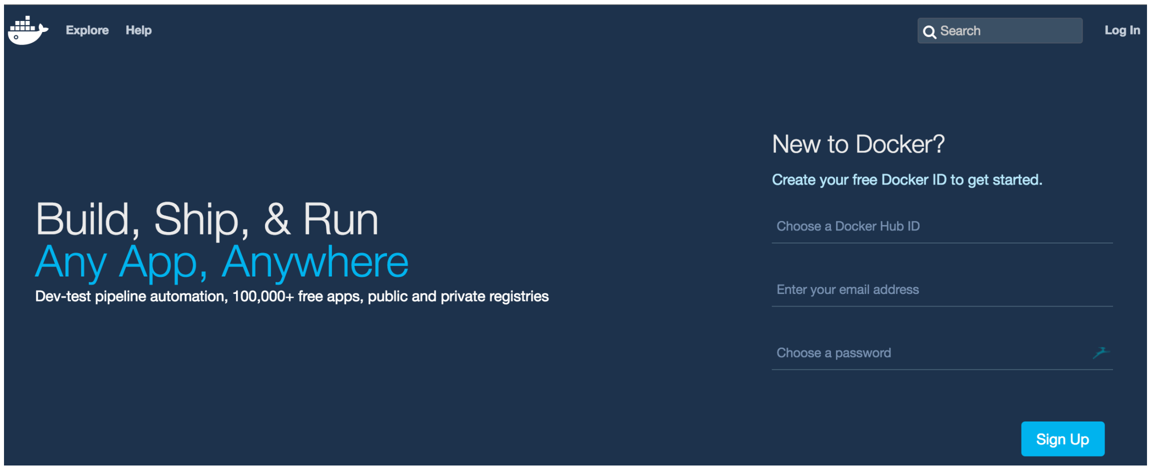


Figure 4.3: Creating a Docker Hub account.

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:18.04` image as our base image. This specification will build an image on top of an Ubuntu 18.04 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; apt-get install -y nginx
 ---> Running in 4b989d4730dd
 ---> 93fb180f3bc9
Removing intermediate container 4b989d4730dd
Step 3 : RUN echo 'Hi, I am in your container'
>/var/www/html/index.html
 ---> Running in b51bacc46eb9
 ---> b584f4ac1def
Removing intermediate container b51bacc46eb9
Step 4 : 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:18.04`. 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 up-loaded 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
```

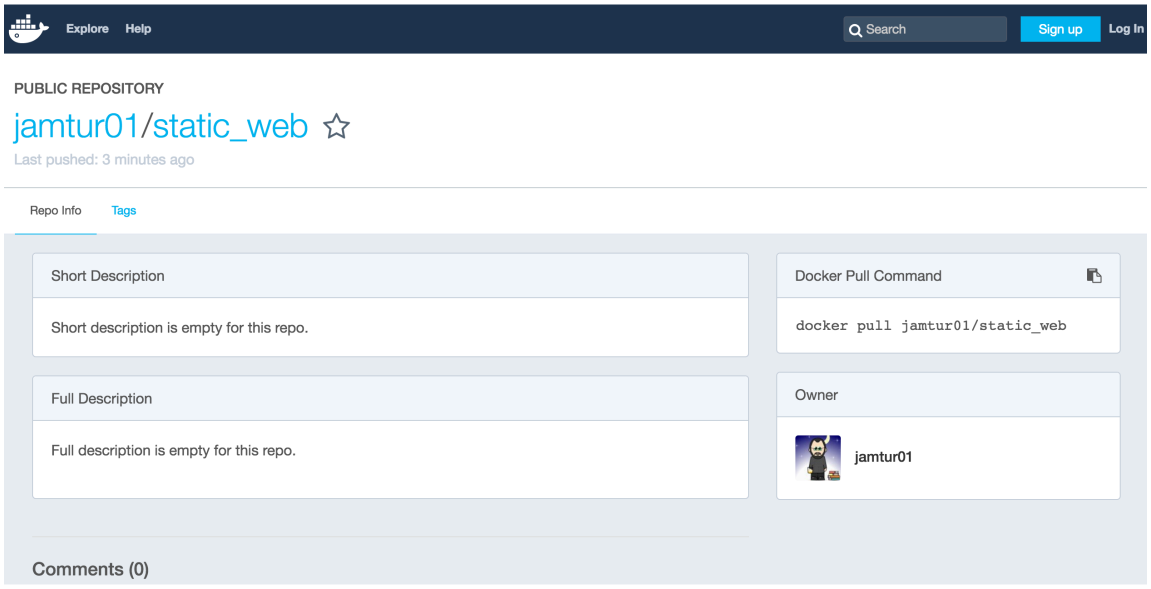
What's gone wrong here? We've tried to push our image to the repository `static_web`, but Docker knows this is a root repository. Root repositories are managed only by the Docker, Inc., team and will reject our attempt to write to them as unauthorized. Let's try again, rebuilding our image with a user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



PUBLIC REPOSITORY

jamtur01/static_web

Last pushed: 3 minutes ago

Short Description

Short description is empty for this repo.

Full Description

Full description is empty for this repo.

Docker Pull Command

```
docker pull jamtur01/static_web
```

Owner

jamtur01

Comments (0)

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.

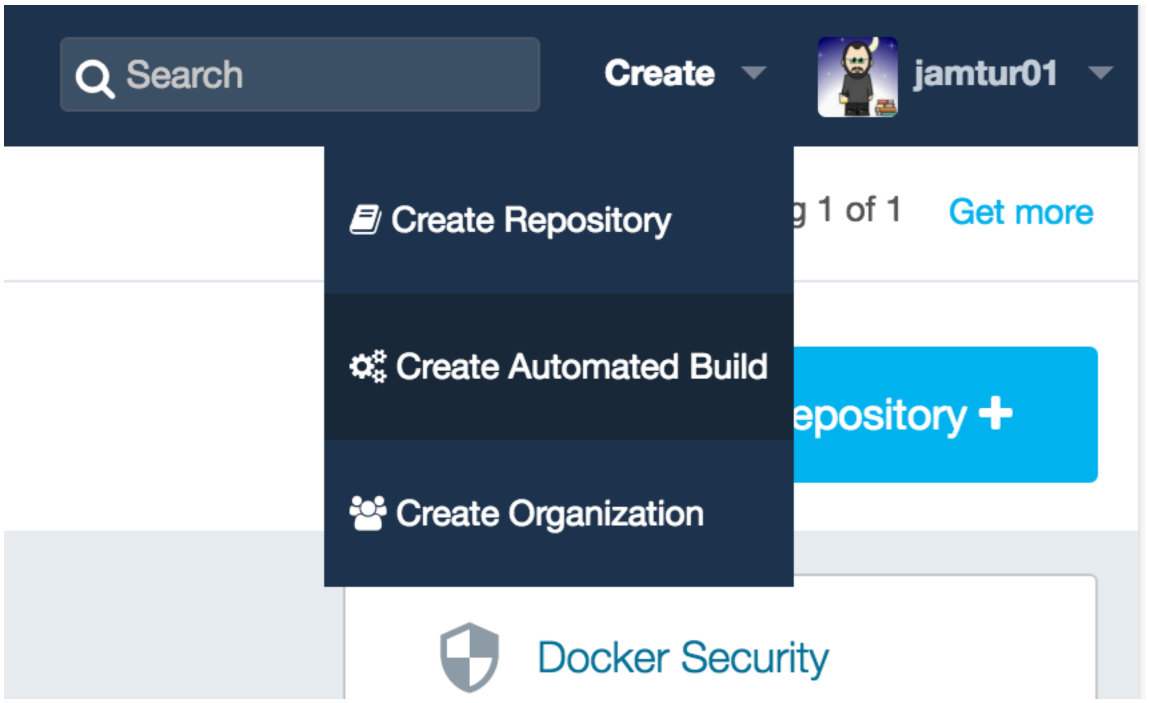


Figure 4.5: The Add Repository button.

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

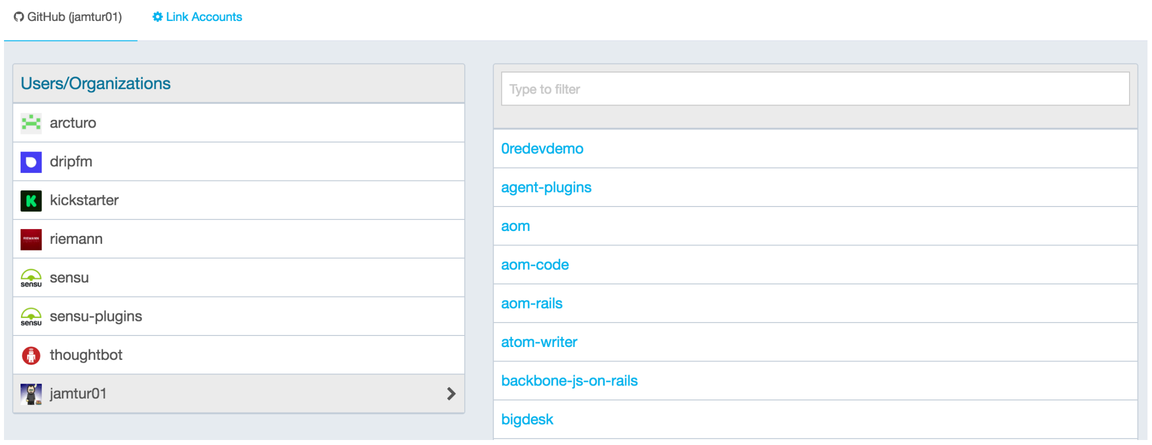


Figure 4.6: Selecting your repository.

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

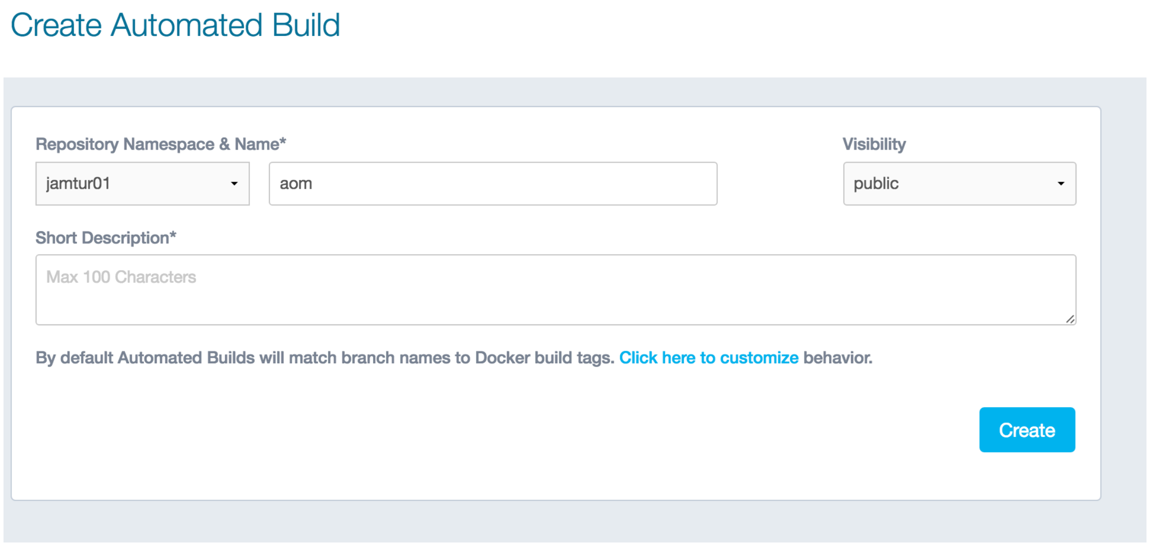

Create Automated Build
Repository Namespace & Name*
jamtur01
aom
Visibility
public
Short Description*
Max 100 Characters
By default Automated Builds will match branch names to Docker build tags. Click here to customize behavior.
Create

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.

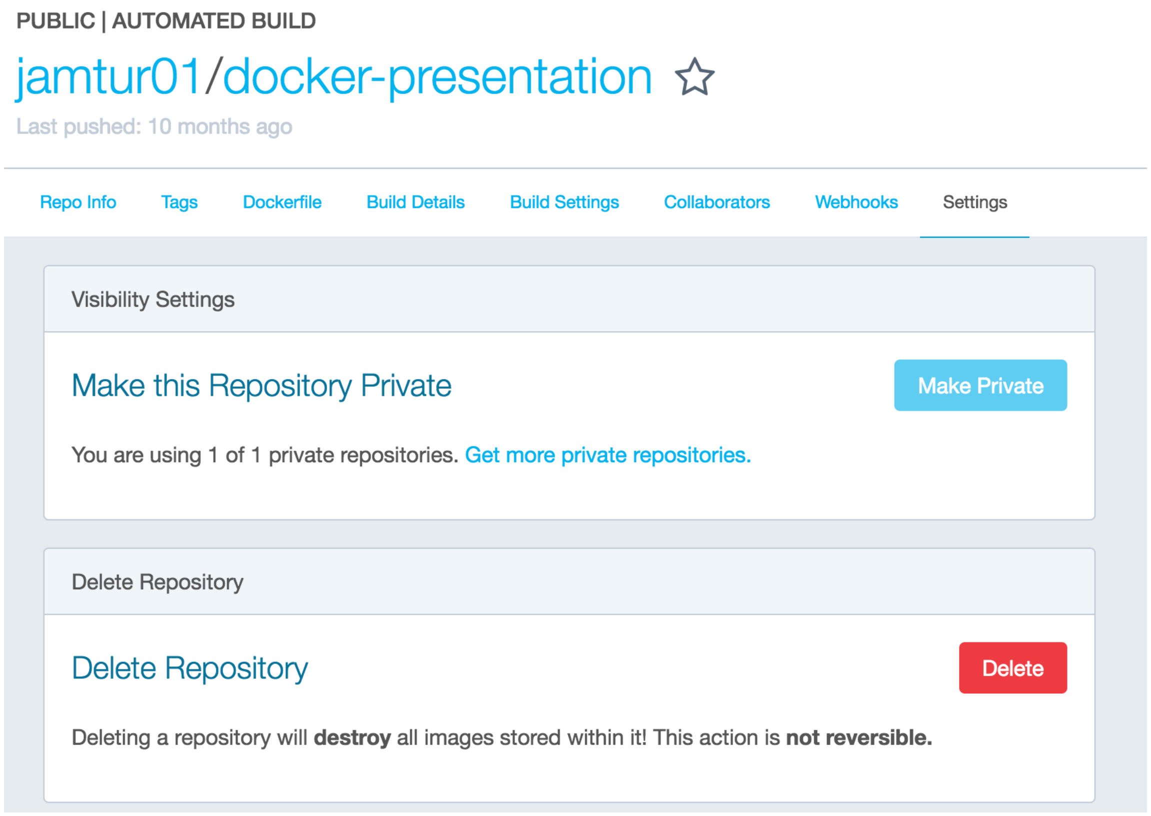
Listing 4.90: Deleting a Docker image

```
$ 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.



You are using 1 of 1 private repositories. Get more private repositories.

Deleting a repository will **destroy** all images stored within it! This action is **not reversible.**

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.