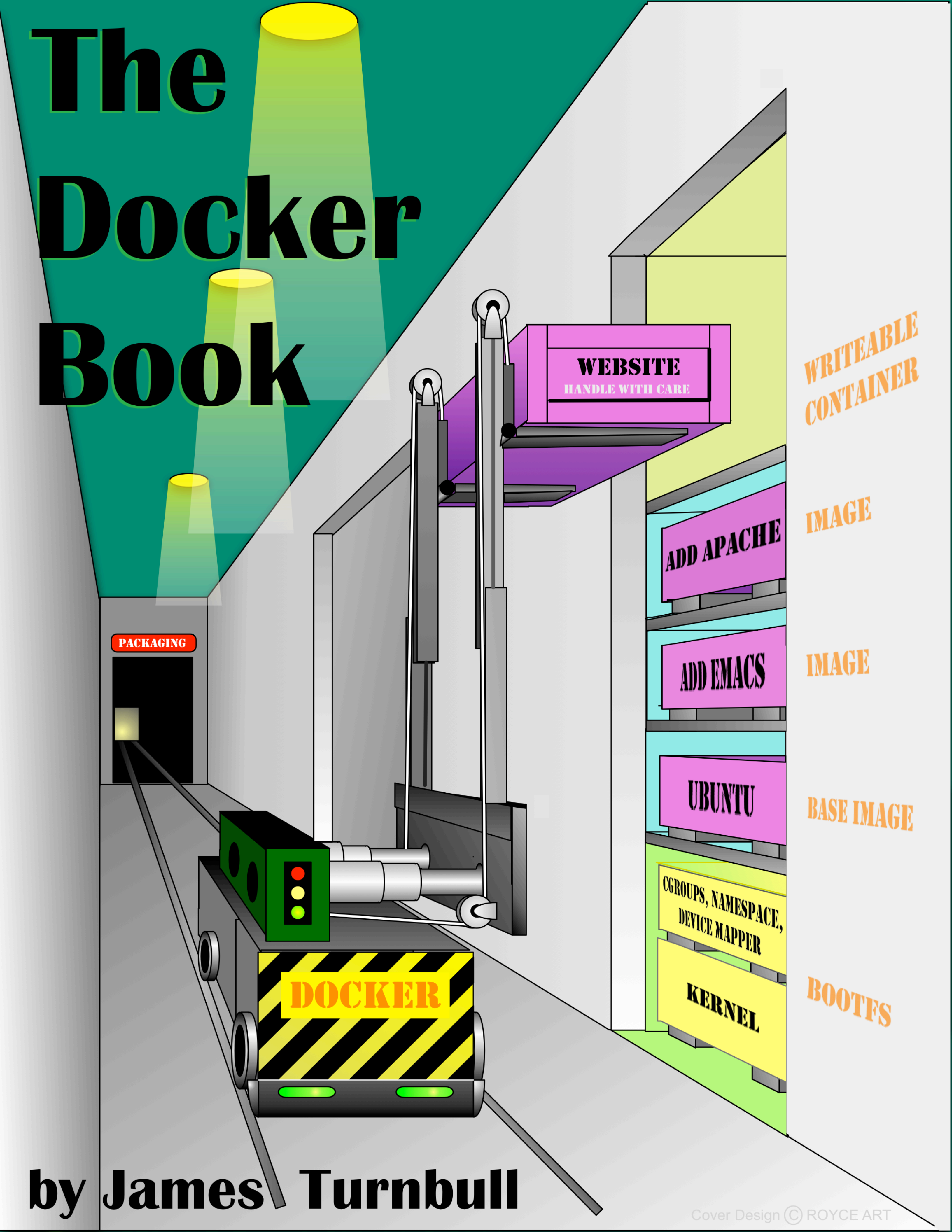
The Docker Book



by James Turnbull

Cover Design © ROYCE ART

The Docker Book

James Turnbull

July 17, 2014

Version: v1.0.2 (6741882)

Website: The Docker Book

Contents

List of Figures

Listings

Chapter 1

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 1.1: Revisiting creating a basic Docker container

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

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

In this chapter, we're going to explore images, the repositories that hold images, and the registries that store repositories. 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.

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.

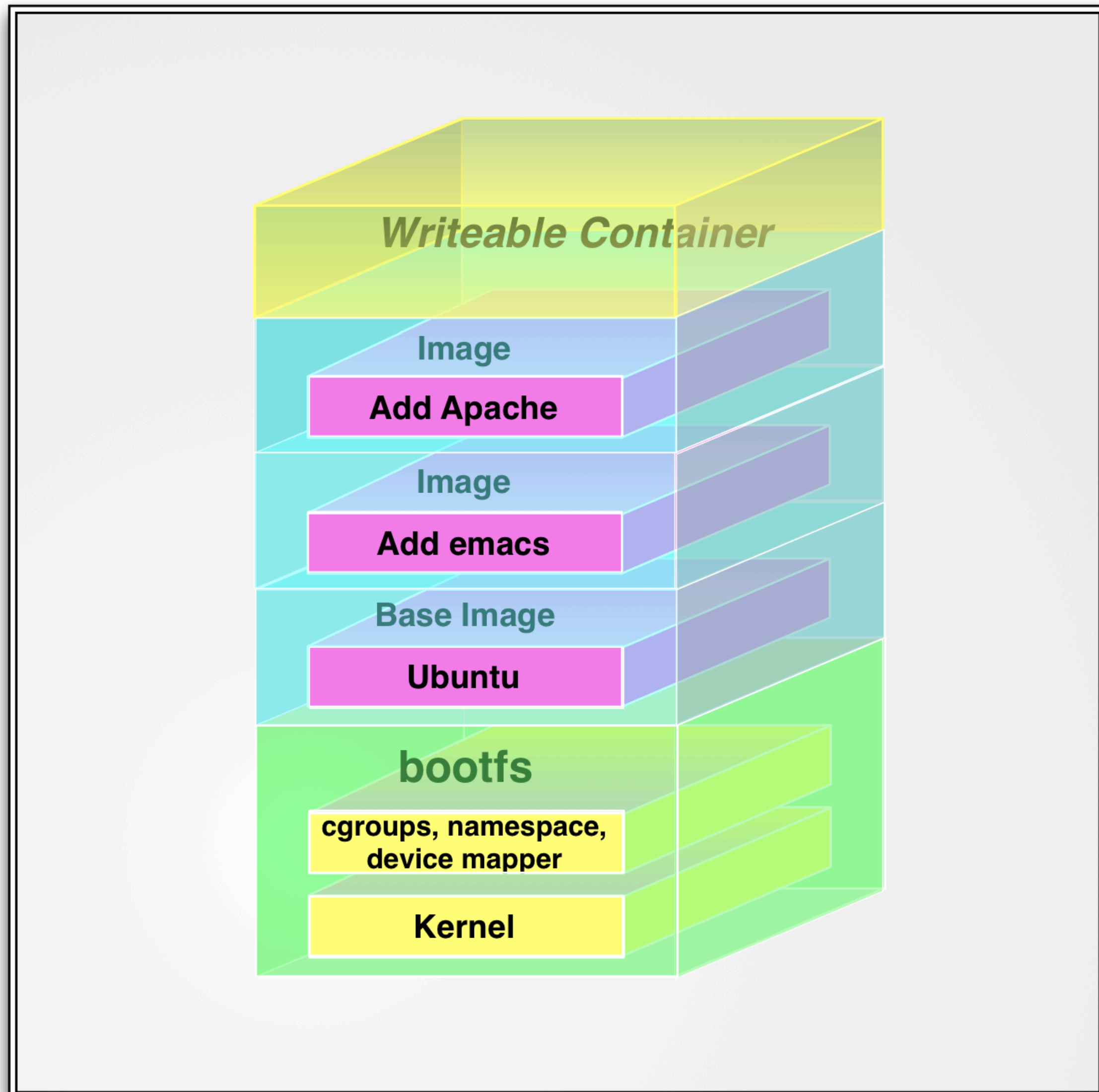


Figure 1.1: The Docker filesystem layers

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 1.2: Listing Docker images

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

We can see that we've got a list of images, from a repository called `ubuntu`. So where do these images 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 These 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.

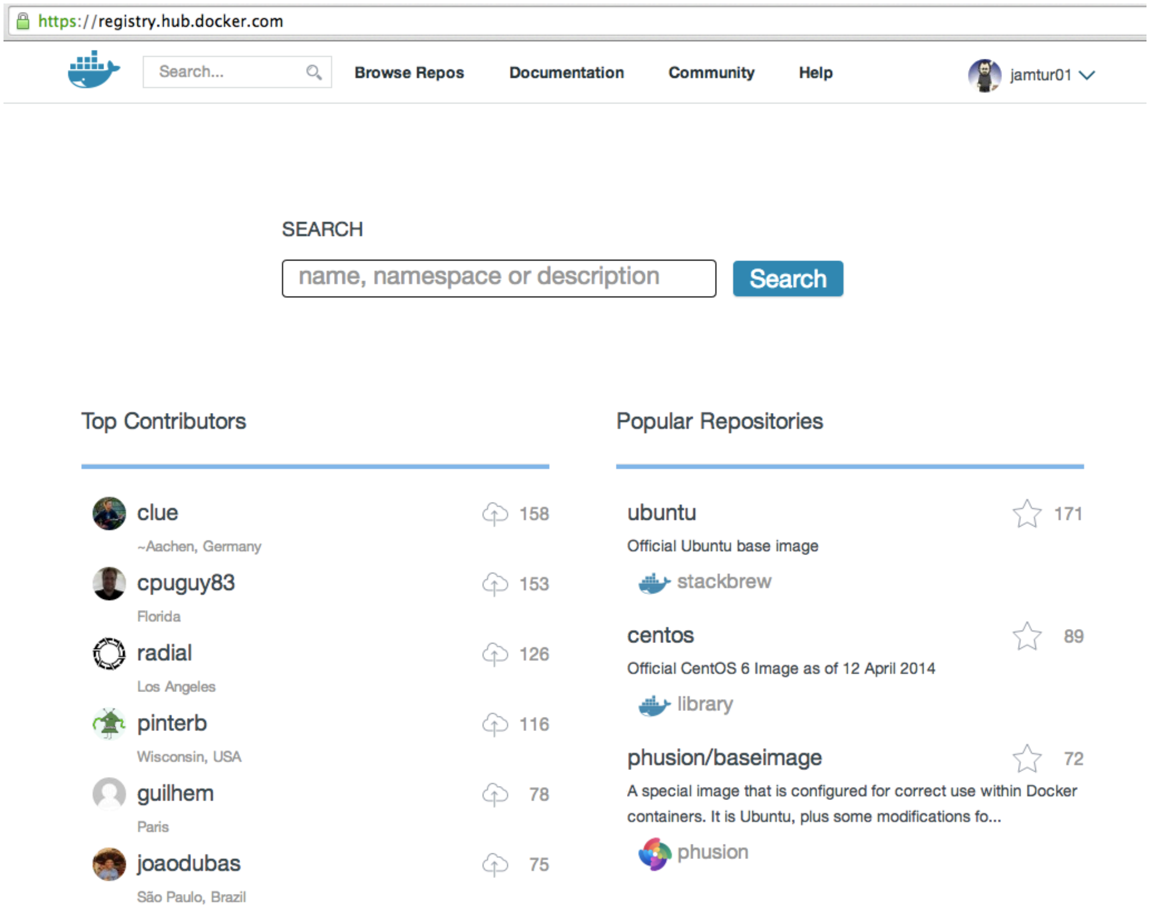


Figure 1.2: Docker Hub

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, and 14.04). Each of these images is identified by tags; for example, if we want the Ubuntu 14.04 image, we specify:

Listing 1.3: Specifying an image via tags

```
ubuntu:14.04
```

where `ubuntu` is the repository name and `14.04` is the tagged image we specifically want to use.

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

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

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

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

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

Notice something else about our list of images? The list contains more than one entry for `ubuntu`. But didn't we just download one image called `ubuntu`? So why does the list contain thirteen entries? Well, the `ubuntu` image is not just one image. It's actually a series of images collected under a single repository. In this case, when we downloaded the `ubuntu` image, we actually got several versions of the Ubuntu operating system, including 10.04, 12.04, 13.04, and 14.04.

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

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

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

Listing 1.4: Running a tagged Docker image

```
$ sudo docker run -t -i --name new_container ubuntu:12.04 /bin/↩
  bash
root@79e36bff89b4:/#
```

This launches a container from the `ubuntu:12.04` image, which is an Ubuntu 12.04 operating system. We can also see that some images with the same ID (see image ID `74fe38d11401`) are tagged more than once. Image ID `74fe38d11401` is actually tagged both `12.0.4` and `precise`: the version number and code name for that Ubuntu release, respectively.

It's always a good idea to build a container from specific tags. That way we'll know exactly what the source of our container is. There are differences, for example, between Ubuntu 12.0.4 and 14.04, so it would be useful to specifically state that we're using `ubuntu:12.04` so we know exactly what we're getting.

Pulling images

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

Listing 1.5: Pulling the fedora image

```
$ sudo docker pull fedora
```

```
Pulling repository fedora
5cc9e91966f7: Download complete
b7de3133ff98: Download complete
511136ea3c5a: Download complete
ef52fb1fe610: Download complete
```

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

Listing 1.6: Viewing the fedora image

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

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

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

Searching for images

We can also search all of the publicly available images on Docker Hub using the `docker search` command:

Listing 1.7: Searching for images

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

TIP You can also browse the available images online at Docker Hub.

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

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

---

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

Let's pull down one of these images.

Listing 1.8: Pulling down the jamtur01/puppetmaster image

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

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

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

Listing 1.9: Creating a Docker container from the Puppet master image

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

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

Building our own images

So we've seen that we can pull down pre-prepared images with custom contents. How do we go about modifying our own images and updating and managing them? There are two ways to create a Docker image:

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

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

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

Creating a Docker Hub account

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

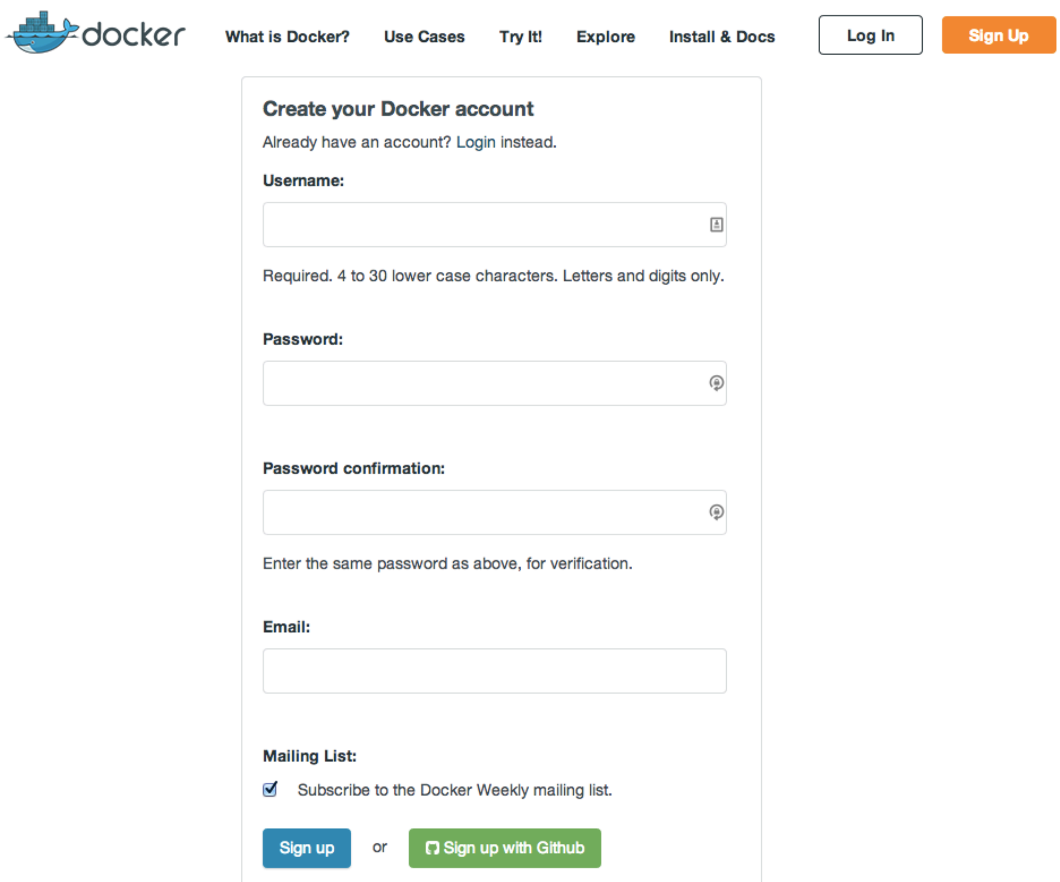


Figure 1.3: Creating a Docker Hub account.

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 1.10: Logging into the Docker Hub

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

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

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

Using Docker commit to create images

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

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

Listing 1.11: 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 1.12: 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 1.13: Committing the custom container

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

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

Let's look at our new image.

Listing 1.14: 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 1.15: Committing another custom container

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

```
f99ebb6fed1f559258840505a0f5d5b6173177623946815366f3e3acff01adef
```

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

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

Listing 1.16: Inspecting our committed image

```
$ sudo docker inspect jamtur01/apache2
[{
    "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 1.17: Running a container from our committed image

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

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

Building images with a Dockerfile

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

TIP The team at Docker, Inc., have also published a `Dockerfile` tutorial to help you learn how to build `Dockerfiles` here.

Our first Dockerfile

Let's now create a directory and an initial `Dockerfile`.

Listing 1.18: Creating a sample repository

```
$ mkdir sshd
$ cd sshd
$ touch Dockerfile
```

We've created a directory called `sshd` 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.

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 an SSH server.

Listing 1.19: Our first Dockerfile

```
# Version: 0.0.1
FROM ubuntu:12.04
MAINTAINER James Turnbull "james@example.com"
RUN apt-get update
```

```
RUN apt-get install -y openssh-server
RUN mkdir /var/run/sshd
RUN echo "root:password" | chpasswd
EXPOSE 22
```

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

Each instruction adds a new layer to the image and then commits the image. Docker executing instructions roughly follow a workflow:

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

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

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

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

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

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

We've followed these instructions with three `RUN` instructions. The `RUN` instruction executes commands on the current image. The commands in our example: updating the installed APT repositories, installing the `openssh-server` package, then creating the `/var/run/sshd` directory and setting the `root` password. 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 1.20: The RUN instruction in exec form

```
RUN [ "apt-get", " install", "-y", "openssh-server" ]
```

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 22) on the container. For security reasons, Docker doesn't open the port automatically, but waits for you to do it when you run the container using the `docker run` command. We'll see this shortly when we create a new container from this image.

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

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

Building the image from our Dockerfile

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

Listing 1.21: Running the Dockerfile

```
$ cd sshd
$ sudo docker build -t="jamtur01/sshd" .
Sending build context to Docker daemon  2.56 kB
Sending build context to Docker daemon
Step 1 : FROM ubuntu:12.04
  ---> 8dbd9e392a96
Step 2 : MAINTAINER James Turnbull "james@example.com"
  ---> Running in d97e0c1cf6ea
  ---> 85130977028d
Step 3 : RUN apt-get update
  ---> Running in 85130977028d
  ---> 997485f46ec4
Step 4 : RUN apt-get install -y openssh-server
  ---> Running in ffca16d58fd8
  ---> 9f551a68e60f
Step 5 : RUN mkdir /var/run/sshd
  ---> Running in 9f551a68e60f
  ---> 4faf762fcbe3
Step 6 : RUN echo "root:password" | chpasswd
  ---> Running in 4faf762fcbe3
  ---> 254a39b9d511
Step 7 : EXPOSE 22
  ---> Running in 4faf762fcbe3
  ---> 286b8a745bc2
Successfully built 286b8a745bc2
```

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 `sshd`. 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 1.22: Tagging a build

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

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

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

Listing 1.23: Building from a Git repository

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

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

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

Listing 1.24: Uploading the build context to the daemon

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

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

Next, you can see that each instruction in the `Dockerfile` has been executed with the image ID, `286b8a745bc2`, being returned as the final output of the build process. Each step and its associated instruction are run individually, and Docker has the committed the result of the 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 `opensshd-server`.

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

Listing 1.25: Managing a failed instruction

```
$ cd sshd
$ sudo docker build -t="jamtur01/sshd" .
Sending build context to Docker daemon  2.56 kB
Sending build context to Docker daemon
Step 1 : FROM ubuntu:12.04
  ---> 8dbd9e392a96
Step 2 : MAINTAINER James Turnbull "james@example.com"
  ---> Running in d97e0c1cf6ea
  ---> 85130977028d
Step 3 : RUN apt-get update
  ---> Running in 85130977028d
  ---> 997485f46ec4
Step 4 : RUN apt-get install -y opensshd-server
  ---> Running in ffca16d58fd8
Reading package lists...
Building dependency tree...
Reading state information...
E: Unable to locate package opensshd-server
```

```
2014/06/04 18:41:11 The command [/bin/sh -c apt-get install -y ↩
  opensshd-server] returned a non-zero code: 100
```

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

Listing 1.26: 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 opensshd-server` 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 1.27: Bypassing the Dockerfile build cache

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

Using the build cache for templating

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

Listing 1.28: A template Ubuntu Dockerfile

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

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

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

Listing 1.29: A template Fedora Dockerfile

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

which performs a very similar function for Fedora using Yum.

Viewing our new image

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

Listing 1.30: Listing our new Docker image

```
$ sudo docker images jamtur01/sshd
REPOSITORY      TAG     ID              CREATED         SIZE
jamtur01/sshd   latest  286b8a745bc2    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 1.31: Using the docker history command

```
$ sudo docker history 286b8a745bc2
IMAGE         CREATED        CREATED BY ↩
                                     SIZE
286b8a745bc2  19 minutes ago /bin/sh -c #(nop) CMD [/bin/sh -c /↩
  usr/sbin/s 0 B
c7b715435fc6  19 minutes ago /bin/sh -c #(nop) EXPOSE map[22/tcp↩
  :{}]       0 B
6b7732845bd6  19 minutes ago /bin/sh -c echo "root:password" | ↩
  chpasswd    655 B
6b494658fda7  20 minutes ago /bin/sh -c mkdir /var/run/sshd ↩
                0 B
9cf923171082  20 minutes ago /bin/sh -c apt-get install -y ↩
  openssh-server  42.38 MB
. . .
```

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

Launching a container from our new image

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

Listing 1.32: Launching a container from our new image

```
$ sudo docker run -d -p 22 --name sshd jamtur01/sshd \
/usr/sbin/sshd -D
6751b94bb5c001a650c918e9a7f9683985c3eb2b026c2f1776e61190669494a8
```

Here I've launched a new container called `sshd` 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 SSH daemon.

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

- Docker can randomly assign a high port from the range `49000` to `49900` on the Docker host that maps to port 22 on the container.
- You can specify a specific port on the Docker host that maps to port 22 on the container.

This will open a random port on the Docker host that will connect to port 22 on the Docker container.

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

Listing 1.33: Viewing the Docker port mapping

```
$ sudo docker ps -l
CONTAINER ID  IMAGE                ... PORTS                  ↩
  NAMES
6751b94bb5c0  jamtur01/sshd:latest ... 0.0.0.0:49154->22/tcp  ↩
  sshd
```

We can see that port 49154 is mapped to the container port of 22. We can get the same information with the `docker port` command.

Listing 1.34: The docker port command

```
$ sudo docker port 6751b94bb5c0 22
0.0.0.0:49154
```

We've specified the container ID and the container port for which we'd like to see the mapping, `22`, and it has returned the mapped port, `49154`.

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

Listing 1.35: Exposing a specific port with -p

```
$ sudo docker run -d -p 22:22 --name sshd jamtur01/sshd \
/usr/sbin/sshd -D
```

This will bind port 22 on the container to port 22 on the local host. Obviously, it's important to be wary of this direct binding: if you're running multiple containers, only one container can bind a specific port on the local host. This can limit Docker's flexibility.

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

Listing 1.36: Binding to a different port

```
$ sudo docker run -d- p 2222:22 --name sshd jamtur01/sshd \
/usr/sbin/sshd -D
```

This would bind port 22 on the container to port 2222 on the local host.

We can also bind to a specific interface.

Listing 1.37: Binding to a specific interface

```
$ sudo docker run -d -p 127.0.0.1:22:22 --name sshd jamtur01/sshd↩
  \
/usr/sbin/sshd -D
```

Here we've bound port 22 of the container to port 22 on the `127.0.0.1` interface on the local host. We can also bind to a random port using the same structure.

Listing 1.38: Binding to a random port on a specific interface

```
$ sudo docker run -d -p 127.0.0.1::22 --name sshd jamtur01/sshd \
/usr/sbin/sshd -D
```

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 22 on the container.

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

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

Listing 1.39: Exposing a port with docker run

```
$ sudo docker run -d -P --name sshd jamtur01/sshd \
/usr/sbin/sshd -D
```

This would expose port 22 on a random port on our local host. It would also expose any additional ports we had specified with other `EXPOSE` instructions in the `Dockerfile` that built our image.

TIP You can find more information on port redirection here.

With this port number, we can now SSH into the running container using the IP address of our local host on `127.0.0.1`.

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

Listing 1.40: Connecting to the container via SSH

```
$ ssh -p 49154 root@10.0.2.15
The authenticity of host '[10.0.2.15]:49154 ([10.0.2.15]:49154)' ↩
  can't be established.
. . .
root@10.0.2.15's password:
Welcome to Ubuntu 12.04 LTS (GNU/Linux 3.8.0-19-generic x86_64)
root@bf42aadc7f09:~#
```

Now we've got a Docker-based SSH server.

Dockerfile instructions

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

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

CMD

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

Listing 1.41: Specifying a specific command to run

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

This would be articulated in the `Dockerfile` as:

Listing 1.42: Using the CMD instruction

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

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

Listing 1.43: 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 1.44: 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 1.45: 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 1.46: 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.

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 1.47: Specifying an ENTRYPOINT

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

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

Listing 1.48: Specifying an ENTRYPOINT parameter

```
ENTRYPOINT ["/usr/sbin/sshd", "-d"]
```

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 and launch a new container from our `jamtur01/sshd` image.

Listing 1.49: Using docker run with ENTRYPOINT

```
$ sudo docker build -t="jamtur01/sshd" .
. . .
$ sudo docker run -d jamtur01/sshd -D
```

As we can see, we've rebuilt our image and then launched a detached container. We specified the argument `-D`; this argument will be passed to the command specified in the `ENTRYPOINT` instruction, which will thus become `/usr/sbin/sshd -D`. This command would then launch the SSH daemon in the foreground and leave the container running as an SSH 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 1.50: Using ENTRYPOINT and CMD together

```
ENTRYPOINT ["/usr/sbin/sshd"]
CMD ["-T"]
```

Now when we launch a container, any option we specify will be passed to the SSH daemon; for example, we could specify `-D` as we did above to run the daemon in the foreground. If we don't specify anything to pass to the container, then the `-T` is passed by the `CMD` instruction and returns the SSH daemon configuration `/usr/sbin/sshd -T`.

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 1.51: 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 1.52: Overridding 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 1.53: 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 1.54: Prefixing a RUN instruction

```
RUN gem install unicorn
```

would be executed as:

Listing 1.55: Executing with an ENV prefix

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

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

Listing 1.56: Persisent 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 1.57: Runtime environment variables

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

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

USER

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

Listing 1.58: 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.

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

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

VOLUME

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

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

This allows us to add data (like source code), a database, or other content into an image without committing it to the image and allows us to share that data betweeen 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 1.59: Using the VOLUME instruction

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

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

Or we can specify multiple volumes by specifying an array:

Listing 1.60: Using multiple VOLUME instructions

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

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

ADD

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

Listing 1.61: 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. Any filename or directory must be specified relative to the build directory; for example:

Listing 1.62: Specifying a directory relative to the build

```
ADD ../app /opt/
```

This instruction will recursively copy the `app` directory, located in the directory above the build directory, to the `/opt/` directory. 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 1.63: 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 1.64: URL 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 1.65: Using the COPY instruction

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

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

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

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

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

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

ONBUILD

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

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

Listing 1.66: Adding ONBUILD instructions

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

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

Listing 1.67: 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 1.68: A new ONBUILD image Dockerfile

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

Now we'll build this image.

Listing 1.69: 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/html/` 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 1.70: The webapp Dockerfile

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

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

Listing 1.71: Building our webapp image

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

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

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

Listing 1.72: Trying to push a root image

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

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

Listing 1.73: Pushing a Docker image

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

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

We can now see our uploaded image on the Docker Hub.

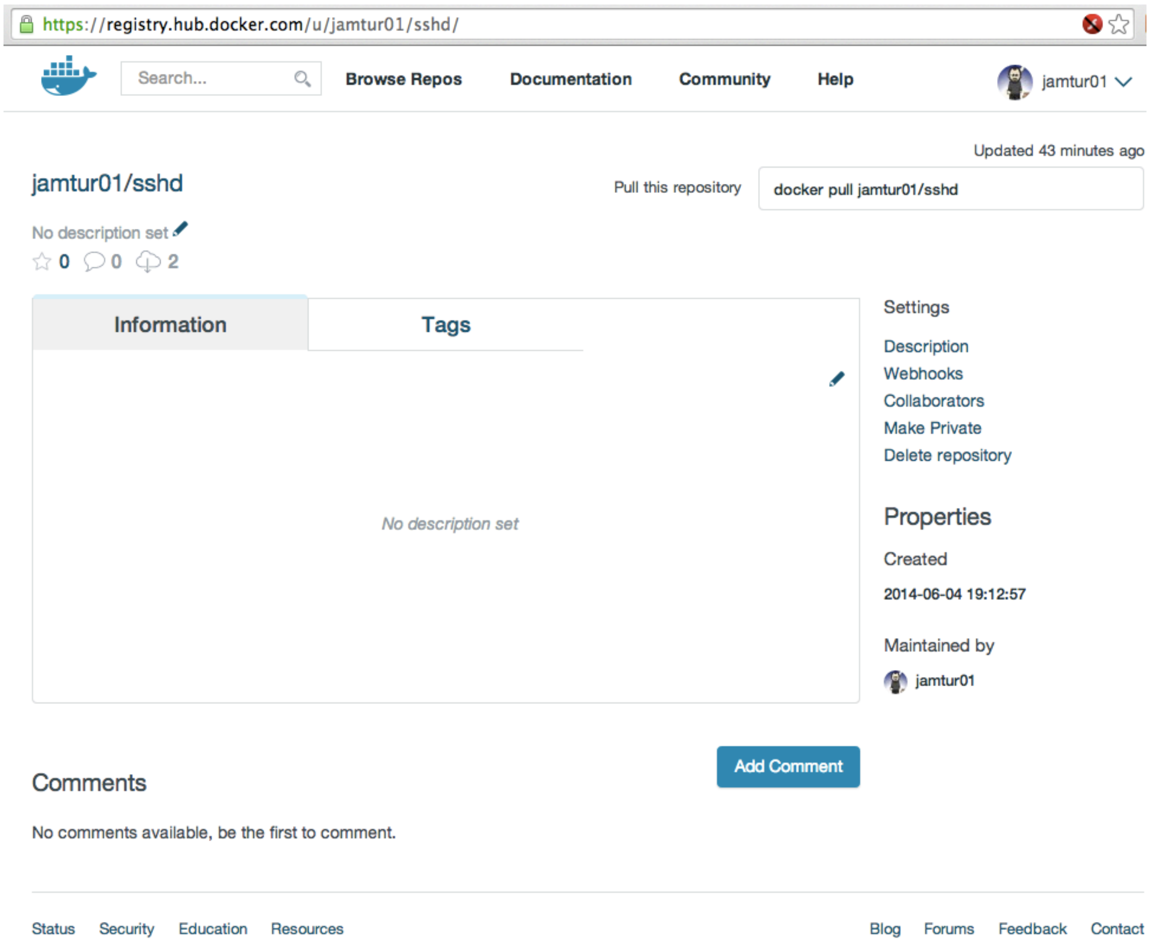


Figure 1.4: Your image on the Docker Hub.

TIP You can find documentation and more information on the features of the Docker Hub here.

Automated Builds

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

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

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

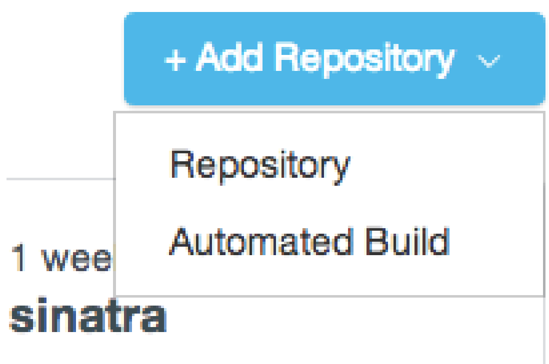


Figure 1.5: The Add Repository button.

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

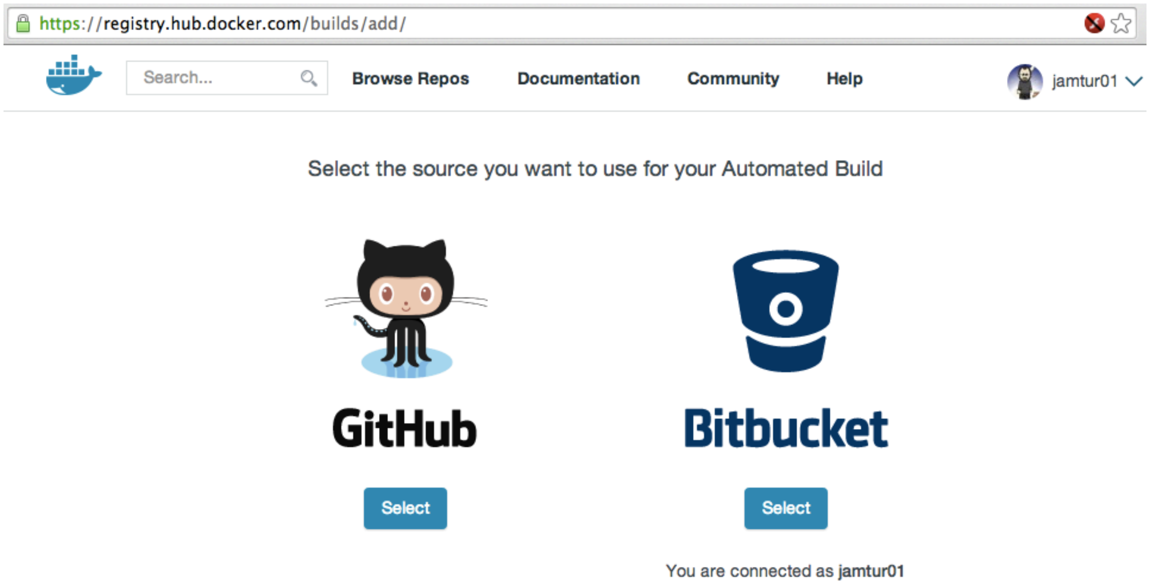


Figure 1.6: Account linking options.

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

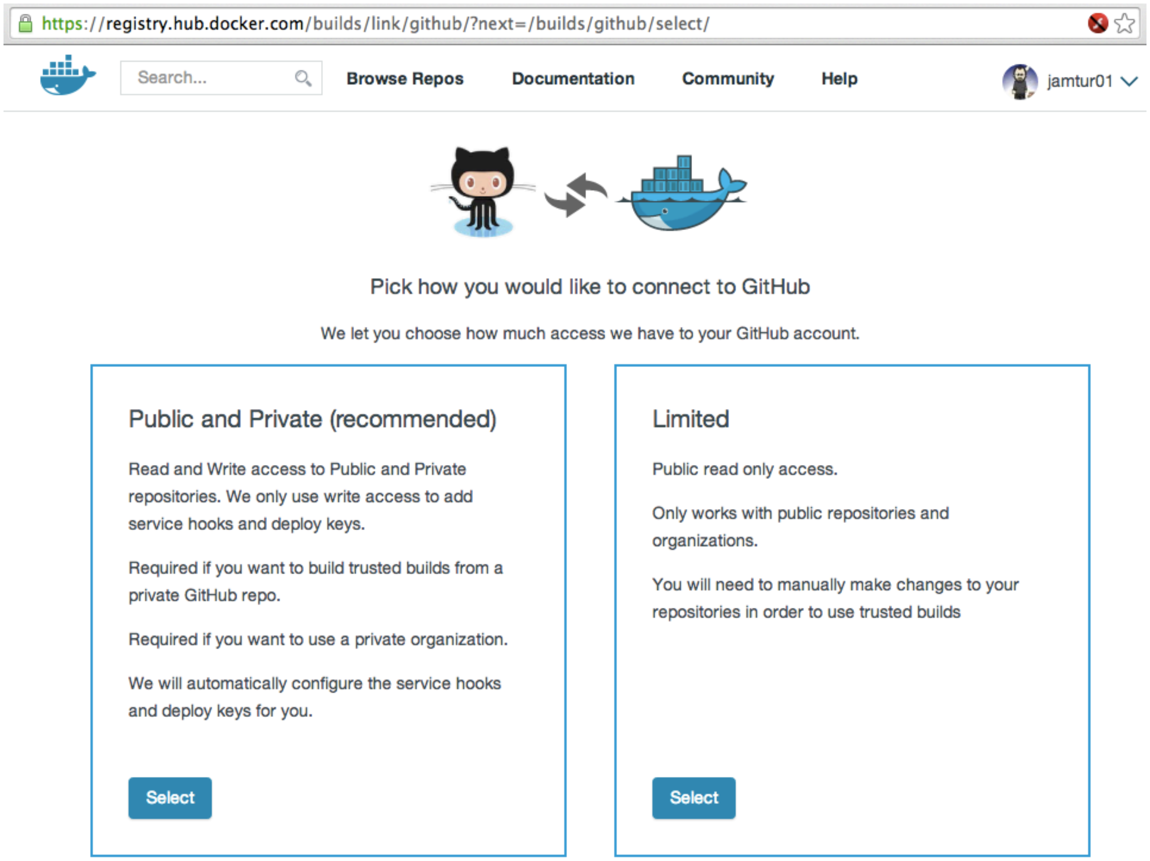


Figure 1.7: Linking your GitHub account

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

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

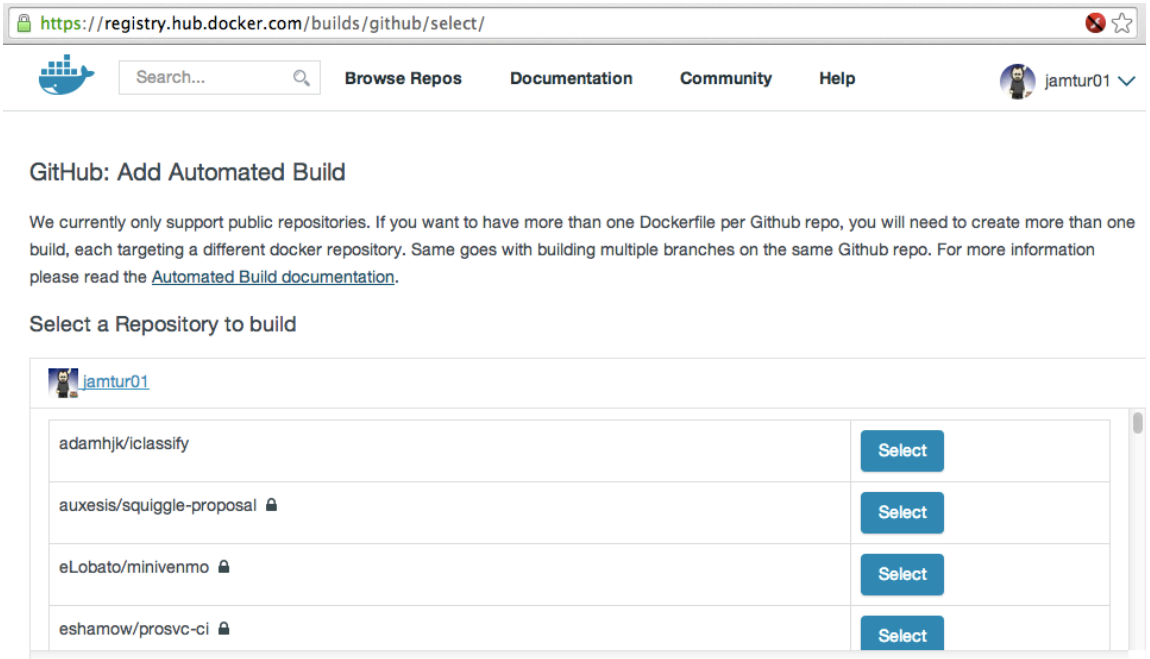


Figure 1.8: Selecting your repository.

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

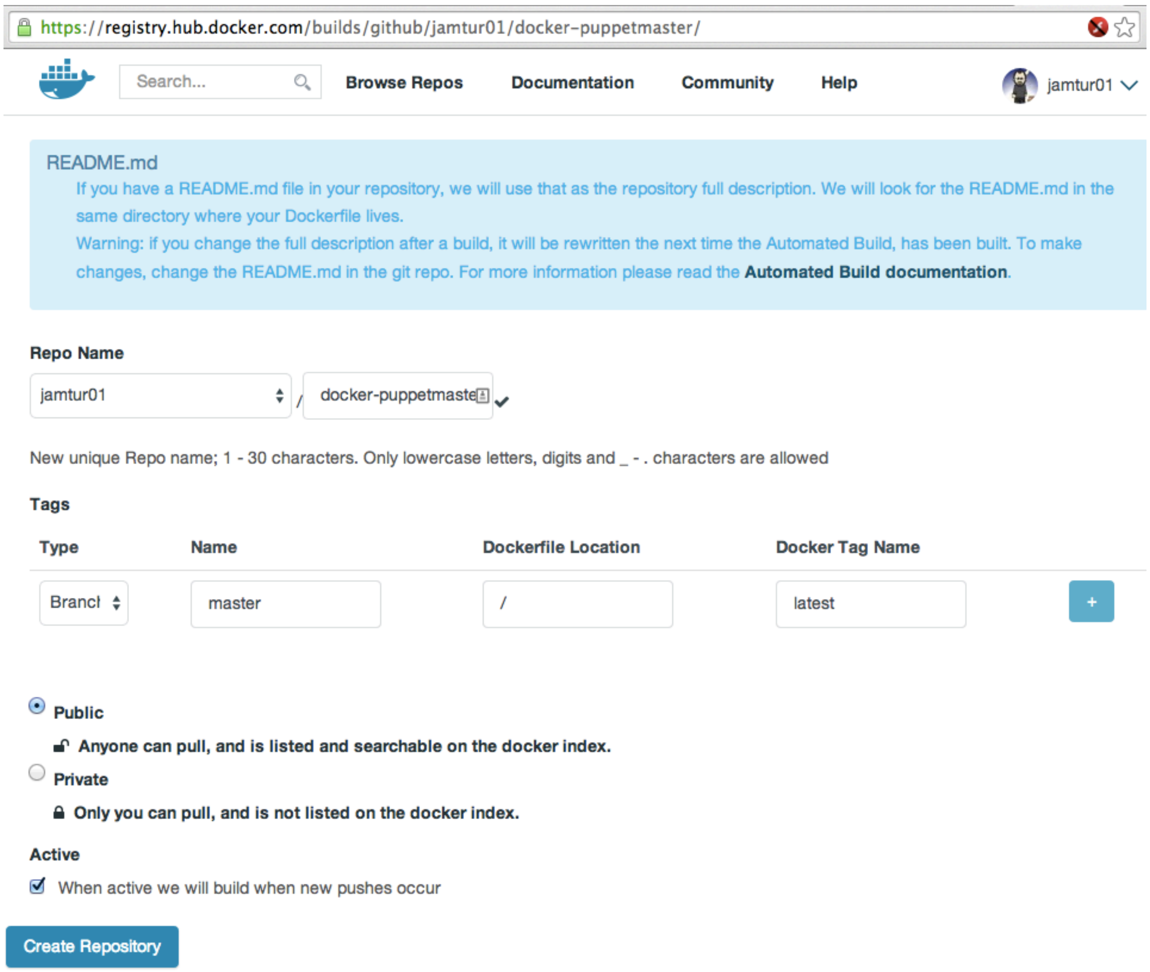


Figure 1.9: Configuring your Automated Build.

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

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

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

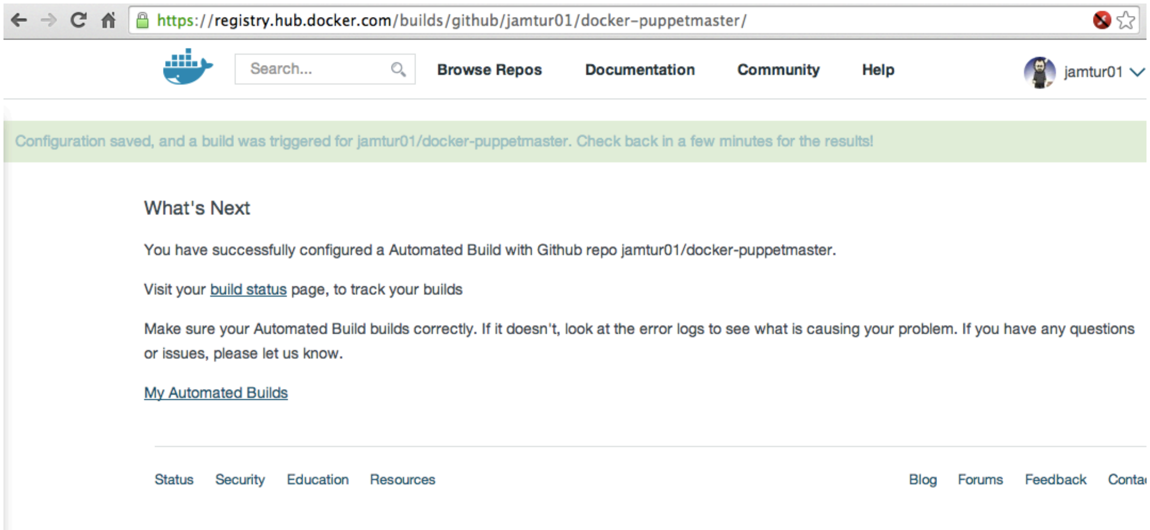


Figure 1.10: Creating your Automated Build.

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

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

Deleting an image

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

Listing 1.74: Deleting a Docker image

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

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

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

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

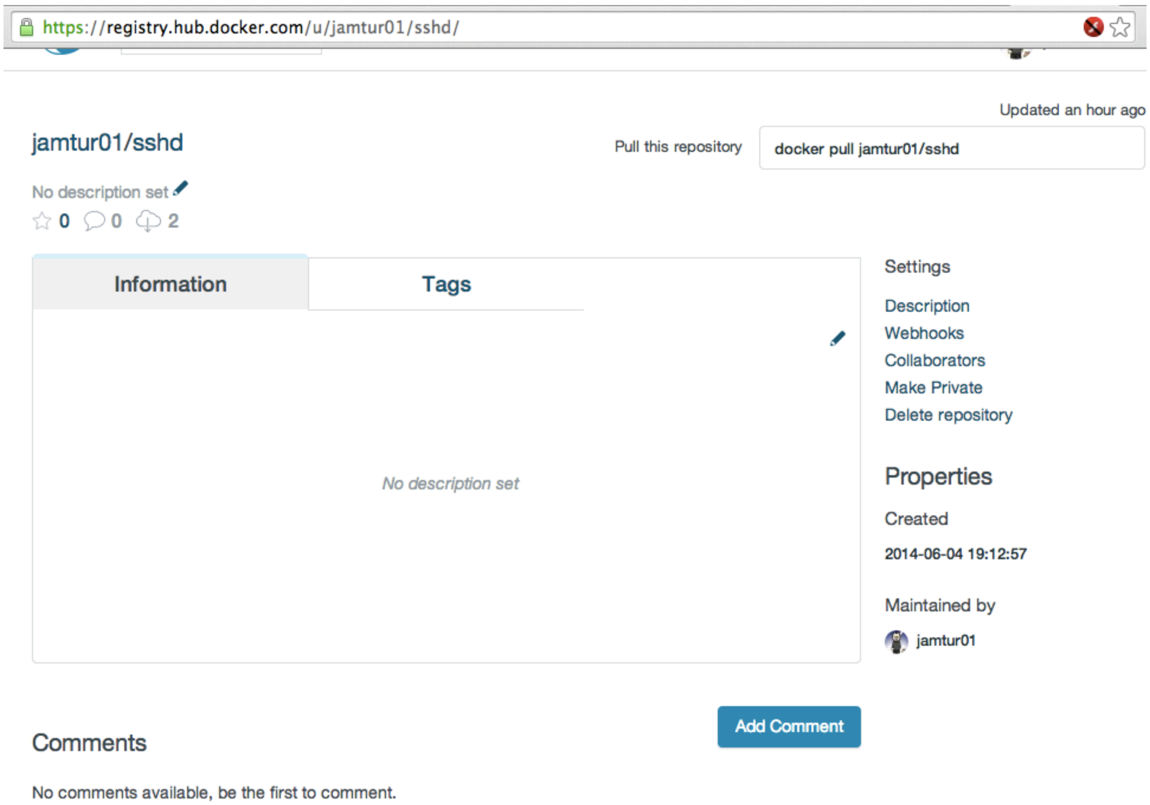


Figure 1.11: Deleting a repository.

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

Listing 1.75: 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 1.76: Deleting all images

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

Running your own Docker registry

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

- Make use of private repositories on the Docker Hub.
- Run your own registry behind the firewall.

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

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

Running a registry from a container

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

Listing 1.77: Running a container-based registry

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

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

Testing the new registry

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

Listing 1.78: Listing the jamtur01 sshd Docker image

```
$ sudo docker images jamtur01/sshd
REPOSITORY      TAG     ID              CREATED         SIZE
jamtur01/sshd   latest  286b8a745bc2    24 seconds ago  12.29 kB ↩
  (virtual 326 MB)
```

Next we take our image ID, `286b8a745bc2`, 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 1.79: Tagging our image for our new registry

```
$ sudo docker tag 8dbd9e392a96 docker.example.com:5000/jamtur01/↩
  sshd
```

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

Listing 1.80: Pushing an image to our new registry

```
$ sudo docker push docker.example.com:5000/jamtur01/sshd
The push refers to a repository [docker.example.com:5000/jamtur01↩
  /sshd] (len: 1)
Processing checksums
Sending image list
Pushing repository docker.example.com:5000/jamtur01/sshd (1 tags)
Pushing 8↩
  dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c
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 1.81: Building a container from our local registry

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

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

Alternative Indexes

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

Quay

The Quay service provides a private hosted registry that allows you to upload both public and private containers. Unlimited public repositories are currently free. Private repositories are available in a series of scaled plans.

Orchard

The Orchard service offers a private registry service, but also provides Docker container hosting in the Cloud. Pricing comprises a flat monthly fee and a price per hour for running containers.

Summary

In this chapter, we've seen how to use and interact with Docker images and the basics of modifying, updating, and uploading images to the Docker Index. We've also learnt 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.

Index

Thanks! I hope you enjoyed the book.

ISBN 978-0-9888202-0-3