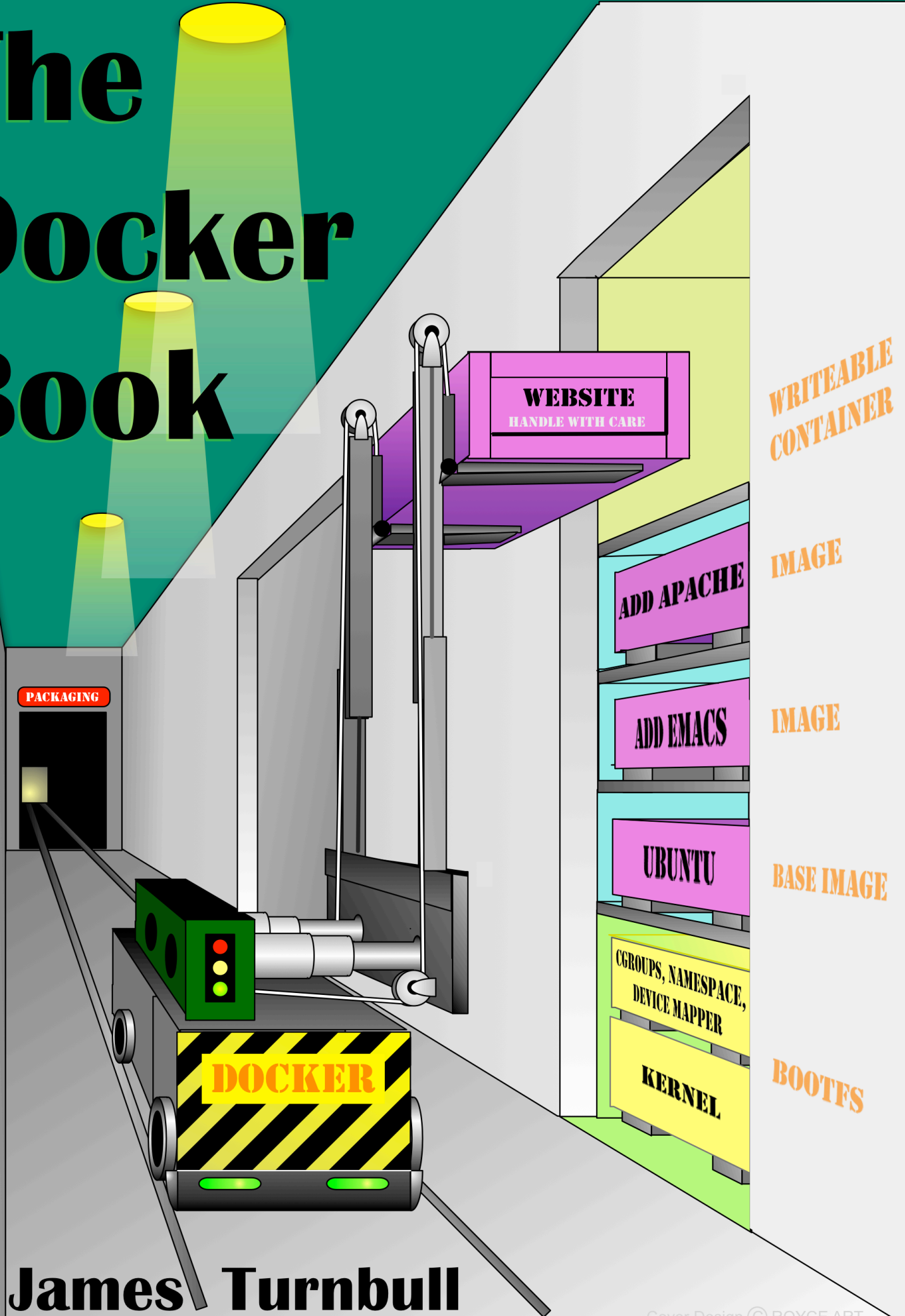


The Docker Book



by James Turnbull

The Docker Book

James Turnbull

February 3, 2014

Version: v0.0.1 (b3dcb4f)

Website: [The Docker Book](#)

Contents

	Page
List of Figures	iii
List of Listings	vi
Chapter 1 Working with Docker images and repositories	1
What is a Docker image?	1
Listing the Docker images	4
Docker repositories	5
Searching for images	6
Pulling down images	7
Making changes to images	8
Pushing a container to the repository	10
Creating an account	10
Pushing the image	11
Deleting an image	13
Building containers with a Dockerfile	14
Building the container from our Dockerfile	18
Dockerfile instructions	23
Trusted builds	30
Running your own Docker Index	36
Running the Index from a container	36
Installing the Docker Index manually	36
Configuring the Docker Index	39
Launching the Docker Index	41
Testing the new Index	42
Production configuration	44

Alternative Indexes	44
Quay	44
Orchard	45
Summary	45
Index	46

List of Figures

1.1 The Docker filesystem layers	3
1.2 The Docker index site	4
1.3 Listing Docker images	5
1.4 Creating a Docker index account.	11
1.5 Your image on the Index.	13
1.6 The Trusted Build tab.	31
1.7 Linking to your GitHub account.	31
1.8 Authorizing GitHub access.	32
1.9 Selecting a GitHub repository.	33
1.10Configuring the trusted build.	34
1.11The Trusted Build status window.	35
1.12The Docker Index home page.	42

Listings

1.1	Revisiting creating a basic Docker container	1
1.2	Creating a container based on a tag	6
1.3	Narrowing down the image list	6
1.4	Searching for images	6
1.5	Pulling an image	7
1.6	Creating a Docker container from the Puppet master image	7
1.7	Creating a custom container	8
1.8	Checking the custom container	8
1.9	Committing the custom container	9
1.10	Committing another custom container	9
1.11	Trying to push a root image	11
1.12	Pushing a Docker image	12
1.13	Storing Docker credentials locally	12
1.14	Deleting a Docker image	13
1.15	Deleting multiple Docker images	14
1.16	Creating a sample repository	14
1.17	A simple Dockerfile	15
1.18	The RUN instruction in exec form	16
1.19	Exposing a port with docker run	16
1.20	Exposing a specific port with -p	17
1.21	Binding to a different port	17
1.22	Binding to a specific interface	17
1.23	Binding to a random port on a specific interface	17
1.24	Exposing a port with docker run	18
1.25	Running the Dockerfile	18
1.26	Tagging a build	19

1.27 A template Dockerfile	20
1.28 Listing our new Docker image	21
1.29 Launching a container from our new image	21
1.30 Showing the exposed ports	22
1.31 The docker port command	22
1.32 Connecting to the container via SSH	22
1.33 Specifying a specific command to run	23
1.34 Using the CMD instruction	23
1.35 Passing parameters to the CMD instruction	24
1.36 Overriding CMD instructions in the Dockerfile	24
1.37 Launching a container with a CMD instruction	25
1.38 Overriding a command locally	25
1.39 Specifying an ENTRYPOINT	26
1.40 Specifying an ENTRYPOINT parameter	26
1.41 Using docker run with ENTRYPOINT	26
1.42 Using ENTRYPOINT and CMD together	27
1.43 Setting an environment variable in Dockerfile	27
1.44 Prefixing a RUN instruction	27
1.45 Executing with an ENV prefix	28
1.46 Using the USER instruction	28
1.47 Using the VOLUME instruction	28
1.48 Using the ADD instruction	29
1.49 Specifying a directory relative to the build	29
1.50 URL as the source of an ADD instruction	29
1.51 URL as the source of an ADD instruction	29
1.52 Running a container-based index	36
1.53 Ubuntu prerequisites for the Docker Index	37
1.54 Installing EPEL for RHEL/CentOS 6	37
1.55 Red Hat-based prerequisites for the Docker Index	37
1.56 Downloading the Docker Index registry application	38
1.57 Changing into the Docker Index directory	38
1.58 Using Pip to install the Docker Index	38
1.59 Copying the Docker Index sample configuration	39
1.60 The Docker Index configuration file	39
1.61 Setting the Docker Index flavor	40

1.62 Running Docker Index locally	41
1.63 Identifying the Ubuntu image ID	42
1.64 Tagging our image for our new registry	43
1.65 Pushing an image to our new index	43
1.66 Building a container from our local registry	44

Chapter 1

Working with Docker images and repositories

In the last chapter we learnt how to install Docker on a host and run a basic Docker container from an image using the `docker run` command.

Listing 1.1: Revisiting creating a basic Docker container

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

This launched a shell inside a new container built from the `ubuntu` image. In this chapter we're going to learn a lot more about Docker images, what they are, how to manage them, how to modify them and how to create and store your own images.

What is a Docker image?

Let's continue our journey with Docker by learning a bit more about Docker images and Docker itself. A Docker environment is made up of filesystems layered over each other. At the base is a boot filesystem, `bootfs`, which resembles the typical Linux/Unix boot file system and contains the bootloader and the kernel. A Docker

Chapter 1: Working with Docker images and repositories

user will never interact with the boot filesystem, indeed when a container has booted the kernel 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 and indeed Docker next layers a root filesystem, `rootfs`, on top of the boot filesystem. This `rootfs` can one or more operating systems, for example 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. 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. Docker calls each of these filesystems images. Images can be layered on top of one another. The image below is called the 'parent' image, until you reach the bottom of the image stack where the final image is called the base image. Finally, 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 but it is best represented by a diagram:

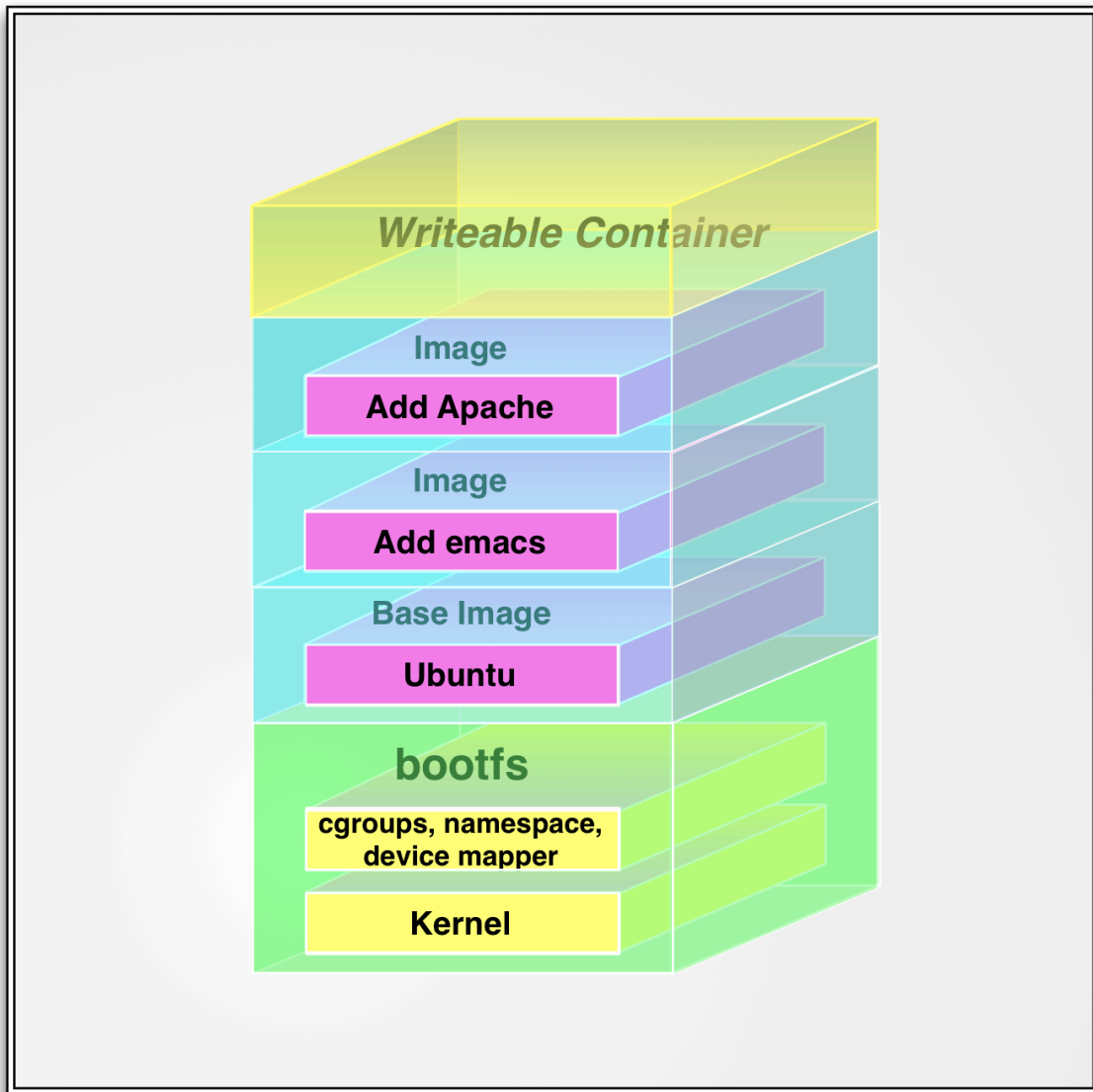


Figure 1.1: The Docker filesystem layers

When Docker first starts 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. Docker calls the top read-write layer and all the read-only layers combined a union

file system.

It is this combination that is one of the features that makes Docker so powerful. Each read-only image layer is read-only and 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 and create containers that we can build our applications and services on.

Listing the Docker images

Let's get started with Docker images by looking at what images are available to us on this host using the `docker images` command. Docker images are stored on [the Docker index](#) and downloaded via the `docker pull` command.

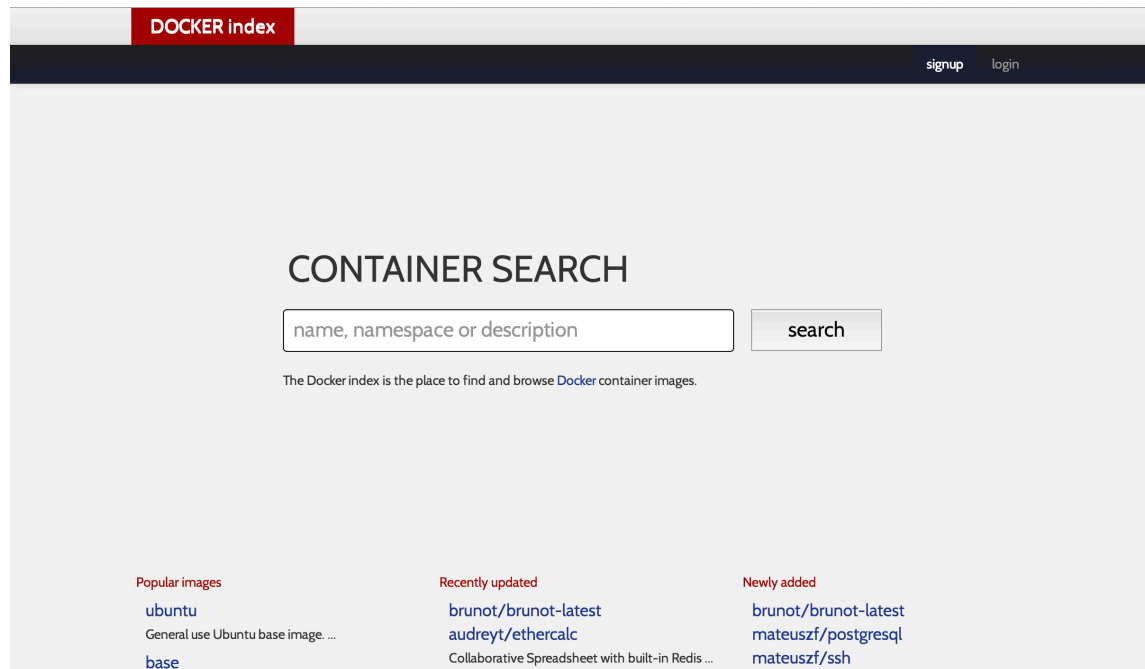


Figure 1.2: The Docker index site

This is a registry and image store and you can interact with it using the `docker images` command.

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ubuntu	12.04	8dbd9e392a96	6 months ago	131.5 MB (virtual 131.5 MB)
ubuntu	latest	8dbd9e392a96	6 months ago	131.5 MB (virtual 131.5 MB)
ubuntu	precise	8dbd9e392a96	6 months ago	131.5 MB (virtual 131.5 MB)
ubuntu	12.10	b750fe79269d	7 months ago	24.65 kB (virtual 180.1 MB)
ubuntu	quantal	b750fe79269d	7 months ago	24.65 kB (virtual 180.1 MB)

Figure 1.3: Listing Docker images

Docker repositories

We can see that we've got images installed from two repositories: `ubuntu` and `base`. There are two types of repositories: user repositories which contains images contributed by Docker users and top-level repositories which are controlled by the people behind Docker.

TIP In addition to the `ubuntu` and `base` images there is also a `busybox` image available [here](#).

A user repository takes the form of a username and a repository name, for example: `jamtur01/puppet`. A top-level repository is managed by the Docker team. They can be recognized by not having the username and slash in their names. The top-level repositories also represent a commitment from the Docker team 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 the Docker team.

Notice something else about the list? The `ubuntu` and `base` top-level repositories each contain two images, with the IDs `b750fe79269d` (an Ubuntu 12.10 image) and `8dbd9e392a96` (an Ubuntu 12.04 image), but the list contains nine entries? What's happening is that each image is being listed by the tags applied to it, for example `12.10`, `12.04`, `quantal` or `precise` and so on. We can refer to a specific image by suffixing the repository name with a colon and a tag name, like so: `ubuntu:12.04`. I can then create a Docker container based on a specific tag:

Listing 1.2: Creating a container based on a tag

```
$ sudo docker run -i -t ubuntu:12.04 /bin/bash
```

We can also narrow down the list by specifying the specific repository with the `docker images` command like so:

Listing 1.3: Narrowing down the image list

```
$ sudo docker images ubuntu
```

Searching for images

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

Listing 1.4: Searching for images

```
$ sudo docker search puppet
Found 2 results matching your query ("puppet")
NAME                                DESCRIPTION
wfarr/puppet-module-builder
jamtur01/puppetmaster
```

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

Here we've searched the Docker index site for the term puppet. It'll search images and repository names as well as image descriptions for the required keyword.

Pulling down images

If you find an image you want to use you can then use the `docker pull` command

Listing 1.5: Pulling an image

```
$ sudo docker pull jamtur01/puppetmaster
Pulling repository jamtur01/puppetmaster from https://index.docker.io/v1
Pulling image 27cf784147099545 () from jamtur01/puppetmaster
Pulling 27cf784147099545 metadata
Pulling 27cf784147099545 fs layer
Downloading 94.86 MB/94.86 MB (100%)
. . .
```

This will pull down the `jamtur01/puppetmaster` image (which by the way contains a pre-installed Puppet master). You can then use the image to build a new container.

Listing 1.6: Creating a Docker container from the Puppet master image

```
$ sudo docker run -i -t jamtur01/puppetmaster /bin/bash
root@091268172c10:/# facter
architecture => amd64
augeasversion => 0.10.0
. . .
root@091268172c10:/# puppet --version
2.7.18
```

Here you can see we've launched a new container from our `jamtur01/puppetmaster` image and run `Factor` (Puppet's inventory application) which was pre-installed on our image. We've also run the `puppet` binary to confirm it is also installed.

Making changes to images

So we've seen that we can pull down pre-prepared images with custom contents. How do we go about creating our own images and updating and managing them? Thankfully it is an incredibly simple process. Let's start by creating a container from the `ubuntu` image we've used in the past.

Listing 1.7: Creating a custom container

```
$ sudo docker run -i -t ubuntu /bin/bash
root@4aab3ce3cb76:/# apt-get install apache2
. . .
```

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

NOTE We could also update an existing image but in this case our container is based on the `ubuntu` image which is managed by the Docker Inc team and not editable by us.

First we exit from our container and then list all the containers to confirm we're working with the right container.

Listing 1.8: Checking the custom container

Chapter 1: Working with Docker images and repositories

```
root@4aab3ce3cb76:/# exit
$ sudo docker ps -a
. . .
4aab3ce3cb76          ubuntu:12.04          /bin/bash ↵
                About an hour ago    Exit 0
```

Next we use the `docker commit` command to save a new image.

Listing 1.9: Committing the custom container

```
$ sudo docker commit 4aab3ce3cb76 jamtur01/apache2
8ce0ea7a1528
$ sudo docker images
. . .
jamtur01/apache2      latest          8ce0ea7a1528    ↵
    13 seconds ago    90.63 MB (virtual 222.1 MB)
```

You can see we've used the `docker commit` command and specified the ID of the container we've just changed as well as a target. The `docker commit` command only commits the differences between the image the container was created from and the current state of the container.

TIP You can also build containers from scratch. See an example [here](#).

We can also provide some more data when committing your image, for example:

Listing 1.10: Committing another custom container

```
$ sudo docker commit -m="A new custom image" -author="James ↵
    Turnbull" -run='{ "PortSpecs": ["22", "80"], "Env": ["HTTP_PORT↵
    =8080"] }' 4aab3ce3cb76 jamtur01/apache2 apache2
8ce0ea7a1528
. . .
```

While committing our new image we've specified some options. 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 document the author of the image. Additionally, we've used the `-run` option. This option allows you to specify a hash of settings and commands to be run when a container launched from the image. In this case we've specified the `PortSpecs` option which exposes one or more of the container's ports publicly. We've provided an array containing ports 22 and 80 that will be exposed on the container.

NOTE We'll see more about exposing and using network ports on containers later in this chapter.

We've also provided the `Env` option which creates an environment variable, here `HTTP_PORT`, when the container is launched.

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

Pushing a container to the repository

Once we've got our updated image we can then upload it to the Docker index site. This allows us to make it available for use when needed by anyone who wants to download it.

Creating an account

In order to upload an image though we first need an account on the Docker index site. You can join Docker index [here](#).

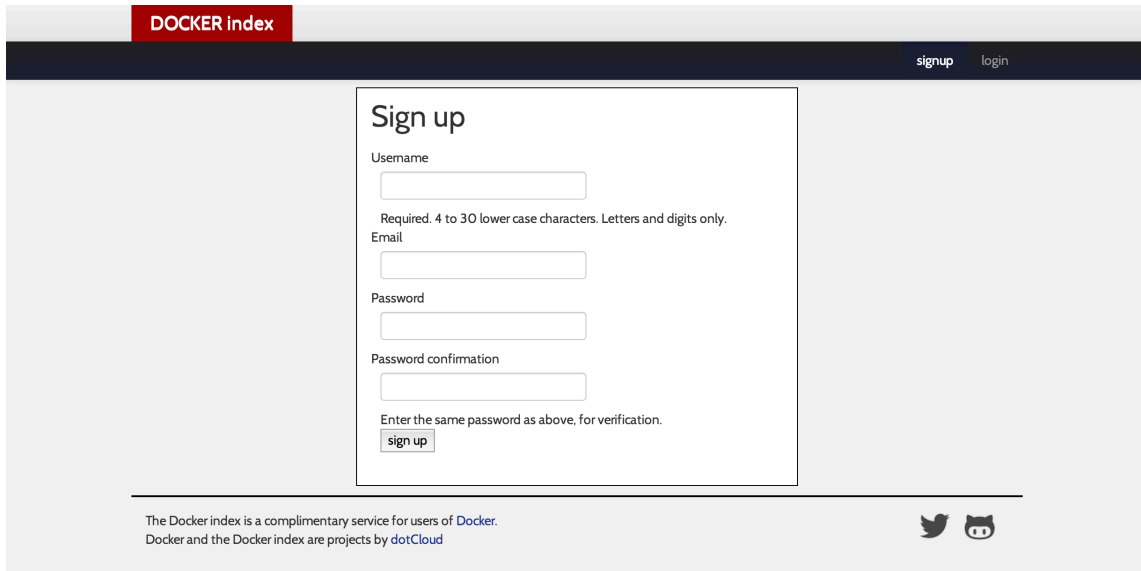
The screenshot shows the Docker index sign-up page. At the top, there is a dark blue header with the 'DOCKER index' logo on the left and 'signup' and 'login' links on the right. The main content area is light gray and contains a white box titled 'Sign up'. Inside this box, there are four input fields: 'Username', 'Email', 'Password', and 'Password confirmation'. Below the 'Password' field, there is a small text requirement: 'Required. 4 to 30 lower case characters. Letters and digits only.' Below the 'Password confirmation' field, there is a note: 'Enter the same password as above, for verification.' At the bottom of the form is a 'sign up' button. Below the sign-up box, there is a footer section with text stating 'The Docker index is a complimentary service for users of Docker.' and 'Docker and the Docker index are projects by dotCloud', along with Twitter and GitHub icons.

Figure 1.4: Creating a Docker index account.

Create an account and verify your email address. Keep note of your user ID and password as you'll need these to push an image to the Index.

Pushing the image

Now we have a Docker Index account we can push our newly updated image to the Index. We do this using the `docker push` command.

Let's try a push now.

Listing 1.11: Trying to push a root image

```
$ sudo docker push apache2
Username (): jamtur01
Password:
Email (): james@example.com
Login Succeeded
```

```
2013/07/01 18:34:47 Impossible to push a "root" repository. ↵  
Please rename your repository in <user>/<repo> (ex: jamtur01/↵  
apache2)
```

You can see we've been prompted for the username and password we've just created and we've logged in fine. But what's gone wrong here? Well we've tried to push our image to the repository `apache2` but Docker knows this is a root repository, which are only managed by the Docker Inc team and rejects our attempt to write to it. Let's try again.

Listing 1.12: Pushing a Docker image

```
$ sudo docker push jamtur01/apache2  
The push refers to a repository [jamtur01/apache2] (len: 1)  
Processing checksums  
Sending image list  
Pushing repository jamtur01/apache2 to registry-1.docker.io (1 ↵  
tags)  
. . .
```

Here we were not prompted for our username and password because Docker has remembered our logged in session. In addition to this we could also store our Docker credentials locally rather than having to be prompted for them when we push images. Our credentials are stored in a JSON file called `.dockercfg` located in our home directory. We can use the `docker login` command to create entries in this file.

Listing 1.13: Storing Docker credentials locally

```
$ sudo docker login
```

By default Docker will create credentials for the [central Docker Index](#).

NOTE You can also store credentials for private repositories in this file too

by specifying the URL of the index you wish to connect with the `docker login` command.

This time our push has worked though and we've written to a user repository, `jamtur01/apache2`. You would write to your own user ID, which you created earlier, and to an appropriately named image, for example `youruser/yourimage`. You can now see your uploaded image on the Docker index.

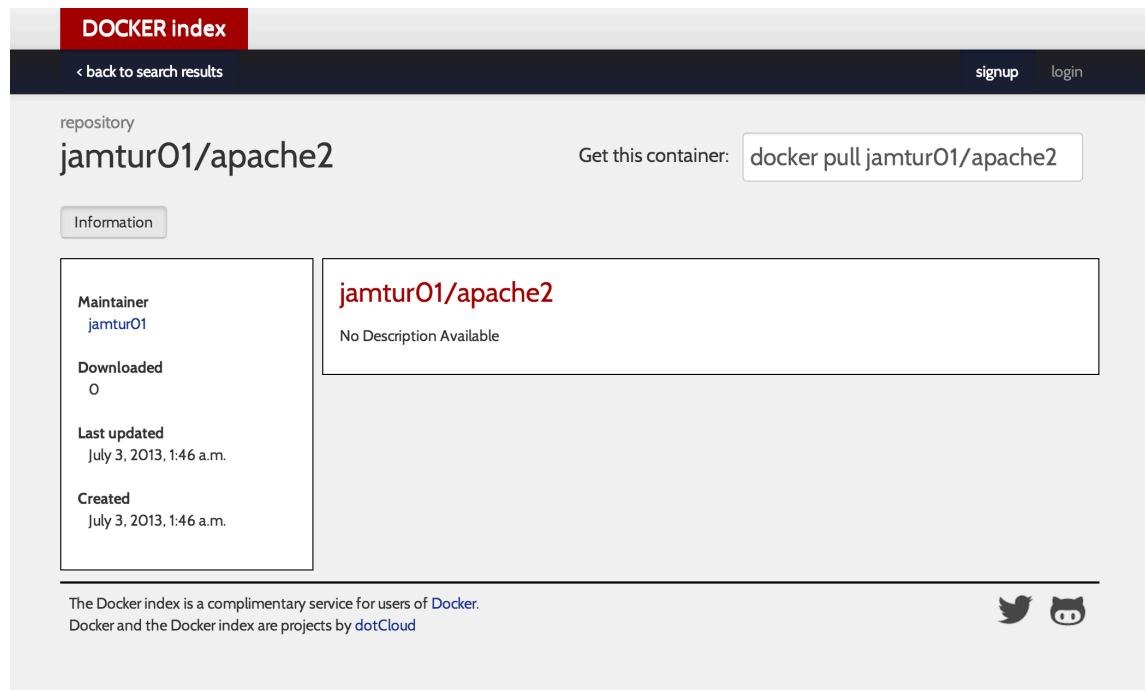


Figure 1.5: Your image on the Index.

Deleting an image

Just like building your image you can also delete those images when you don't need them anymore. To do this we use the `docker rmi` command.

Listing 1.14: Deleting a Docker image

```
$ sudo docker rmi jamtur01/apache2
Untagged: 06c6c1f81534
Deleted: 06c6c1f81534
Deleted: 9f551a68e60f
Deleted: 997485f46ec4
Deleted: a101d806d694
Deleted: 85130977028d
```

Here we've deleted the `jamtur01/apache2` image. You can delete more than one image by specifying a list on the command line.

Listing 1.15: Deleting multiple Docker images

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

Or like the `docker rm` command cheat we saw in Chapter 2 you can do the same with the `docker rmi` command: `docker rmi $(docker images -a -q)`.

Building containers with a Dockerfile

You can also build images using a definition file called a Dockerfile. You place a Dockerfile in the root of a repository. It contains a basic DSL for building Docker images. You can then use the `docker build` command to build a new image from this file.

TIP The team at Docker Inc have also published a Dockerfile tutorial to help you learn how to build Dockerfile's [here](#).

Let's now create a repository and a sample Dockerfile.

Listing 1.16: Creating a sample repository

```
$ mkdir dockertest
$ cd dockertest
$ touch Dockerfile
```

Now let's look at an example of a simple Dockerfile. We're going to create a Docker container that will act as a simple SSH server.

Listing 1.17: A simple Dockerfile

```
# VERSION 0.0.1
FROM ubuntu
MAINTAINER James Turnbull "james@example.com"
RUN echo "deb http://archive.ubuntu.com/ubuntu precise main ←
    universe" > /etc/apt/sources.list
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 should be in upper-case and be followed by an argument, for example FROM ubuntu. The Dockerfile is processed from the top down in order so you should order instructions accordingly.

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. In our

Chapter 1: Working with Docker images and repositories

sample Dockerfile we've specified the ubuntu image as our base image.

Next we've specified the MAINTAINER instruction which tells Docker who the author of the image is.

We've followed these instructions with three RUN instructions. The RUN instruction executes commands on the current image and commits the results. In our example we're setting the root password, updating the installed APT repositories, installing the openssh-server package and then creating the /var/run/sshd directory.

By default the RUN instruction executes inside a shell using the command `\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 form like so:

Listing 1.18: The RUN instruction in exec form

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

Using this form we 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 to open a specific port on the container, in this case port 22. This doesn't mean you can access whatever service is running on port 22 on the container via that port. Docker randomly assigns a public port on the virtual host to map to port 22 on the container.

Using the EXPOSE instruction primes Docker to open this port but for security reasons it doesn't actually open the port automatically. To do this we need to specify the `-p` option to the `docker run` command when we spawn our container like so:

Listing 1.19: Exposing a port with docker run

```
$ sudo docker run -p 22 -i -t jamtur01/sshd /bin/bash
```

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

Chapter 1: Working with Docker images and repositories

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

Listing 1.20: Exposing a specific port with `-p`

```
$ sudo docker run -p 22:22 -i -t jamtur01/sshd /bin/bash
```

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 then only one container can bind a specific port on the local host. This can limit the flexibility of Docker.

To avoid this we can bind to a different port.

Listing 1.21: Binding to a different port

```
$ sudo docker run -p 2222:22 -i -t jamtur01/sshd /bin/bash
```

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

We can also bind to a specific interface.

Listing 1.22: Binding to a specific interface

```
$ sudo docker run -p 127.0.0.1:22:22 -i -t jamtur01/sshd /bin/↵  
bash
```

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 1.23: Binding to a random port on a specific interface

```
$ sudo docker run -p 127.0.0.1::22 -i -t jamtur01/sshd /bin/bash
```

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` commands 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 a shortcut, `-P`, since Docker version 0.7.0, that allows you to expose all ports you've specified in your Dockerfile.

Listing 1.24: Exposing a port with docker run

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

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.

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

Building the container 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 1.25: Running the Dockerfile

```
$ cd dockertest
$ sudo docker build -t jamtur01/sshd .
Uploading context 10240 bytes
Step 1 : FROM ubuntu
---> 8dbd9e392a96
Step 2 : MAINTAINER James Turnbull "james@example.com"
---> Running in d97e0c1cf6ea
---> 85130977028d
```

```
Step 3 : RUN echo "deb http://archive.ubuntu.com/ubuntu precise ↵
main universe" > /etc/apt/sources.list
---> Running in 85130977028d
---> a101d806d694
Step 4 : RUN apt-get update
---> Running in a101d806d694
---> 997485f46ec4
Step 5 : RUN apt-get install -y openssh-server
---> Running in ffca16d58fd8
---> 9f551a68e60f
Step 6 : RUN mkdir /var/run/sshd
---> Running in 9f551a68e60f
---> 4faf762fcbe3
Step 7 : RUN echo "root:password" | chpasswd
---> Running in 4faf762fcbe3
---> 254a39b9d511
Step 8 : EXPOSE 80
---> 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 like so:

Listing 1.26: Tagging a build

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

NOTE The trailing `.` that tells Docker to look in the local directory to find the Dockerfile.

You can see 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 is run individually and, where required, Docker has committed the result of the operation before outputting that final image ID.

As a result of each step being committed Docker is able to be really clever about building images. It treats these layers as a cache. If a previous image exists with the same parent and an instruction that has already been run then Docker will use this image to construct your new image rather than re-running any instructions. For example if you install a package with a RUN instructions it will use this image rather than building a new one. This capability works top down from your Dockerfile.

NOTE If you don't want to keep any layers you can run `docker build` with the `-rm` option which will remove all layers except the final image.

So it's a good idea to keep common instructions in a Dockerfile, for example adding a repo 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 like so:

Listing 1.27: A template Dockerfile

```
FROM      ubuntu
MAINTAINER James Turnbull "james@example.com"
RUN echo "deb http://archive.ubuntu.com/ubuntu precise main ←
    universe" > /etc/apt/sources.list
ENV REFRESHED_AT 2013-07-28
RUN apt-get update
```

This means that a new image built from the same parent ubuntu image will reuse the previous layers rather than have to rebuild them. I've specified the ENV

instruction to set an environment variable showing when the template was last updated. When I want to refresh the build I change the date and the `RUN apt-get update` instruction is re-run.

NOTE Deleting the previous image will also obviously invalidate the cache.

Now let's see our new image in our `docker images` list.

Listing 1.28: Listing our new Docker image

```
$ sudo docker images
```

REPOSITORY	TAG	ID	
	CREATED	SIZE	
jamtur01/sshd	latest	286b8a745bc2	24
	seconds ago	12.29 kB (virtual 326 MB)	

TIP You can see a full history of each commit and step taken to build an image using the `docker history` command.

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

Listing 1.29: Launching a container from our new image

```
$ sudo docker run -d jamtur01/sshd -p 22 /usr/sbin/sshd -D
db206a9fb848
```

Here I've launched a new container using the `docker run` command and the image ID 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 the `-p` option to expose port 22 on a random port on the local host. We can see what port it is specifically mapped to using the `docker ps` or `docker port` commands.

Listing 1.30: Showing the exposed ports

```
$ sudo docker ps 1bb7a52a0c40
```

ID	IMAGE	COMMAND	
CREATED	STATUS	PORTS	
1bb7a52a0c40	jamtur01/sshd:latest	/usr/sbin/sshd -D	5↔
minutes ago	Up 5 minutes	49168->22	

We can see that port 49168 is mapped to the container port of 22. We can get the same information with the `docker port` command.

Listing 1.31: The docker port command

```
$ sudo docker port 1bb7a52a0c40 22
49168
```

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, 49168.

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

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

Listing 1.32: Connecting to the container via SSH

```
$ ssh -p 49168 root@10.0.2.15
```

```
The authenticity of host '[10.0.2.15]:49169 ([10.0.2.15]:49169)' ↵  
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@1bb7a52a0c40:~#
```

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, VOLUME, WORKDIR, USER and ENV. You can see a full list of the available Dockerfile instructions [here](#).

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 like so:

Listing 1.33: Specifying a specific command to run

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

This would be articulated in the Dockerfile like so:

Listing 1.34: Using the CMD instruction

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

You can specify parameters to the command like so:

Listing 1.35: 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 you can override the `CMD` instruction using the `docker run` command. If you specify a `CMD` in your Dockerfile and one on the `docker run` command line then the command line will override the Dockerfile instruction.

NOTE It's also important to understand the interaction between the `CMD` instruction and the `ENTRYPOINT` instruction. Please 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 1.36: Overriding `CMD` instructions in the Dockerfile

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

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

Listing 1.37: Launching a container with a CMD instruction

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

Notice something different? I 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 let's see what would happen?

Listing 1.38: 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 I 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.

WORKDIR

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

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 you can override the CMD instruction on the docker run command line. Sometimes this isn't great when you want a container to behave in a certain way. The ENTRYPOINT instruction provides a command that isn't as easily overridden. Instead any arguments you 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 1.39: Specifying an ENTRYPOINT

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

Like the CMD instruction, you also specify parameters by adding to the array, for example:

Listing 1.40: Specifying an ENTRYPOINT parameter

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

NOTE You can see, like the CMD instruction above, 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 1.41: Using docker run with ENTRYPOINT

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

```
. . .  
$ sudo docker run -d jamtur01/sshd -D
```

You 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.

You can also combine `ENTRYPOINT` and `CMD` to do some neat things. For example, we might want to specify the following in our Dockerfile.

Listing 1.42: 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`.

ENV

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

Listing 1.43: 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 you can specified an environment variable prefix to a command like so:

Listing 1.44: Prefixing a RUN instruction

```
RUN gem install unicorn
```

Would be executed as:

Listing 1.45: 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.

USER

The USER instruction specifies a user that the image should be run as, for example.

Listing 1.46: Using the USER instruction

```
USER nginx
```

Will cause containers created from the image to be run by the nginx user. You can specify a username or a UID.

VOLUME

The VOLUME instruction adds volumes to any container created from the image.

Listing 1.47: Using the VOLUME instruction

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

This would attempt to mount the volume /opt/project to any container created from the image. If the volume doesn't exist container creation will fail.

ADD

The ADD instruction adds files and directories from your build environment into your image, for example installing an application. It is specified using a source and a destination like so:

Listing 1.48: 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 1.49: Specifying a directory relative to the build

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

This instruction will copy, recursively, 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 `/` then it considers the source a file.

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

Listing 1.50: 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 1.51: 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 you including any directories. New files and directories will be created with a mode of 0755 and a UID and GID of 0.

Trusted builds

In addition to being able to build and push your images from the command line the Docker Index also allows you to define automatic builds. This is done by connecting a [GitHub](#) repository containing a Dockerfile to the Docker Index. When you push to this repository an image build will be triggered and a new image created. This is also known as a trusted build.

NOTE Currently trusted builds only work on public GitHub repositories. Support for private repositories is forthcoming.

The first step in adding a trusted build to the Index is to connect your GitHub account to your Docker Index account. To do this navigate to the Docker Index, sign in and click on your profile link and then on the Trusted Build tab.

Chapter 1: Working with Docker images and repositories

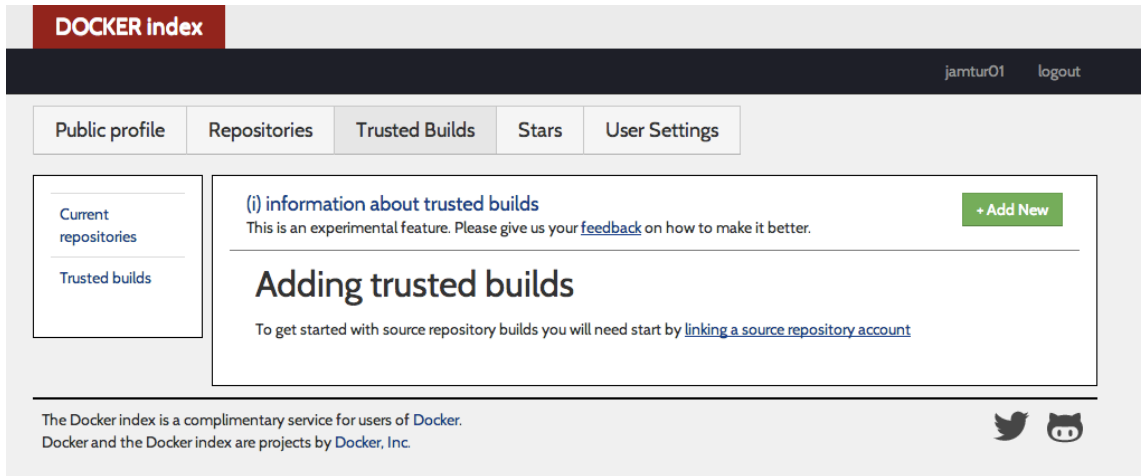


Figure 1.6: The Trusted Build tab.

Click on the +Add New button to add a new trusted build.

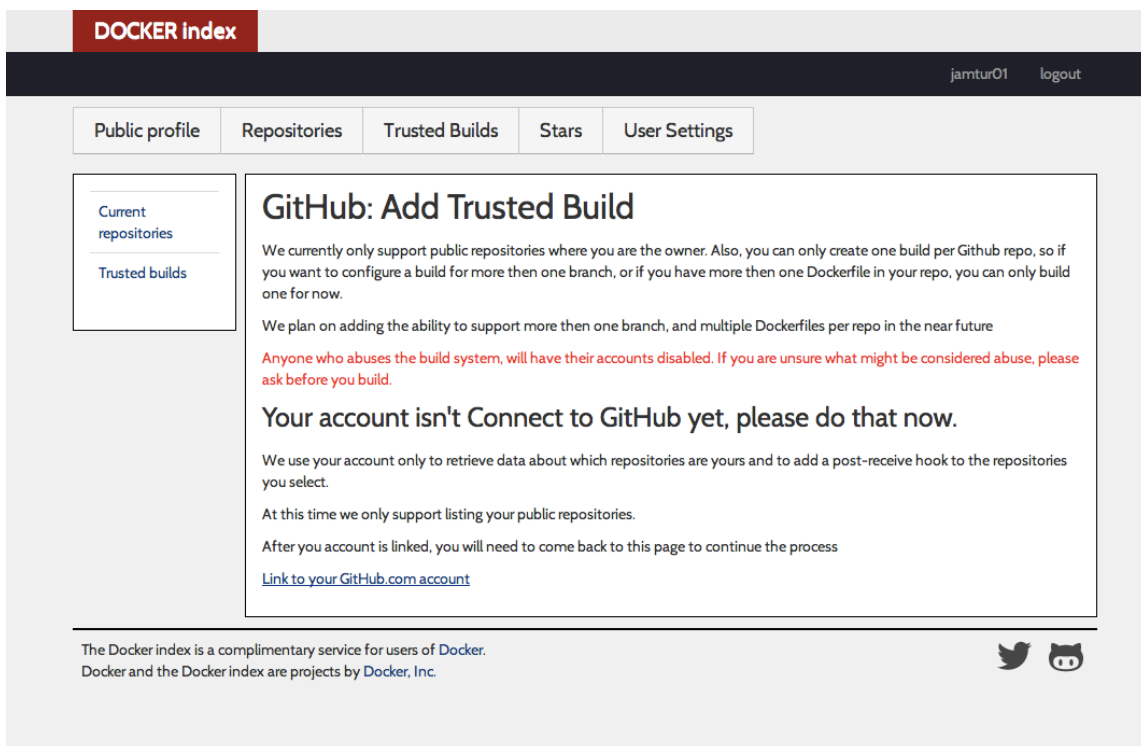


Figure 1.7: Linking to your GitHub account.

Chapter 1: Working with Docker images and repositories

Click the Link to your GitHub.com account link to initiate the account linkage. You will be taken to GitHub and asked to authorize access for the Docker Index.

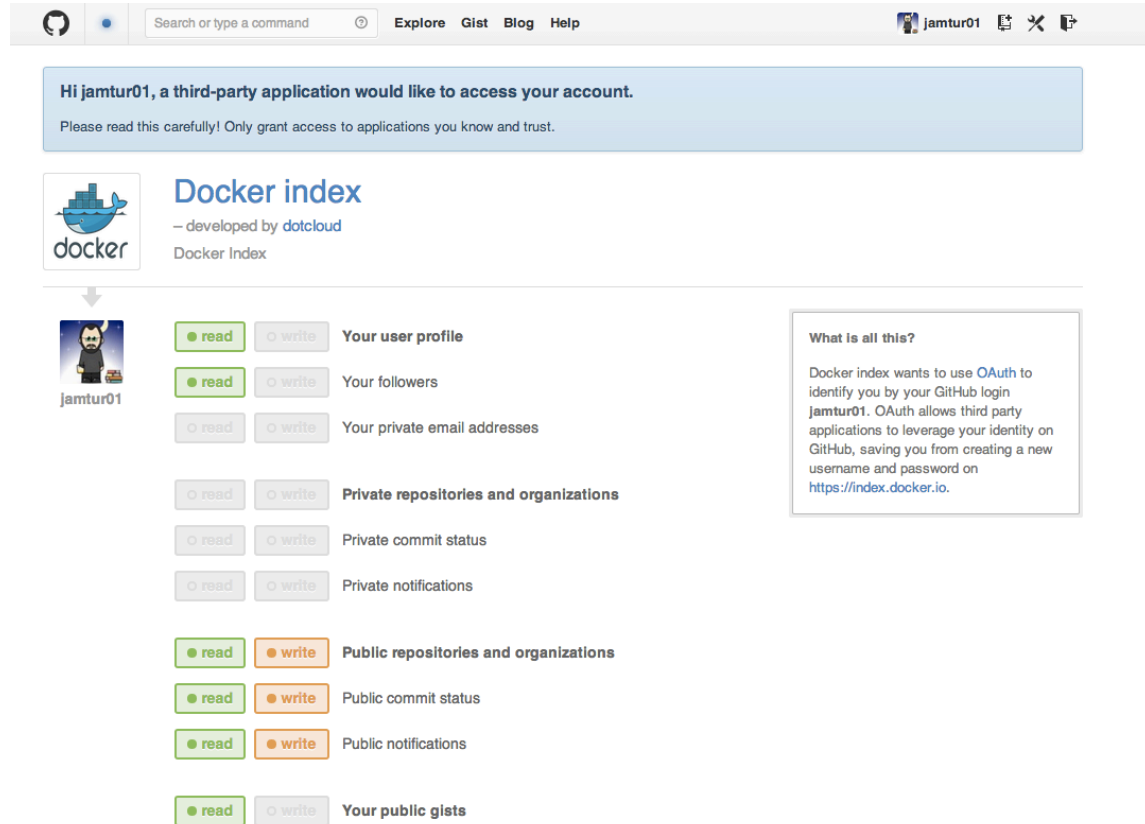


Figure 1.8: Authorizing GitHub access.

Click the 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 repository from which you want to construct a trusted build.

Chapter 1: Working with Docker images and repositories

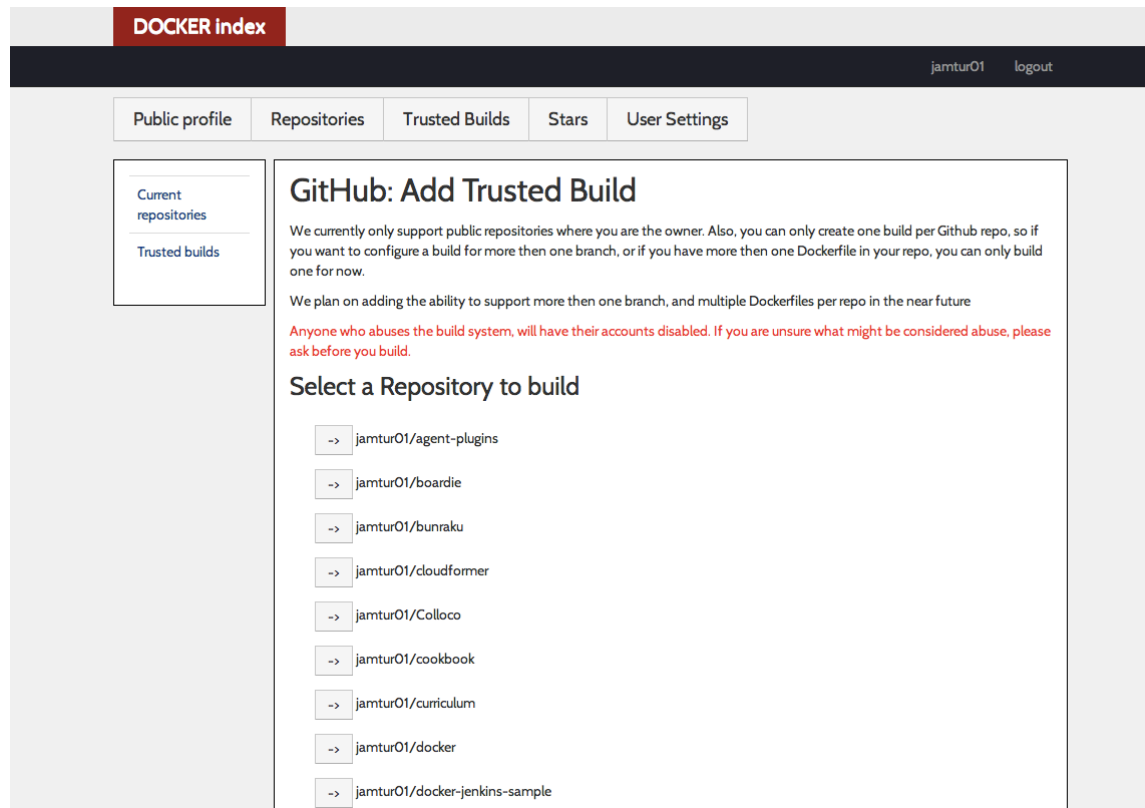


Figure 1.9: Selecting a GitHub repository.

Select the repository from which you wish to create a trusted build and then configure the build.

The screenshot shows the Docker Index interface for configuring a trusted build. At the top, there's a 'DOCKER index' header and a user profile section for 'jamtur01' with a 'logout' link. Below this is a navigation bar with tabs: 'Public profile', 'Repositories', 'Trusted Builds' (which is active), 'Stars', and 'User Settings'. On the left side of the 'Trusted Builds' page, there's a sidebar with 'Current repositories' and 'Trusted builds' (which is active). The main content area is titled 'GitHub:jamtur01/docker-jenkins-sample'. It contains several form fields: 'Default Branch' (set to 'master'), 'Repo Name' (set to 'docker-jenkins-sample'), 'Docker Tag Name' (set to 'latest'), and 'Dockerfile Location' (set to '/'). Each field has a descriptive text below it. There's also an 'Active' checkbox which is checked, with the text 'When active we will build when new pushes occur'. A 'Submit' button is at the bottom of the form. At the very bottom of the page, there's a footer with text about the Docker index being a complimentary service and links to Docker and the Docker index projects, along with Twitter and GitHub icons.

DOCKER index

jamtur01 logout

Public profile Repositories Trusted Builds Stars User Settings

Current repositories
Trusted builds

GitHub:jamtur01/docker-jenkins-sample

Default Branch
master
The code branch we are watching for changes. Leave empty to use the default value for your VCS (eg. master).

Repo Name
docker-jenkins-sample
New unique Repo name; 3 - 30 characters. Only lowercase letters, digits - and . characters are allowed

Docker Tag Name
latest
The docker tag that is applied to the build

Dockerfile Location
/
Path from project root to Dockerfile (ex. config/)

Active
☒ When active we will build when new pushes occur

Submit

The Docker index is a complimentary service for users of Docker.
Docker and the Docker index are projects by Docker, Inc.

Figure 1.10: Configuring the trusted 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 Submit button to add your trusted build to the Docker Index.

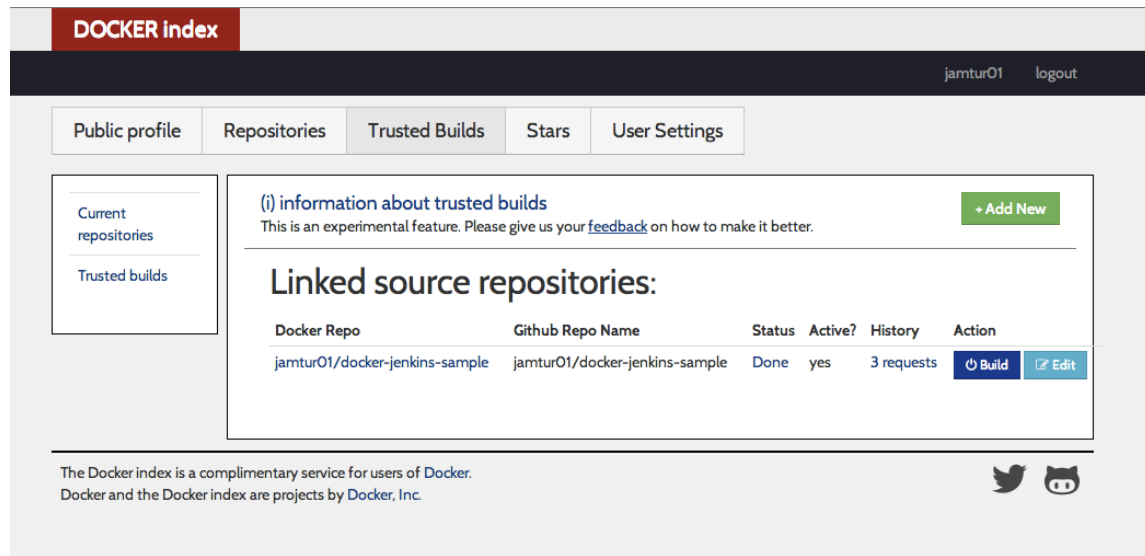


Figure 1.11: The Trusted Build status window.

You will now see your trusted build and its current status. You can also edit or delete the build from this screen. If you want to stop a trusted build from being updated then click the Edit button and un-tick the Active radio button and save your build. This will stop the build being updated when you push changes to your GitHub repository.

Click on the Status 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 trusted build is up-to-date. A status of Error indicates a problem and you can click through to see the log output.

NOTE You can't push to a trusted build using the `docker push` command. You can only update them by pushing updates to your GitHub repository.

Running your own Docker Index

Obviously having a public registry of Docker images is highly useful. Sometimes, however, you are going to want to build and store images that contain information or data that you don't want to make public. Thankfully the team at Docker Inc have [open-sourced the code](#) they use to run the Docker Index registry. This allows you to build your own internal Index.Registry!Private

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

There are two ways to run the Docker Index:

- From a Docker container.
- Via a manual installation.

Running the Index from a container

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

Listing 1.52: Running a container-based index

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

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

Installing the Docker Index manually

We can also perform a manual installation if we want a stand-alone Index running outside of Docker.

Prerequisites

The Docker Index is a Python-based application. We'll need to install some required packages first.

NOTE There's also a Go implementation of the Docker registry [here](#).

On Ubuntu we install the following packages:

Listing 1.53: Ubuntu prerequisites for the Docker Index

```
$ sudo apt-get install build-essential python-dev libevent-dev ↵  
python-pip
```

On Red Hat, CentOS and related distributions we will need the [EPEL repository](#). EPEL contains additional packages not shipped with some Red Hat-based releases.

NOTE You will not need to do this if you are installing on Fedora.

To do this we need to download and install the [EPEL repository RPM](#) like so:

Listing 1.54: Installing EPEL for RHEL/CentOS 6

```
$ rpm -Uvh http://mirror.us.leaseweb.net/epel/6/i386/epel-↵  
release-6-8.noarch.rpm
```

Then we can install the following packages:

Listing 1.55: Red Hat-based prerequisites for the Docker Index

```
$ sudo yum install python-devel libevent-devel python-pip
```

Getting the source code

Next we need to download the Docker Index registry application. We can do this by cloning the GitHub repo. You must have Git installed for this to succeed.

Listing 1.56: Downloading the Docker Index registry application

```
$ git clone https://github.com/dotcloud/docker-registry.git
```

Installing the Docker Index

From here we can change into the cloned directory and install the Docker Index.

Listing 1.57: Changing into the Docker Index directory

```
$ cd docker-registry
```

Now we can run the pip command (the command is python-pip on Red Hat-based distributions) to install the required Python libraries to run the Registry.

Listing 1.58: Using Pip to install the Docker Index

```
$ sudo pip install -r requirements.txt
```

Once this process completes you should have all the required dependencies to run the Docker Index but before we can run it we need to configure it.

TIP The Docker Index also has its own [Dockerfile](#) you can use to build a container with the application. Or you can run a pre-built container: `docker run samalba/docker-registry`.

Configuring the Docker Index

The Docker Index's configuration is controlled by the `config.yml` file. We can edit this file to configure the Docker registry. We can do this either locally in our Docker installation or via the Docker container if you've used that method to run the Index.

In a manual installation, the Docker Index application ships with a sample configuration file called `config_sample.yml`. Let's start by copying this file to be our base configuration.

Listing 1.59: Copying the Docker Index sample configuration

```
$ cp config_sample.yml config.yml
```

Let's open up the `config.yml` file and take a look at the content.

Listing 1.60: The Docker Index configuration file

```
common:
  s3_access_key: REPLACEME
  s3_secret_key: REPLACEME
  s3_bucket: REPLACEME

dev:
  storage: local
  storage_path: /tmp/registry
  loglevel: debug

prod:
  standalone: False
  storage: s3
  storage_path: /prod
  email_exceptions:
    smtp_host: REPLACEME
    smtp_login: REPLACEME
```

```
smtp_password: REPLACEME
from_addr: docker-registry@localdomain.local
to_addr: noise+dockerregistry@localdomain.local

test:
  storage: local
  storage_path: /tmp/test
```

We can see a selection of configuration headings: `common`, `dev`, `prod` and `test`. The index calls these 'flavors' and they represent different types of run environments. You can specify what flavor to run with the `SETTINGS_FLAVOR` environment variable.

Listing 1.61: Setting the Docker Index flavor

```
$ export SETTINGS_FLAVOR=prod
```

The first flavor, `common`, is a common environment that is used, overrides and is inherited by all flavors. The `dev` flavor is run by default when the application is launched without a flavor being specified.

You can see inside the `dev` flavor that we have three options: `storage`, `storage_path` and `loglevel`. The first option, `storage`, controls where Docker images are stored. The default destination is `local` using the local host's storage with a path specified via the `storage_path` option. The other alternative storage destination is [Amazon S3](#) specified by setting `storage` to `s3`. In this case you also need to set the `s3_access_key`, `s3_secret_key` and the `s3_bucket` options with Amazon Access Key and Secret Key and the name of the S3 bucket to store your images in.

The `loglevel` option controls the log level of the application. In the `dev` flavor we've specified debug output but it normally defaults to `info`.

In the `prod` flavor we can also see the `standalone` option. This option allows you to control whether to dedicate authentication and namespacing to the main [Index](#). When `standalone` is set to `False` then the local Index will use the main Index's authentication and namespacing. When it is set to `True` then the local Index will

mimic the routes and endpoints of the main Index. Enabling standalone will also disable authentication.

Lastly, we can see the email exception configuration. These options control where the Index will send error reports. You will need to specify appropriate SMTP configuration to suit your environment.

Launching the Docker Index

We can run the Docker Index in a couple of different ways. Let's start by running it locally on the command line to see it in action.

Listing 1.62: Running Docker Index locally

```
$ gunicorn --access-logfile - --debug -k gevent -b 0.0.0.0:5000 ←  
-w 1 wsgi:application
```

This will launch the Docker Index on port 5000 and bound on all interfaces. If you browse to this URL, <http://localhost:5000> you can see the server running the dev flavor:

Chapter 1: Working with Docker images and repositories

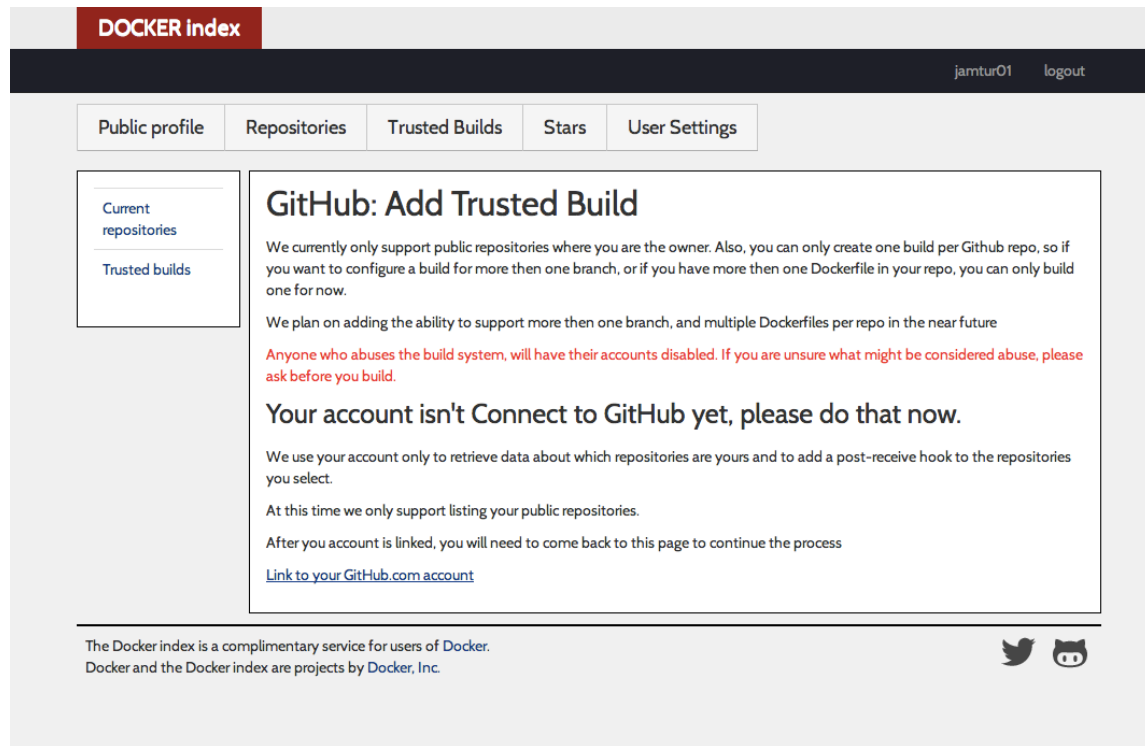


Figure 1.12: The Docker Index home page.

Testing the new Index

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

Listing 1.63: Identifying the Ubuntu image ID

```
$ sudo docker images | grep ubuntu | grep latest
ubuntu                                latest                                8↵
    dbd9e392a96                      3 months ago                      131.5 MB (virtual 131.5 ↵
    MB)
```

Chapter 1: Working with Docker images and repositories

Next we take our image ID, 8dbd9e392a96, 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 we've given our new registry a hostname of `docker.example.com`.

Listing 1.64: Tagging our image for our new registry

```
$ sudo docker tag 8dbd9e392a96 docker.example.com:5000/ubuntu
```

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

Listing 1.65: Pushing an image to our new index

```
$ sudo docker push docker.example.com:5000/ubuntu
Username (): jamtur01
Password:
Email (): james@example.com
Login Succeeded
The push refers to a repository [docker.example.com:5000/ubuntu]↵
  (len: 1)
Processing checksums
Sending image list
Pushing repository docker.example.com:5000/ubuntu (1 tags)
Pushing 8↵
    dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c
Buffering to disk 58375952/? (n/a)
Pushing 58.38 MB/58.38 MB (100%)
. . .
```

You can see that we've been prompted to input our Docker Index username and password. This is because our standalone registry is making use of the main Docker Index's authentication. This is controlled by the `standalone` option in the registry configuration.

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

Listing 1.66: Building a container from our local registry

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

Production configuration

For a more production ready configuration you can easily run a local registry from the `docker-registry` image or using `gunicorn` behind an `Nginx` front-end using `proxy_pass`.

NOTE You will need to run an `Nginx` version later than 1.3.9 as the registry requires chunked transfer-encoding.

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 is a flat per 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 Index 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 build some more complex services.

Index

- .dockercfg, 12
- Docker
 - .dockercfg, 12
 - Bind UDP ports, 18
 - busybox image, 5
 - Dockerfile
 - ADD, 29
 - CMD, 23
 - ENTRYPOINT, 26
 - ENV, 27
 - EXPOSE, 16, 18
 - FROM, 15
 - MAINTAINER, 16
 - RUN, 16
 - USER, 28
 - VOLUME, 28
 - WORKDIR, 25
 - Images, 5
 - Index, 11, 12
 - Index site, 4, 6
 - Running your own Index registry, 36
- docker
 - build, 14, 18, 19
 - commit, 8, 9
 - Env, 10
 - PortSpecs, 10
 - history, 21
 - images, 5, 21, 42
 - inspect, 17
 - login, 12
 - port, 17, 22
 - ps, 22
 - pull, 4, 7
 - push, 11, 35, 43
 - rm, 14
 - rmi, 13, 14
 - run, 1, 16, 21, 23, 24, 44
 - search, 6
 - tag, 43
- Docker Index, 12
- Dockerfile, 14, 30, 45
 - template, 20
- Env, 10
- GitHub, 30
- Index
 - Private, 36
- JSON, 12
- PortSpecs, 10
- Trusted builds, 30

Thanks! I hope you enjoyed the book.

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

