RX-M

Docker Images

Lab - Images

In this lab you will get a chance to work with and explore Docker images. Docker creates containers from images. For the Docker Engine to launch a container, the image the container is based on must be present on the Docker host. If an image isn't already present when a run command requires it, the Docker Engine will download the image automatically from a registry if possible. Registries are network based services that allow you to save and retrieve Docker images. The Docker Hub is the primary public registry.

1. Listing Images

You can list the images present on a particular Docker host using the images sub command. Try it:

```
user@ubuntu:~$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu              latest              97434d46f197        2 days ago          188 MB
ubuntu              12.04               8e3c25486445        2 days ago          138.4 MB
centos              latest              d0e7f81ca65c        2 weeks ago         196.6 MB
hello-world         latest              690ed74de00f        5 months ago        960 B
```

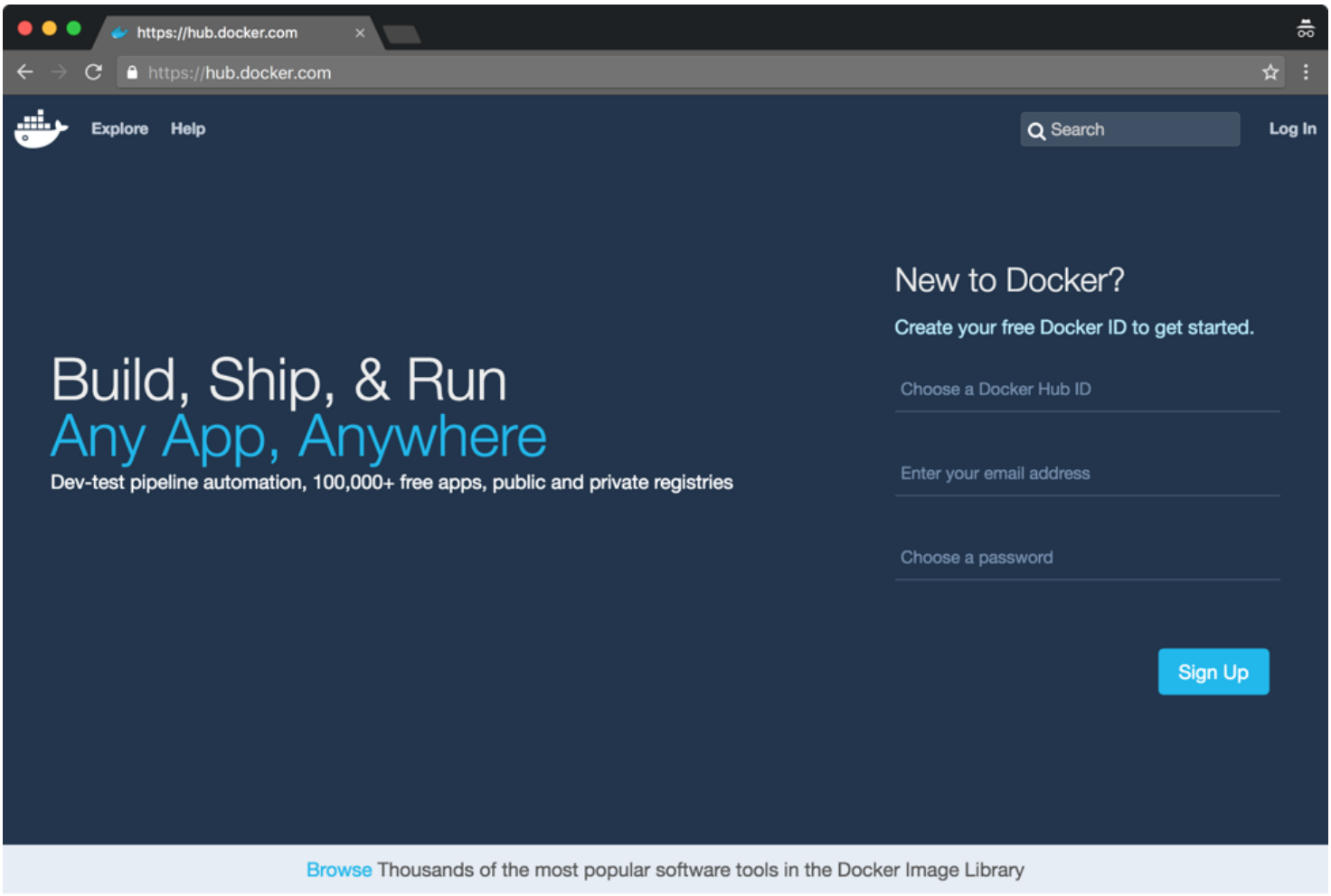
Your output may be different but as you can see the images command displays all of the tagged images on your system. Images are uniquely identified by their image ID. Prior to Docker version 1.10 image IDs generated by the Docker Engine were Universally Unique IDs (UUIDs). UUIDs are like random numbers which meant that two identical images, built on separate systems, would have different UUIDs. Docker 1.10 changed the Docker engine ID format to that of a SHA hash of the image content. This means that identical images will now have identical IDs, no matter where the image is generated.

Images are organized into repositories. A repository is like a conceptual folder. Repositories may have a single-part name or a multipart name. Two part names are of the form account/repositoryName or URL/repositoryName, three part names are of the form URL/account/repositoryName. We'll look at multi-part names more closely later.

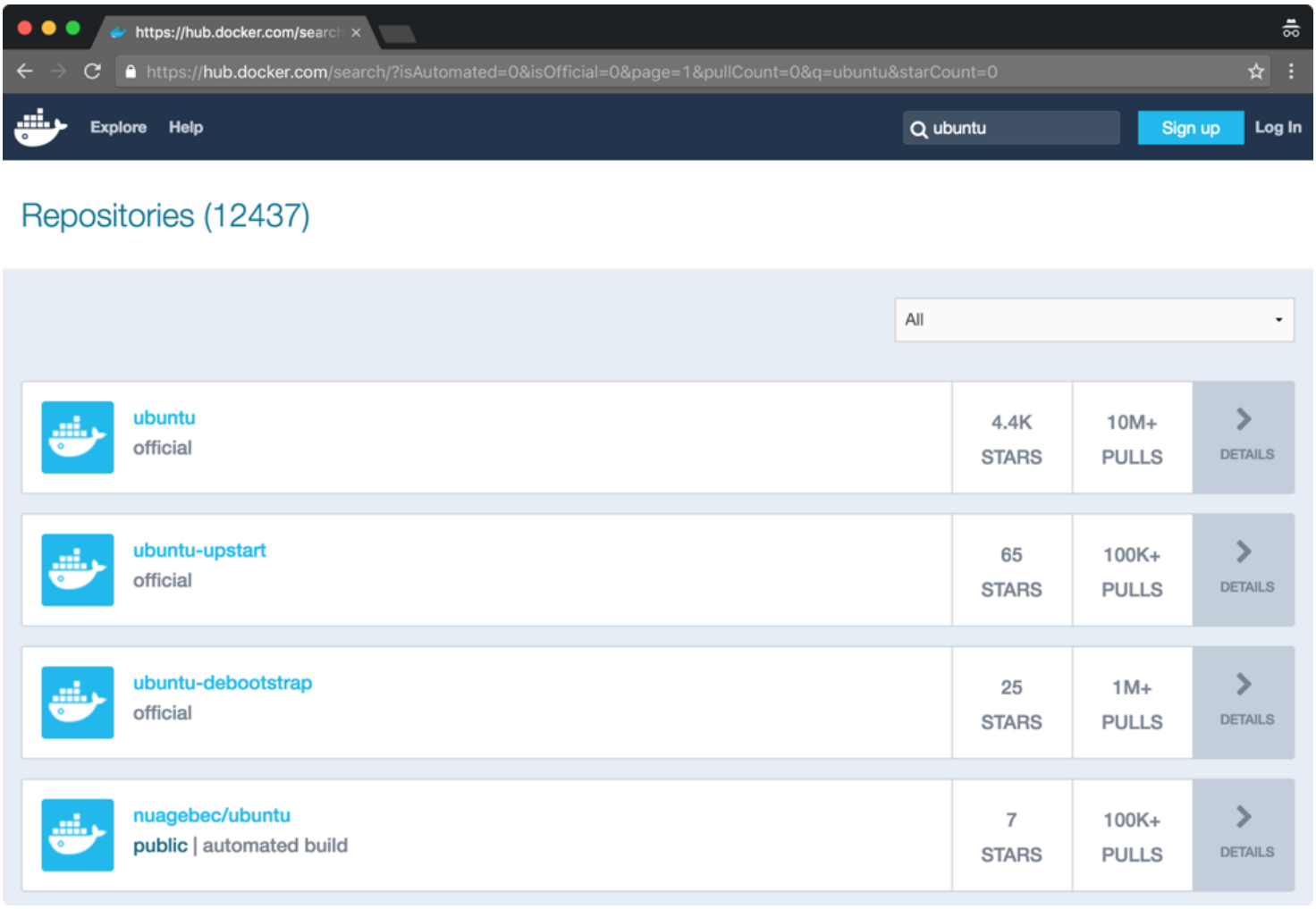
The repositories listed in the `docker images` example above have a single part name, for example "ubuntu". This indicates that the repository is an "official" repository. Such repositories are sourced from Docker Hub and curated directly or indirectly by Docker Inc.

You can always check Docker Hub if you would like to know who maintains an image or repository (which is not a bad idea from a security perspective). To identify the creator of the Ubuntu image go to the Docker Hub Registry and search for ubuntu:

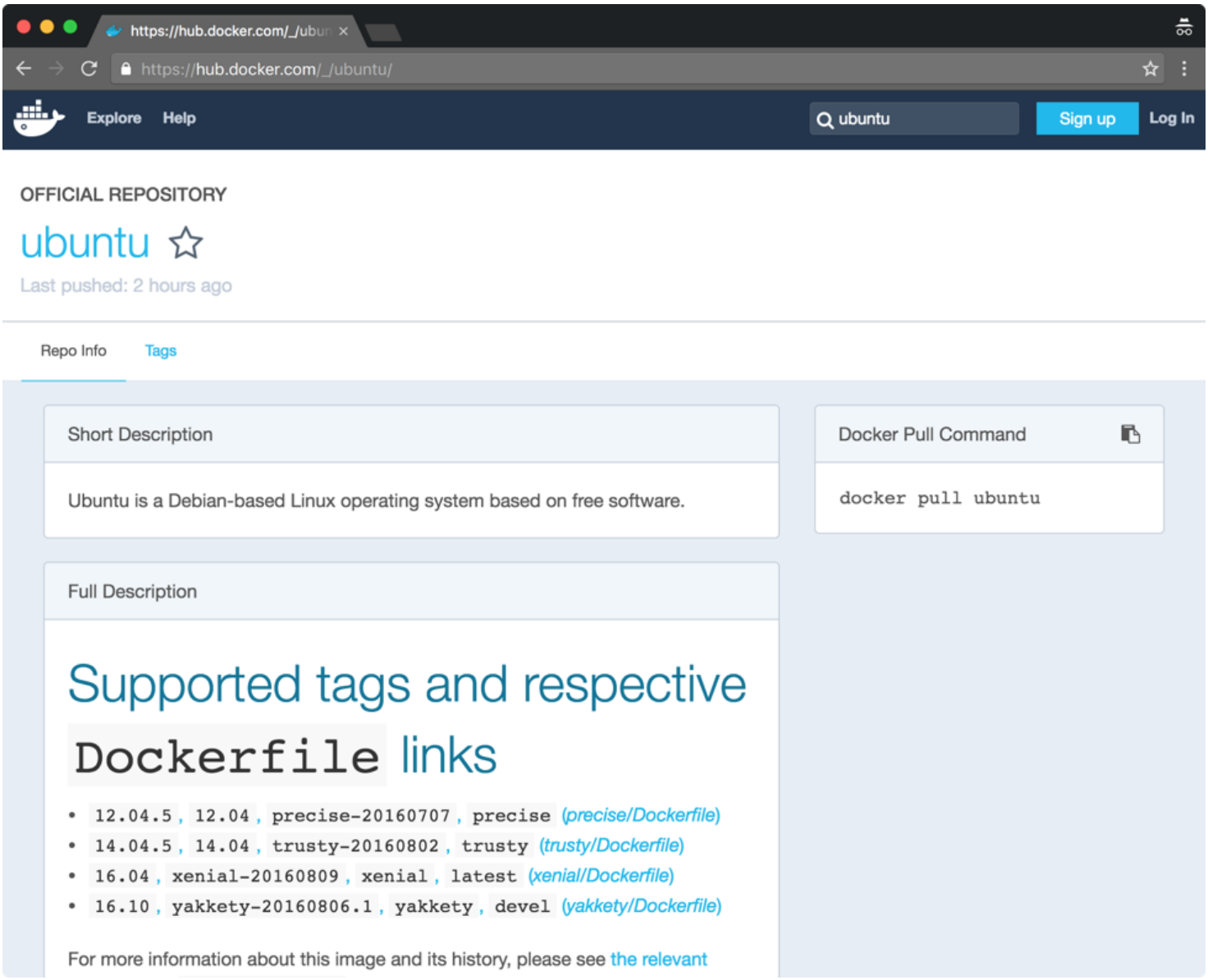
`https://hub.docker.com`



Searching for “ubuntu” should give you a list of repositories with “ubuntu” in the name. Click the official “ubuntu” exact match (it should be listed first).



The Ubuntu repository details page provides useful information about the Ubuntu repository.



Many images can share the same repository name. Individual images with the same repository name are identified by tag names. The Ubuntu repository contains four images, each with several tags. For example the tag “12.04.5” and the tag “12.04” reference the same image which was generated with the “precise/Dockerfile”. We will look at Dockerfiles in a later lab, Dockerfiles are the main way Docker images are constructed in practice. You can click the “precise/Dockerfile” link to see the source and identify the author if you are curious.

2. Docker pull

Imagine a scenario where you are configuring a production machine (perhaps with Puppet, Ansible, or Chef.) Assume this machine will run Docker and several containers. If you know in advance the images you require, you can download them to the host during configuration, avoiding any download delays during operation.

For example, assume we need the Fedora Rawhide image on a system to run some Fedora based containers. You can use the following command to pull the image from Docker Hub:

```
user@ubuntu:~$ docker pull fedora:rawhide
rawhide: Pulling from library/fedora

09e95d7fbe77: Pull complete
Digest: sha256:e72d5c6e74808353dc5849bac9c38ededfe4e197d1ba5ba213de9ff08af063f7
Status: Downloaded newer image for fedora:rawhide
```

Images are always referred to by name when pushing them to, or pulling them from, a registry. If you do not specify a tag name, Docker will assume the tag “latest”. If you need to reference an image other than the one tagged “latest”, you can supply the tag following the repository name and a colon.

Images can be layered. Each layer adds files or overlays files from the layers below. This allows images to be highly reusable. Each image, other than the base image, has precisely one parent image. Because each image is assigned an ID, Docker knows when it already has one or more of the images required by a descendant. In the example above the system downloaded the fedora:rawhide image (e72d5c6e7480). Because images and their filesystem layers are static and can never be changed, Docker can skip downloading images or filesystem layers associated with IDs it already has. Related images can share layers, which can save transfer bandwidth, reduce memory foot prints and simplify builds among other things.

3. Creating Images

Images can be created in several ways. The easiest way to create an image is to run a container interactively and then install the files and configuration you require. When complete you can create an image from the container using the `docker commit` subcommand. If you have ever taken a snapshot of a virtual machine, the process is similar.

Imagine we need to run a web server in production and our production infrastructure runs SUSE and Ubuntu Linux, yet our development team builds and tests web servers on CentOS 6. We can use a container in this situation. A CentOS based web server container can run on both SUSE and Ubuntu. To illustrate we'll build a web server image based on CentOS 6.

Run a container from the centos:6 image, name it "websvr" and attach to a bash shell within the container:

```
user@ubuntu:~$ docker run -it --name "websvr" centos:6 /bin/bash
Unable to find image 'centos:6' locally
6: Pulling from library/centos

08a7a0bb6122: Pull complete
Digest: sha256:cd6d68000b47a91e7c94b558d7e3e653c3f0eac1a77842d97b0b7ad955cad608
Status: Downloaded newer image for centos:6
[root@a11d80ac44ca /]#
```

Now install the apache web server:

```
[root@a11d80ac44ca /]# yum install -y httpd
...
Complete!
[root@a11d80ac44ca /]#
```

In the real world we would probably do more configuring but for our purposes, we will call our web server complete.

In Docker, a container is an instance of an image running (or stopped). You can start and stop containers but you cannot run a container based on another container, containers must be based on an image. Also, Docker can push and pull images to network based registries but not containers. You can however create an image from a given container and then run/push/pull the image.

The `docker commit` subcommand creates an image from a container. Exit the CentOS container you have configured and create an image from it using the `docker commit` subcommand:

```
[root@a11d80ac44ca /]# exit
exit
```

```
user@ubuntu:~$ docker commit -m "C6 web svr" -a "dockerlab" a11d lab/websvr:v0.1
sha256:5d92ac8a103e0ddab17fd83d0ca4e6e64481fb19361d6c7ff53fbe6e8b7ba42
```

We used the following arguments to the `docker commit` subcommand:

- **-m** This switch adds a Message to your image (a commit message, like git commit -m)
- **-a** This switch sets the Author of the image
- **a11d** This is the ID prefix of the container to commit (from the container prompt)
- **lab** This is the account part of the repository string
- **websvr** This is the repository name part of the repository string
- **v0.1** This is the tag assigned to the image

Use the `docker images` command to display your new image:

```
user@ubuntu:~$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
lab/websvr          v0.1                5d92ac8a103e        39 seconds ago      335.8 MB
...
```

To display all of the images your new image is based on, you can use the `docker history` subcommand:

```
user@ubuntu:~$ docker history lab/websvr:v0.1
IMAGE               CREATED             CREATED BY                                      SIZE                COMMENT
5d92ac8a103e        2 minutes ago       /bin/bash                                       106.9 MB            C6 web svr
d0a31e3494fe        2 weeks ago         /bin/sh -c #(nop) CMD ["/bin/bash"]             0 B
<missing>           2 weeks ago         /bin/sh -c #(nop) LABEL name=CentOS Base Imag   0 B
<missing>           2 weeks ago         /bin/sh -c #(nop) ADD file:de20fbb4fe344f0ff5   228.9 MB
<missing>           6 months ago        /bin/sh -c #(nop) MAINTAINER The CentOS Proje   0 B
```

In the example listing, two images have IDs and three do not. The image ID 5d92… is our new websvr image. The image ID d0a3… is the centos:6 image we based our container on. The remaining images are layers of instructions used to build the centos:6 image. Only one of these images has a filesystem layer, and therefore shows a non-zero size. Because these ancestor images were downloaded as part of the centos:6 image they do not have independent IDs locally and cannot be directly executed.

4. Running an image

Run your new image with a bash shell and explore the container created:

```
user@ubuntu:~$ docker run -it lab/websvr:v0.1

[root@97651fda98b0 /]# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 19:38 ?        00:00:00 /bin/bash
root        15     1  0 19:38 ?        00:00:00 ps -ef

[root@97651fda98b0 /]# yum list installed | grep httpd
httpd.x86_64                    2.2.15-54.el6.centos              @updates
httpd-tools.x86_64              2.2.15-54.el6.centos              @updates

[root@97651fda98b0 /]# ls /usr/sbin/httpd
/usr/sbin/httpd

[root@97651fda98b0 /]# exit
exit

user@ubuntu:~$
```

After exploring the container generated from the image exit back to the host. Try running a container from your image as a daemon with the -d switch:

```
user@ubuntu:~$ docker run -d lab/websvr:v0.1
e127eaf997fea55bf5dadb6d4e990b4da7a09fc339cc4a5596c583e575c46961

user@ubuntu:~$ docker ps
CONTAINER ID        IMAGE      COMMAND             CREATED             STATUS              PORTS         NAMES

user@ubuntu:~$ docker ps -a
CONTAINER ID  IMAGE             COMMAND       CREATED              STATUS                          PORTS   NAMES
e127eaf997fe  lab/websvr:v0.1   "/bin/bash"   11 seconds ago       Exited (0) 10 seconds ago               cocky_brahmagupta
97651fda98b0  lab/websvr:v0.1   "/bin/bash"   About a minute ago   Exited (0) About a minute ago           serene_goldstine
26cbb1af8680  centos:6          "/bin/bash"   34 minutes ago       Exited (0) 8 minutes ago                websvr
```

The container launched with the –d switch exited immediately after we launched it. This is because the command associated with the image is “/bin/bash” and bash shells exit immediately if they are not connected to an input stream. What we really want is for the container to run the web server, `/usr/sbin/httpd`.

This image needs more work before it is ready to use.

5. Committing new metadata

Using commit as we did above creates an image with the same basic features (metadata) as the container we used to create the image. The commit “--change” option allows you to change the metadata of the new committed image. You can use any of the following Dockerfile commands with the change switch:

- CMD – sets an overridable set of arguments for the command line
- ENTRYPOINT – sets base command line arguments that are not overridden by default
- ENV – sets an environment variable
- EXPOSE – defines network service ports
- LABEL – creates arbitrary key/value pairs
- ONBUILD – supports image templating
- USER – configures the default container user
- VOLUME – mounts an external volume
- WORKDIR – sets the working directory

We will cover Dockerfiles in detail later but for now you can think of a Dockerfile as a way to script images.

Try running the image you just created interactively (-it) without a command argument, then use ps to see which program the container runs by default:

```
user@ubuntu:~$ docker run -it lab/websvr:v0.1

[root@a19dd81b8bb0 /]# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 17:16 ?        00:00:00 /bin/bash
root        14     1  0 17:16 ?        00:00:00 ps -ef

[root@a19dd81b8bb0 /]# exit
exit
```

```
user@ubuntu:~$
```

In the example above you can see that the image launches a bash shell when no other argument is given. Examine the history of the image:

```
user@ubuntu:~$ docker history lab/websvr:v0.1
IMAGE               CREATED             CREATED BY                                      SIZE                COMMENT
5d92ac8a103e        17 minutes ago      /bin/bash                                       106.9 MB            C6 web svr
d0a31e3494fe        2 weeks ago         /bin/sh -c #(nop) CMD ["/bin/bash"]             0 B
<missing>           2 weeks ago         /bin/sh -c #(nop) LABEL name=CentOS Base Imag   0 B
<missing>           2 weeks ago         /bin/sh -c #(nop) ADD file:de20fbb4fe344f0ff5   228.9 MB
<missing>           6 months ago        /bin/sh -c #(nop) MAINTAINER The CentOS Proje   0 B
```

Note that the image we created by committing the container inherited the `/bin/bash` CMD. Using commit with the --change switch we can override this.

Imagine we want to commit the container as before but we want to run `/usr/bin/httpd` or `/bin/sh` or some other program when the image is executed. Use the command below to recommit the container but this time with `/bin/sh` as the default command:

```
user@ubuntu:~$ docker commit --change 'CMD ["/bin/sh"]' a11d lab/websvr:latest
sha256:40104b973f61cc43e2306157ba309851aebdd900cacdde88fa24db3a00bd3480
```

Now run the new image and see which program runs by default:

```
user@ubuntu:~$ docker run -it lab/websvr:latest

sh-4.1# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 17:20 ?        00:00:00 /bin/sh
root         6     1  0 17:20 ?        00:00:00 ps -ef

sh-4.1# exit
exit
```

If you run `docker history` on the new lab/websvr:latest image you will see that it comes from the same `/bin/bash` based container. However, if you look at the metadata, you will see that the image in fact uses the configured command “/bin/sh”.

```
user@ubuntu:~$ docker inspect -f '{{.Config.Cmd}}' lab/websvr:latest
{[/bin/sh]}
```

Using `docker commit` and multiple --change switches we can configure an image that has the correct environment variables and configuration to run a web server. Commit a new image from your container with the necessary metadata to run our webserver:

```
user@ubuntu:~$ docker commit --change 'CMD ["/usr/sbin/httpd","-D","FOREGROUND"]' \
>                            --change 'ENV APACHE_RUN_USER www-data' \
>                            --change 'ENV APACHE_RUN_GROUP www-data' \
>                            a11d lab/websvr:v0.2
sha256:169c8d107c9468227b614b0184a04602a5240e90b4afacbba9531326cfda7c5f
```

The `docker commit` subcommand above is listed on multiple lines for readability using the backslash `\` character to escape the return key. You can type it all on one line if you like. When the commit completes the resulting web server image can be run in the background as a daemon:

```
user@ubuntu:~$ docker run -d lab/websvr:v0.2
f514a19b6fc2931954a325926d2bdf56206770c4280f7717ad27174f8e350a13
```

Now list the running containers:

```
user@ubuntu:~$ docker ps
CONTAINER ID  IMAGE            COMMAND                CREATED     STATUS     PORTS  NAMES
f514a19b6fc2  lab/websvr:v0.2  "/usr/sbin/httpd -D F"  9 secs ago  Up 9 secs         serene_lovelace
```

Now we can try contacting the web server to see if it is actually running. First we need to discover the new container’s IP address on the Docker network. Fortunately containers have metadata just like images. When Docker launches a new container it assigns the container an IP address on the Docker host’s private Docker network and then saves the IP address in the container’s metadata. The IPAddress key is stored under the NetworkSettings key in the container meta data. Use the following command to lookup the container’s IP address:

```
user@ubuntu:~$ docker inspect -f '{{.NetworkSettings.IPAddress}}' f514
172.17.0.2
```

Now we can try to hit the web server with telnet. Telnet to the webserver's IP using the standard WWW port 80 (make sure to use the IP address from your system). When telnet connects send the command "GET /" to get the root page on the webserver:

```
user@ubuntu:~$ telnet 172.17.0.2 80
Trying 172.17.0.2...
Connected to 172.17.0.2.
Escape character is '^]'.
GET /
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
    <head>
        <title>Apache HTTP Server Test Page powered by CentOS</title>
...
</body>
</html>
Connection closed by foreign host.
user@ubuntu:~$
```

It works! Notice that we are now running a CentOS based Apache web server on an Ubuntu VM! You have just created your first useful container image.

6. Deleting an image

The `docker rmi` command deletes images from the local Docker system. Images stored in a registry can be downloaded again if needed. Delete one of the websvr images with the following command:

```
user@ubuntu:~$ docker rmi lab/websvr:v0.1
Failed to remove image (lab/websvr:v0.1): Error response from daemon: conflict: unable to remove repository reference "lab/websvr:v0.1" (must
force) - container 85020cc9f3bc is using its referenced image 5d92ac8a103e
```

What happened?

Now remove all containers (stop them first if they are still running) that depend on the image and retry the image delete (make sure you use the image and container ids from your system):

```
user@ubuntu:~$ docker rmi lab/websvr:v0.1
Failed to remove image (lab/websvr:v0.1): Error response from daemon: conflict: unable to remove repository reference "lab/websvr:v0.1" (must
force) - container 85020cc9f3bc is using its referenced image 5d92ac8a103e

user@ubuntu:~$ docker stop 85020cc9f3bc
85020cc9f3bc

user@ubuntu:~$ docker rm 85020cc9f3bc
85020cc9f3bc

user@ubuntu:~$ docker rmi lab/websvr:v0.1
Failed to remove image (lab/websvr:v0.1): Error response from daemon: conflict: unable to remove repository reference "lab/websvr:v0.1" (must
force) - container 1d618652d08f is using its referenced image 5d92ac8a103e

user@ubuntu:~$ docker stop 1d618652d08f
1d618652d08f

user@ubuntu:~$ docker rm 1d618652d08f
1d618652d08f

user@ubuntu:~$ docker rmi lab/websvr:v0.1
Failed to remove image (lab/websvr:v0.1): Error response from daemon: conflict: unable to remove repository reference "lab/websvr:v0.1" (must
force) - container b2dce90a56ae is using its referenced image 5d92ac8a103e

user@ubuntu:~$ docker stop b2dce90a56ae
b2dce90a56ae

user@ubuntu:~$ docker rm b2dce90a56ae
b2dce90a56ae

user@ubuntu:~$ docker rmi lab/websvr:v0.1
Untagged: lab/websvr:v0.1
Deleted: sha256:5d92ac8a103e0ddab17fd83d0ca4e6e64481fb19361d6c7ff53fbe6e8b7ba425
Deleted: sha256:19b331ef44e3e4f96b84b544329b32faa45637b600b6bb1742029f0ed3006e48

user@ubuntu:~$
```

7. Cleanup

To cleanup we need to stop any running containers and remove all of the containers on the lab VM. There is an easy way to perform both tasks. The `docker ps` subcommand displays containers and offers the following switches:

- **--no-trunc** to display the full id
- **-a** to display all (running and stopped) containers
- **-q** to display only the container ids

Run the following command:

```
user@ubuntu:~$ docker ps --no-trunc -q
f514a19b6fc2931954a325926d2bdf56206770c4280f7717ad27174f8e350a13
```

Your output will be different from the example. This command displays just the IDs of the running Docker containers. You can try the command without the –q to see the full details.

We can feed the ids generated by the above command to the `docker stop` subcommand to stop all running containers. Try the following command:

```
user@ubuntu:~$ docker stop `docker ps --no-trunc -q`
f514a19b6fc2931954a325926d2bdf56206770c4280f7717ad27174f8e350a13
```

Unix shells run strings enclosed in back ticks (`) in a subshell. The above command thus runs the `docker ps` subcommand to generate the ids of the running containers and then feeds this to the `docker stop` subcommand as its argument list. This is an easy way to stop all running containers on a system. The stop operation may take a moment with each container.

We can use the same technique to remove all containers from a system. Try the following command:

```
user@ubuntu:~$ docker rm `docker ps --no-trunc -aq`
f514a19b6fc2931954a325926d2bdf56206770c4280f7717ad27174f8e350a13
6df1ebdc496d7e51ff7a61a71321977c60935aeb8baf4853f0491a7b7a0c1317
26cbb1af8680cfdd928bcb8807cb52dc57fd3e61099528a7e450cec3488a1281
```

This time we add the –a switch to the sub shell command to generate the ids of all of the containers (running and stopped) and pass that to the rm command. This removes all of the containers from our system.

Run a `docker ps -a` subcommand to ensure everything is deleted.

Congratulations, you have completed the Docker Images lab!