Docker Images

What is a Docker image?

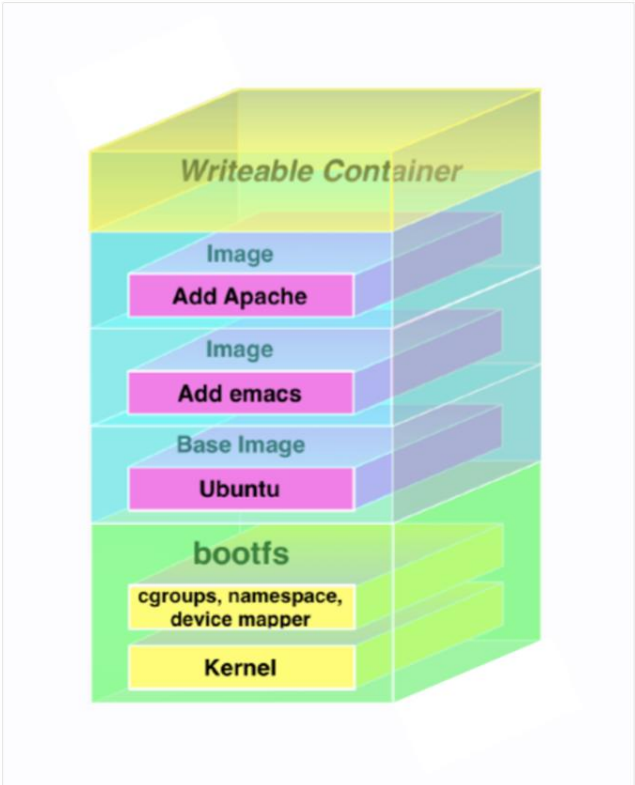
A Docker image is made up of filesystems layered over each other. At the base is a boot filesystem, bootfs, which resembles the typical Linux/Unix boot filesystem. A Docker user will probably never interact with the boot filesystem. Indeed, when a container has booted, it is moved into memory, and the 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 appears 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, let's represent it by a diagram.



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 readwrite 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 learned, 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

We can list the images using the `docker images` command.

```
ubuntu@ip-172-31-34-103:~$ sudo docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
ubuntu              latest              4e5021d210f6        2 weeks ago
hello-world         latest              fce289e99eb9        15 months ago
ubuntu@ip-172-31-34-103:~$
```

We see that we've got images, from a repository called ubuntu and hello-world. When we ran the docker run command, that part of the process was downloading these images.

Tips:

- 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.

Images were 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.

Tips:

- The Docker registry code is open source. You can also run your own registry, as we'll see later in this chapter. The Docker Hub product is also available as a commercial "behind the firewall" product called Docker Trusted Registry, formerly Docker Enterprise Hub.

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 18.04, 19.10, 20.04, 14.04, and 16.04). Let's get another image from the ubuntu repository now.

```
ubuntu@ip-172-31-34-103:~$ sudo docker pull ubuntu:14.04
14.04: Pulling from library/ubuntu
2e6e20c8e2e6: Pull complete
30bb187ac3fc: Pull complete
b7a5bcc4a58a: Pull complete
Digest: sha256:ffc76f71dd8be8c9e222d420dc96901a07b61616689a44c7b3ef6a10b7213de4
Status: Downloaded newer image for ubuntu:14.04
docker.io/library/ubuntu:14.04
ubuntu@ip-172-31-34-103:~$
```

Here we've used the docker pull command to pull down the Ubuntu 14.04 image from the ubuntu repository. Let's see what our docker images command reveals now.

```
ubuntu@ip-172-31-34-103:~$ sudo docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
ubuntu              latest              4e5021d210f6        2 weeks ago
ubuntu              14.04               6e4f1fe62ff1        3 months ago
hello-world         latest              fce289e99eb9        15 months ago
ubuntu@ip-172-31-34-103:~$
```

We can see we've now got the latest Ubuntu image and the 14.04 image. This shows us that the ubuntu image is actually a series of images collected under a single repository.

Tips:

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

We identify each image inside that repository by what Docker calls tags. Each image is being listed by the tags applied to it, so, for example, 18.04, 19.10, 14.04, or xenial and so on. Each tag marks together with a series of image layers that represent a specific image (e.g., the 14.04 tag collects together all the layers of the Ubuntu 14.04 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:

```
ubuntu@ip-172-31-34-103:~$ sudo docker run -t -i --name clarusway_container ubur
root@d5f0f7c29aed:/#
```

This launches a container from the ubuntu:14.04 image, which is an Ubuntu 14.04 operating system. It's always a good idea to build a container from specific tags. That way we'll know exactly what the source of our container is. There are differences, for example, between Ubuntu 14.04 and 18.04, so it would be useful to specifically state that we're using ubuntu:14.04 so we know exactly what we're getting.

Pulling images

When we run a container from images with the docker run command if the image isn't present locally already then Docker will download it from the Docker Hub. By default, if you don't specify a specific tag, Docker will download the latest tag, for example: this will download the ubuntu:latest image if it isn't already present on the host.

```
ubuntu@ip-172-31-34-103:~$ sudo docker run -t -i --name next_container ubuntu /b
root@6851cefbe11c:/#
```

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

```
ubuntu@ip-172-31-34-103:~$ sudo docker pull fedora
Using default tag: latest
latest: Pulling from library/fedora
5c1b9e8d7bf7: Pull complete
Digest: sha256:c97879f8bebe49744307ea5c77ffc76c7cc97f3ddec72fb9a394bd4e4519b388
Status: Downloaded newer image for fedora:latest
docker.io/library/fedora:latest
ubuntu@ip-172-31-34-103:~$
```

Searching for images

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

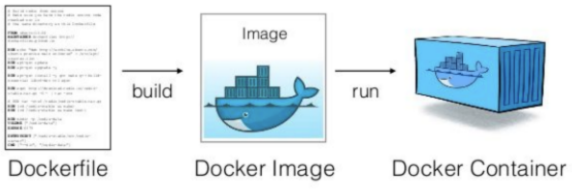
```
ubuntu@ip-172-31-34-103:~$ sudo docker search python
NAME                              DESCRIPTION
python                            Python is an interpreted, interactive, objec…
. . .
```

Here, we've searched the Docker Hub for the term python. It'll search for 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

Dockerfile

A Dockerfile› is a text document that contains all the commands a user could call on the command line to assemble an image. Using docker build, users can create an automated build that executes several command-line instructions in succession. Docker can build images automatically by reading the instructions from a ***Dockerfile****.*



Docker File

Some instructions in the Dockerfile described below:

- **FROM**: The FROM instruction initializes a new build stage and sets the Base Image for subsequent instructions. As such, a valid Dockerfile must start with a *FROM* instruction.
- **RUN**: The RUN instruction will execute any commands in a new layer on top of the current image and commit the results. The resulting committed image will be used for the next step in the Dockerfile.
- **CMD**: The main purpose of a CMD is to provide defaults for an executing container.
- **ADD**: The ADD instruction copies new files, directories or remote file URLs from `<src>` and adds them to the filesystem of the image at the path `<dest>`.
- **COPY**: The COPY instruction copies new files or directories from `<src>` and adds them to the filesystem of the container at the path `<dest>`.
- **WORKDIR**: The WORKDIR instruction sets the working directory for any `RUN`, `CMD`, `COPY` and `ADD` instructions that follow it in the Dockerfile. If the WORKDIR doesn't exist, it will be created even if it's not used in any subsequent Dockerfile instruction.

> **Note:**
> When ADD'ing files Docker uses the ending character of the destination to determine what the source is. If the destination ends in a /, then it considers the source a directory. If it doesn't end in a /, it considers the source a file.
>
> COPY only supports the basic copying of local files into the container, while ADD has some features (like local-only tar extraction and remote URL support) that are not immediately obvious. Consequently, the best use for ADD is local tar file auto-extraction into the image, as in ADD rootfs.tar.xz /.

Our first Dockerfile

Let's create a directory and an initial Dockerfile. We're going to build a Docker image that contains a simple web server.

```
ubuntu@ip-172-31-34-103:~$ mkdir static_web
ubuntu@ip-172-31-34-103:~$ cd static_web
ubuntu@ip-172-31-34-103:~/static_web$ touch Dockerfile
ubuntu@ip-172-31-34-103:~/static_web$
```

We've created a directory called static_web to hold our Dockerfile. This directory is our build environment, which is what Docker calls a context or build context. Docker will upload the build context, as well as any files and directories contained in it, to our Docker daemon when the build is run. This provides the Docker daemon with direct access to any code, files or other data you might want to include in the image.

We've also created an empty Dockerfile file to get started. Now let's look at an example of a Dockerfile to create a Docker image that will act as a Web server.

```
FROM ubuntu:18.04
MAINTAINER Clarusway "clarusway@example.com"
RUN apt-get update && apt-get install -y nginx
RUN echo 'Hi, I am in your container' \
>/usr/share/nginx/html/index.html
EXPOSE 80
```

The Dockerfile contains a series of instructions paired with arguments. Each instruction, for example FROM, should be in upper-case and be followed by an argument: FROM ubuntu:18.04. Instructions in the Dockerfile are processed from the top down, so you should order them accordingly.

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

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

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

Dockerfile Format

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

In our sample Dockerfile we've specified the ubuntu:18.04 image as our base image. This specification will build an image on top of an Ubuntu 18.04 base operating system. As with running a container, you should always be specific about exactly from which base image you are building.

Next, we've specified the 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 two RUN instructions. The RUN instruction executes commands on the current image. The commands in our example: updating the installed APT repositories and installing the nginx package and then creating the /usr/share/nginx/html/index.html file containing some example text. As we've discovered, each of these instructions will create a new layer and, if successful, will commit that layer and then execute the next instruction.

By default, the RUN instruction executes inside a shell using the command wrapper /bin/sh -c. If you are running the instruction on a platform without a shell or you wish to execute without a shell (for example, to avoid shell string munging), you can specify the instruction in exec format:

```
RUN [ "apt-get", " install", "-y", "nginx" ]

```

We use this format to specify an array containing the command to be executed and then each parameter to pass to the command.

Next, we've specified the EXPOSE instruction, which tells Docker that the application in this container will use this specific port on the container. That doesn't mean you can automatically access whatever service is running on that port (here, port 80) on the container. For security reasons, Docker doesn't open the port automatically but waits for you to do it when you run the container using the docker run command.

> Q: Explain Dockerfile
> A: Dockerfile contain a set of instructions that specify what environment to use and which commands to run. It enable you to create your own images. A Dockerfile describes the software that makes up an image.
>
> — - Interview Q&A

Building our own images

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:

```
ubuntu@ip-172-31-34-103:~$ cd clarusway_web/
ubuntu@ip-172-31-34-103:~/clarusway_web$ sudo docker build -t="clarusway_web" .
Sending build context to Docker daemon  2.048kB
Step 1/5 : FROM ubuntu:18.04
18.04: Pulling from library/ubuntu
Digest: sha256:bec5a2727be7fff3d308193cfde3491f8fba1a2ba392b7546b43a051853a341d
Status: Downloaded newer image for ubuntu:18.04
 ---> 4e5021d210f6
Step 2/5 : MAINTAINER Clarusway "clarusway@example.com"
 ---> Running in 9e95fbd3f1c0
Removing intermediate container 9e95fbd3f1c0
 ---> 6f3695c27424
Step 3/5 : RUN apt-get update && apt-get install -y nginx
 ---> Running in 455186c90238
 ---> e004ac10cab7
Step 4/5 : RUN echo 'Hi, I am in your container' >/usr/share/nginx/html/index.html
 ---> Running in 87d2cf2b74f1
Removing intermediate container 87d2cf2b74f1
 ---> 1d867b5db951
Step 5/5 : EXPOSE 80
 ---> Running in 82fc3f677dad
Removing intermediate container 82fc3f677dad
 ---> 5ea70fe27e8c
Successfully built 5ea70fe27e8c
Successfully tagged clarusway_web:latest

```

We've used the `docker build` command to build our new image.

Launching a container from our new image

Let's launch a new container using our new image.

```
ubuntu@ip-172-31-34-103:~$ sudo docker run -d -p 80 --name clarusway clarusway_web
    nginx -g "daemon off;"
4202131d6dfac5b7288bcc7469d67f00ca3d82bf3b3b85229edd04b30255d3f4
ubuntu@ip-172-31-34-103:~$

```

Here we've launched a new container called `clarusway` using the docker `run command` and the name of the image we've just created. We've specified the `-d` option, which tells Docker to *run detached* in the background. This allows us to run long-running processes like the Nginx daemon. We've also specified a command for the container to run: `nginx -g "daemon off;"`. This will launch Nginx in the *foreground* to run our web server.

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

- Docker can randomly assign a high port from the range 32768 to 61000 on the Docker host that maps to port 80 on the container.
- You can specify a specific port on the Docker host that maps to port 80 on the container.

The docker run command will open a random port on the Docker host that will connect to port 80 on the Docker container.

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

```
ubuntu@ip-172-31-34-103:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
4202131d6dfa        clarusway_web       "nginx -g 'daemon of…"   4 minutes ago
ubuntu@ip-172-31-34-103:~$
```

We see that port 32768 is mapped to the container port of 80. We can get the same information with the `docker port` command

```
ubuntu@ip-172-31-34-103:~$ sudo docker port 4202131d6dfa 80
0.0.0.0:32768

```

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.

Set up your Docker Hub account

If you don't have a Docker ID, follow these steps to create one. A Docker ID allows you to share images on Docker Hub.

1. Visit the Docker Hub sign up page.
2. Fill out the form and submit to create your Docker ID.
3. Verify your email address to complete the registration process.
4. Click on the Docker icon in your toolbar or system tray, and click Sign in / Create Docker ID.
5. Fill in your new Docker ID and password. After you have successfully authenticated, your Docker ID appears in the Docker Desktop menu in place of the 'Sign in' option you just used.

You can also sign into Docker Hub from the command line by typing docker login.

Create a Docker Hub repository and push your image

1. Click on the Docker icon in your menu bar, and navigate to **Repositories>Create.** You'll be redirected to the **Create Repository** page on Docker Hub.
2. Type the repository name and click Create at the bottom of the page. Do not fill any other details for now.
3. You are now ready to share your image on Docker Hub, however, there's one thing you must do first: images must be namespaced correctly to share on Docker Hub. Specifically, you must name images like /:.

Tips:

- 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.

```
ubuntu@ip-172-31-34-103:~$ sudo docker push hub_user_name/image_name

```

Deleting an image

We can also delete images with `docker rmi` (docker image rm) command.

```
ubuntu@ip-172-31-34-103:~$ sudo docker rmi clarusway_web
Untagged: clarusway_web:latest
Deleted: sha256:3978e21208b843f54a49be6423c8da04881c3db91df484e56a00655391988451
Deleted: sha256:cb9ea17024c282150438a02b0bc86a2510993661eac196c4224fdd23dca415d1
Deleted: sha256:cfb8f83c1819c272c108d4d71092ea9c9deb2c62b94358054d7585af83097c7b
Deleted: sha256:e9a1752c7f16b77e62c44878c6a95c71ef8360e15b56fece7ce60656336189000
Deleted: sha256:91637e878a27287485535aeb784eeeb0fe022a6416592fb7163fee6f37e785ff
Deleted: sha256:8718cc340c9b71092618166c6abec2c3a136d5f557aacdaed038d066353d9899
ubuntu@ip-172-31-34-103:~$

```