docker

W RKSHOP

Barry Evans: using some slides from Stefan Koospal & Mohsen Haghaieghshenasfard

Pictures: Stefan Koospal

Agenda

- **Why?**
- Installing Docker on Linux Running Your First Image
- The Basic Commands
- Create a Dockerized Cowsay Application
- Building Images from Dockerfile
- Working with Registries
- Docker Fundamentals
- Dockerfile Instructions
- Connecting Containers to the World
- Common Docker Commands
- Managing Containers
- Practical Section

Why?

Resources
Updates
Fail Safety
Recovery
Roll Back
Security
Portability
Easy Application Delivery

Virtualisation

Hypervisor (different OS)

- VM
- XEN
- Virtual Box

OS-Container (Using one Kernel)

- openvz
- zone
- jails
- lxc

Docker

What is the difference to lxc, openvz, jails, zones?

- Using only kernel and network
- No special kernel
- Using layers
- Offers repositories
- Offers orchestration

Docker (Wikipedia I)

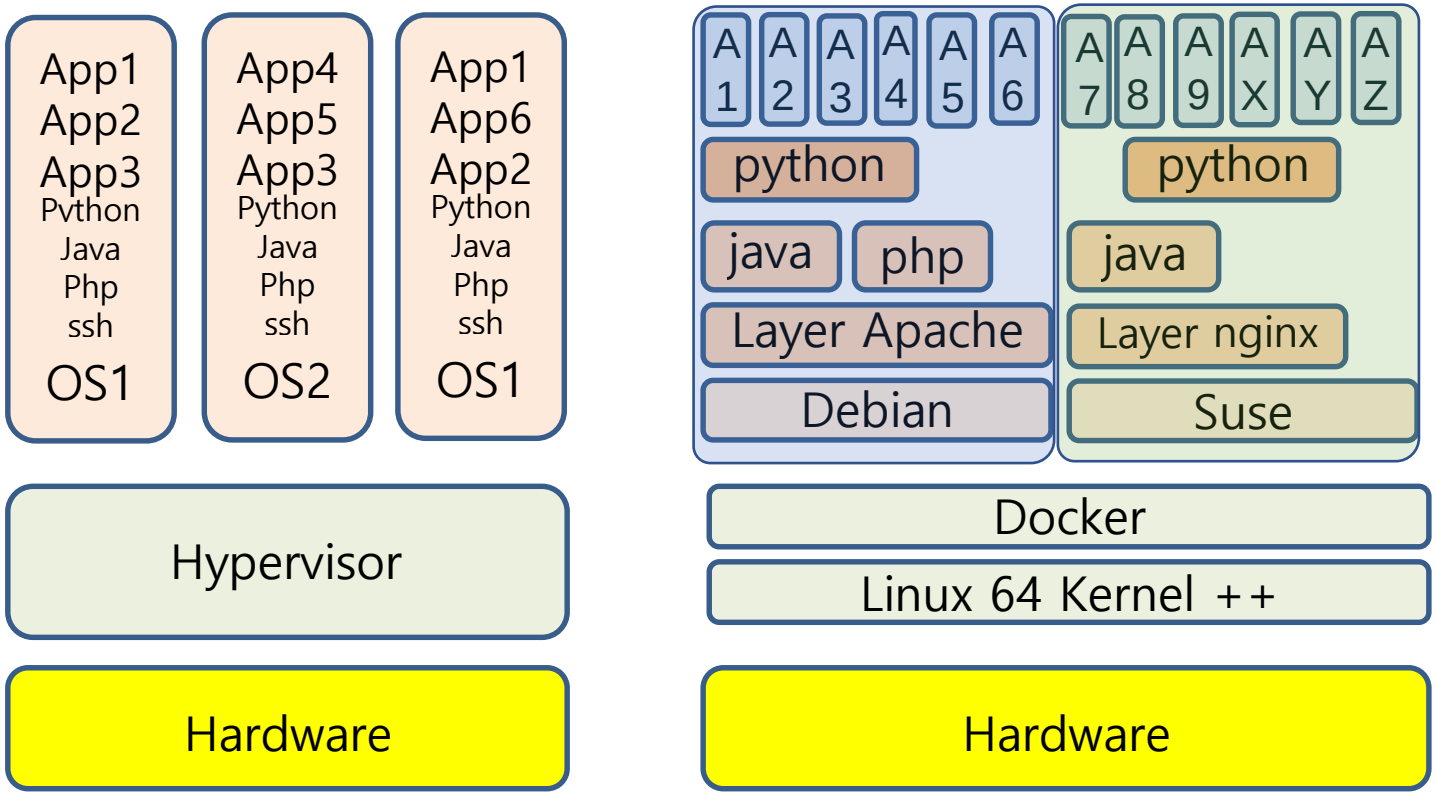
- Container
  - Is a running virtual OS executing one ore more applications
- Image
  - Is a portable memory image to run as a container
- Dockerfile
  - Is a textfile with commands to create an image

Docker (Wikipedia II)

- Docker Hub
  - A registry to store docker images

- libcontainer
  - An interface to basic functions of docker
- Libswarm (Kubernetes)
  - An interface for orchestration
- libchan
  - An interface to the docker network

Hypervisor-VMs Versus Docker Containers



Containers Versus VMs (cont.)

- Containers share resources with the host OS, which makes them an order of magnitude more efficient. Containers can be started and stopped in a fraction of a second.
- The portability of containers has the potential to eliminate a whole class of bugs caused by subtle changes in the running environment.
- The lightweight nature of containers means developers can run dozens of containers at the same time, making it possible to emulate a production-ready distributed system.
- Users can download and run complex applications without needing to spend hours on configuration and installation issues.

The What and Why of Containers

- Containers are fundamentally changing the way we develop, distribute, and run software. Developers can build software locally, knowing that it will run identically regardless of host environment.
- Operations engineers can concentrate on networking, resources, and uptime and spend less time configuring environments. Containers are also an encapsulation of an application with its dependencies.
- Docker containers share the underlying resources of the Docker host.

Containers are very small (some base OS images are less than 3MBs) start up very quickly (< 3/8s of a second) because you're not booting a full operating system. You're just starting a process.

The What and Why

Before Docker

- Ship packages: deb, rpm, gem, jar...
- Dependency hell.
- "Works on my machine."
- Base deployment often done from scratch (debootstrap...) and unreliable.

After Docker

- Ship container images with all their dependencies.
- Break image into layers.
- Only ship layers that have changed.
- Save disk, network, memory usage.

Docker and Containers

Containers are an old concept. Some examples are:

- UNIX systems have had the chroot command that provides a simple form of filesystem isolation.
- FreeBSD has had the jail utility, which extended chroot sandboxing to processes.
- Solaris Zones offered a comparatively complete containerization technology around 2001 but was limited to the Solaris OS.

But:

Docker took the existing Linux container technology and wrapped and extended it in various ways—primarily through portable images and a user-friendly interface—to create a complete solution for the creation and distribution of containers.

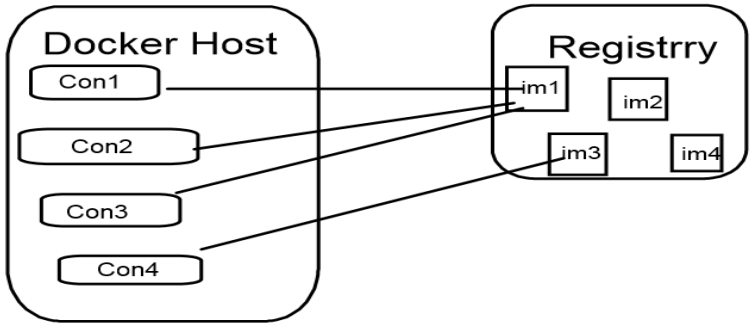
Docker Components

The Docker platform has two main components:

- **Docker host** which provides a fast and convenient interface for creating images and running containers.

- **Registry Service (Docker Hub or Docker Trusted Registry),** Cloud or server based storage and distribution service for your images. It provides an enormous number of public container images for download, allowing users to quickly get started and avoid duplicating work already done by others.

Docker Components (cont.)



Agenda

- The What and Why of Containers
- **Installing Docker on Linux Running Your First Image**
- The Basic Commands
- Create a Dockerized Cowsay Application
- Building Images from Dockerfile
- Working with Registries
- Docker Fundamentals
- Dockerfile Instructions
- Connecting Containers to the World
- Common Docker Commands
- Managing Containers
- Practical Section

Installing Docker on Linux

By far the best way to install Docker on Linux is through the **installation script** provided by Docker.

You should be able to the use the script provided at the following link to automatically install Docker. The official instructions will tell you to simply run:

curl -sSL https://get.docker.com/ | sh

wget -qO- https://get.docker.com/ | sh

Note: installing Docker Requires 64 bit Linux and a least **kernel 3.10**

A Quick Check

Just to make sure everything is installed correctly and working, try running the **docker version** command. The output is like this:

```
Client:
 Version:      17.12.0-ce
 API version:  1.35
 Go version:   go1.9.2
 Git commit:   c97c6d6
 Built:        Wed Dec 27 20:11:19 2017
 OS/Arch:      linux/amd64

Server:
 Engine:
  Version:      17.12.0-ce
  API version:  1.35 (minimum version 1.12)
  Go version:   go1.9.2
  Git commit:   c97c6d6
  Built:        Wed Dec 27 20:09:54 2017
  OS/Arch:      linux/amd64
  Experimental:                false
```

Running Your First Image

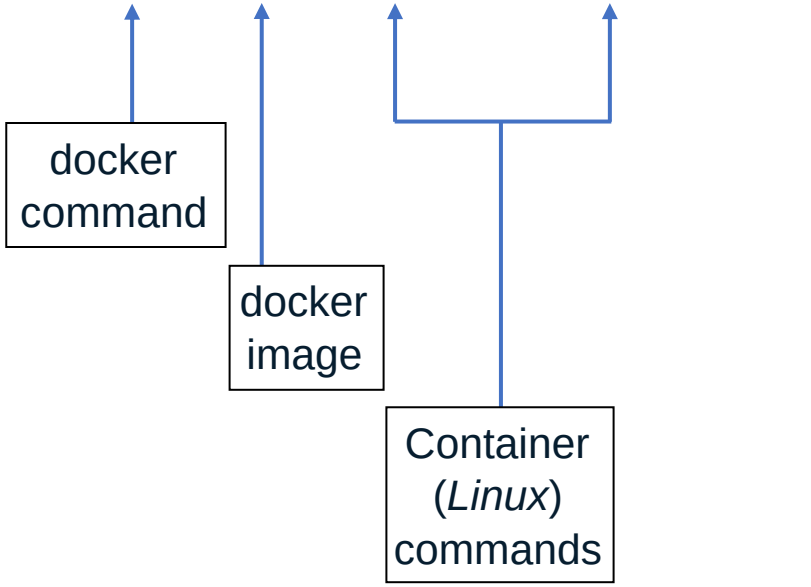
To test Docker is installed correctly, try running:

```
# docker run ubuntu echo "Hello DubJUG"
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
1be7f2b886e8: Pull complete
6fbc4a21b806: Pull complete
c71a6f8e1378: Pull complete
4be3072e5a37: Pull complete
06c6d2f59700: Pull complete
Digest: sha256:e27e9d7f7f28d67aa9e2d7540bdc2b33254b452ee8
e60f388875e5b7d9b2b696
Status: Downloaded newer image for ubuntu:latest
Hello DubJUG
```

What happens now?

And why?

```
# docker run ubuntu echo "Hello DubJUG"
```



Differences between containers and images

- An image is a read-only filesystem.
- A container is an encapsulated set of processes running in a read-write copy of that filesystem.
- To optimize container boot time, copy-on-write is used instead of regular copy.
- **docker run** starts a container from a given image.

Running Your First Image (cont.)

We can ask Docker to give us a shell inside a container with the following command:

```
# docker run -i -t ubuntu "/bin/bash"
root@5aadb5ce8631:/# echo "Hello Container world"
Hello Container world
root@5aadb5ce8631:/# exit
exit
```

Agenda

- The What and Why of Containers
- Installing Docker on Linux Running Your First Image
- **The Basic Commands**
- Create a Dockerized Cowsay Application
- Building Images from Dockerfile
- Working with Registries
- Docker Fundamentals
- Dockerfile Instructions
- Connecting Containers to the World
- Common Docker Commands
- Managing Containers
- Practical Section

The Basic Commands

Let's try to understand Docker a bit more by launching a container and seeing what effect various commands and actions have. First, let's launch a new container; but this time, we'll give it a new hostname with the -h flag:

```
# docker run -h container -i -t ubuntu "/bin/bash"
root@container:/#
```

The name of container may be **infallible_bhaskara**. Docker-generated names are a random adjective followed by the name of a famous scientist, engineer, or hacker. You can instead set the name by using the --name argument.

The Basic Commands (cont.)

Get more information on a given container by running **docker inspect** with the name or ID of the container:

docker inspect infallible_bhaskara


```
[
    {
        "Id":
"f5b0bd3817f632ad5e30efc13cd12fbe1e613a32990ab42f75fea332dc546cef",
        "Created": "2018-02-06T16:51:51.25395522Z",
        "Path": "/bin/bash",
        "Args": [],
        "State": {
            "Status": "running",
```

The Basic Commands (cont.)

Use **grep** or the **--format argument** to filter for the information we're interested in.

```
# docker inspect infallible bhaskara | grep IPAddress
            "SecondarvIPAddresses": null.
            "IPAddress": "172.17.0.4".
# docker inspect --format {{.NetworkSettings.IPAddress}} infallible bhaskara
172.17.0.4
```

The Basic Commands (cont.)

docker diff:

root@container:/tmp# touch /tmp/xx

docker diff infallible_bhaskara

C /tmp

A /tmp/xx

Here is the list of files that have changed in the running container compared with the original image.

The Basic Commands (cont.)

docker logs:

docker logs infallible_bhaskara

root@container:/# cd tmp

root@container:/tmp# touch /tmp/xx

If you run this command with the name of your container, you will get a list of everything that’s happened inside the container:

The Basic Commands (cont.)

docker rm:

To get rid of the container, use the docker rm command

docker rm infallible_bhaskara

If you want to get rid of all your stopped containers, you can use the following command which gets the IDs of all stopped containers. For example:

docker rm -v $(docker ps -aq -f status=exited)

Agenda

- The What and Why of Containers
- Installing Docker on Linux Running Your First Image
- The Basic Commands
- **Create a Dockerized Cowsay Application**
- Building Images from Dockerfile
- Working with Registries
- Docker Fundamentals
- Dockerfile Instructions
- Connecting Containers to the World
- Common Docker Commands
- Managing Containers
- Practical Section

Create a Dockerized Cowsay Application

```
# docker run --name cowsay -h cowsay  -i -t ubuntu "/bin/bash"
root@cowsay:/# apt-get update
...
root@cowsay:/# apt-get install -y fortune-mod cowsay
...
root@cowsay:/# /usr/games/fortune | /usr/games/cowsay

< You will pass away very quickly. >
 ---------------------------------
        ₩    ^    ^
         ₩  (oo)₩
            (   )₩        )₩/₩
                ||----w |
                ||      ||
```

The Basic Commands (cont.)

docker commit:

To turn the cowsay container into an image, use the docker commit command. It doesn’t matter if the container is running or stopped·

root@cowsay:/ # exit

exit

docker commit cowsay dubjug/cowsay

sha256:7a09e1aa2872ff37258e0557670bd8d9e166ddd9a5b400d510d5ac77c9b23ab2

The returned value is the unique ID of our image.

The Basic Commands (cont.)

Now we have an image with cowsay installed that we can run:

```
~/cowsay# docker run dubjug/cowsay "/usr/games/cowsay" "Muh"

< Muh >
 -----
        ₩   ^   ^
         ₩  (oo)₩
            (   )₩        )₩/₩
                ||----w |
                ||      ||
```

This is great! However, there are a few problems. If we need to change something, we have to manually repeat our steps from that point.

The Basic Commands (cont.)

For example, if we want to use a different base image;

- we would have to start again from scratch.
- More importantly, it isn't easily repeatable; it's difficult and potentially error-prone to share or repeat the set of steps required to create the image.

The solution to this is to use a **Dockerfile** to create an automated build for the image.

Agenda

- The What and Why of Containers
- Installing Docker on Linux Running Your First Image
- The Basic Commands
- Create a Dockerized Cowsay Application
- **Building Images from Dockerfile**
- Working with Registries
- Docker Fundamentals
- Dockerfile Instructions
- Connecting Containers to the World
- Common Docker Commands
- Managing Containers
- Practical Section

Building Images from Dockerfile

A Dockerfile is simply a text file that contains a set of steps that can be used to create a Docker image. Start by creating a new folder and file for this example:

mkdir cowsay

cd cowsay

~/cowsay# **touch Dockerfile**

And insert the following contents into Dockerfile:

FROM ubuntu

RUN apt-get update && apt-get install -y fortune-mod cowsay

Building Images from Dockerfile (cont.)

We can now build the image by running the **docker build** command inside the same directory:

~/cowsay# **ls Dockerfile**

Dockerfile

~/cowsay# **docker build -t dubjug/cowsay-dockerfile .**

Sending build context to Docker daemon 2.048kB

Step 1/2 : FROM ubuntu

 ---> 0458a4468cbc

Step 2/2 : RUN apt-get update && apt-get install -y fortune-mod cowsay

 ---> Running in 7ddeeca5dca9

….

removing intermediate container 7ddeeca5dca9

 ---> 72359aa0bff8

Successfully built 72359aa0bff8

Successfully tagged dubjug/cowsay-dockerfile:latest

Building Images from Dockerfile (cont.)

Then we can run the image in the same way as before:

~/cowsay# **docker run dubjug/cowsay-dockerfile "/usr/games/cowsay" "Muh"**

```
 _____
< Muh >
 -----
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Building Images from Dockerfile (cont.)

But we can actually make things a little bit easier for the user by taking advantage of the ENTRYPOINT Dockerfile instruction. The ENTRYPOINT instruction lets us specify an executable that is used to handle any arguments passed to docker run.

Add the following line to the bottom of the Dockerfile:

ENTRYPOINT "/usr/games/cowsay" "Muh“

~/cowsay# **docker build -t dubjug/cowsay-dockerfile .**

~/cowsay# **docker run dubjug/cowsay-dockerfile**

```
 _____
< Muh >
 -----
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Building Images from Dockerfile (cont.)

Much easier! But now we've lost the ability to use the fortune command inside the container as input to cowsay.

We can fix this by providing our own script. Create a file app.sh with the following contents and save it in the same directory as the Dockerfile.

Building Images from Dockerfile (cont.)

```
#!/bin/bash
if [ $# -eq 0 ]
then
 /usr/games/fortune | /usr/games/cowsay
else
 /usr/games/cowsay "$@"
fi
```

Set the file to be executable with:

~/cowsay# **chmod +x app.sh**

Building Images from Dockerfile (cont.)

We next need to modify the Dockerfile to add the script into the image and call it as argument running the container. Edit the Dockerfile so that it looks like:

```
FROM ubuntu
RUN apt-get update && apt-get install -y fortune-mod cowsay
COPY app.sh /
```

The COPY instruction simply copies a file from the host into the image's filesystem, the first argument being the file on the host and the second the destination path, very similar to cp.

Building Images from Dockerfile (cont.)

Try building a new image and running the container starting app.sh without arguments:

~/cowsay# **docker build -t dubjug/cowsay-dockerfile .**

~/cowsay# **docker run dubjug/cowsay-dockerfile "./app.sh"**

```
 _____________________________
< Be different: conform. >
 -----------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Building Images from Dockerfile (cont.)

And with arguments:

~/cowsay# **docker run dubjug/cowsay-dockerfile "./app.sh" "Muh" "Muh"**

```
 ___________
< muh muh >
 ----------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Building Images from Dockerfile (cont.)

how do we persist and **back up** our data?

For this, we don't want to use the standard container filesystem—instead we need something that can be easily shared between the container and the host or other containers. Docker provides this through the concept of volumes.

Volumes are files or directories that are directly mounted on the host and not part of the normal union file system. This means they can be shared with other containers and all changes will be made directly to the host filesystem.

Building Images from Dockerfile (cont.)

Volumes:

There are two ways of declaring a directory as a volume;

- using the VOLUME instruction inside a Dockerfile

  Volume /data

- specifying the -v flag to docker run.

#**docker run --name dhost -h dhost -v /data -i -t ubuntu "/bin/bash"**

Both the following Dockerfile instruction and docker run command have the effect of creating a volume as /data inside a container.

Building Images from Dockerfile (cont.)

Volumes (cont.)

- By default, the directory or file will be mounted on the host inside your Docker installation directory (normally /var/lib/docker/ non persistent).
- It is possible to specify the host directory to use as the mount via the docker run command (this directory is persistent)

#**mkdir -p /vol/dhost**

#**docker run --name dhost -h dhost -v /vol/dhost:/data -i -t ubuntu "/bin/bash"**

- It isn’t possible to specify a host directory inside a Dockerfile for reasons of portability and security (the file or directory may not exist in other systems, and containers shouldn’t be able to mount sensitive files like etc/passwd without explicit permission).

Agenda

- The What and Why of Containers
- Installing Docker on Linux Running Your First Image
- The Basic Commands
- Create a Dockerized Cowsay Application
- Building Images from Dockerfile
- **Working with Registries**
- Docker Fundamentals
- Dockerfile Instructions
- Connecting Containers to the World
- Common Docker Commands
- Managing Containers
- Practical Section

Working with Registries

Now that we've created something amazing, how can we **share** it with others?

- When we first ran the Debian image at the start of the workshop, it was downloaded from the **official Docker registry**—the **Docker Hub**.

- Similarly, we can upload our own images to the Docker Hub for **others** to download and use.

Working with Registries (cont.)

In order to upload our cowsay image, you will need:

- to **sign up** for an account with the Docker Hub;
- Then, **tag the image** into an appropriately named repository and use the **docker push** command to upload it to the Docker Hub.

Working with Registries (cont.)

Before that, add a **MAINTAINER** instruction to the Dockerfile, which simply sets the author contact information for the image:

```
FROM ubuntu
MAINTAINER Barry Evans <barryevans80@gmail.com>
```

Working with Registries (cont.)

Now rebuild the image and upload it to the Docker Hub. This time, you will need to use a **repository name** that starts with your username on the Docker Hub (in this case barryevans80), followed by / and whatever name you want to give the image. For example:

~/cowsay# **docker build -t barryevans80/cowsay-dubjug .**

~/cowsay# **docker login**

~/cowsay# **docker push barryevans80/cowsay-dubjug**

Working with Registries (cont.)

As I didn’t specify a tag after the repository name, it was automatically assigned the latest tag. To specify a tag, just add it after the repository name with a colon.

#**docker build -t barryevans80/cowsay-dubjug:stable .**

Once the upload has completed, the world can download your image via the docker pull command:

#**docker pull barryevans80/cowsay-dubjug**

Pull openjdk from Docker-Registry

docker pull openjdk

mkdir openjdk-jshell; cd openjdk-jshell

Create Dockerfile for jshell

https://github.com/docker-library/openjdk/blob/cbbefa82b92964e6fd98b20353be7010f970c60a/9-jdk/Dockerfile

docker build -t test/openjdk-jshell .

docker run -i -t test/openjdk-jshell

Feb 08, 2018 1:34:22 PM java.util.prefs.FileSystemPreferences$1 run

INFO: Created user preferences directory.

| Welcome to JShell -- Version 9.0.1

| For an introduction type: /help intro

jshell>

Agenda

- The What and Why of Containers
- Installing Docker on Linux Running Your First Image
- The Basic Commands
- Create a Dockerized Cowsay Application
- Building Images from Dockerfile
- Working with Registries
- **Docker Fundamentals**
- Dockerfile Instructions
- Connecting Containers to the World
- Common Docker Commands
- Managing Containers
- Practical Section

The Docker Architecture

The major components of a Docker installation:

- **Docker daemon**, which is responsible for creating, running, and monitoring containers, as well as building and storing images, and launched by running docker daemon, which is normally taken care of by the host OS.

- **Docker client** is used to talk to the Docker daemon via HTTP. By default, this happens over a Unix domain socket, but it can also use a TCP socket to enable remote clients or a file descriptor for system-managed sockets.

- **Docker registries** store and distribute images.

The Docker Architecture (cont.)

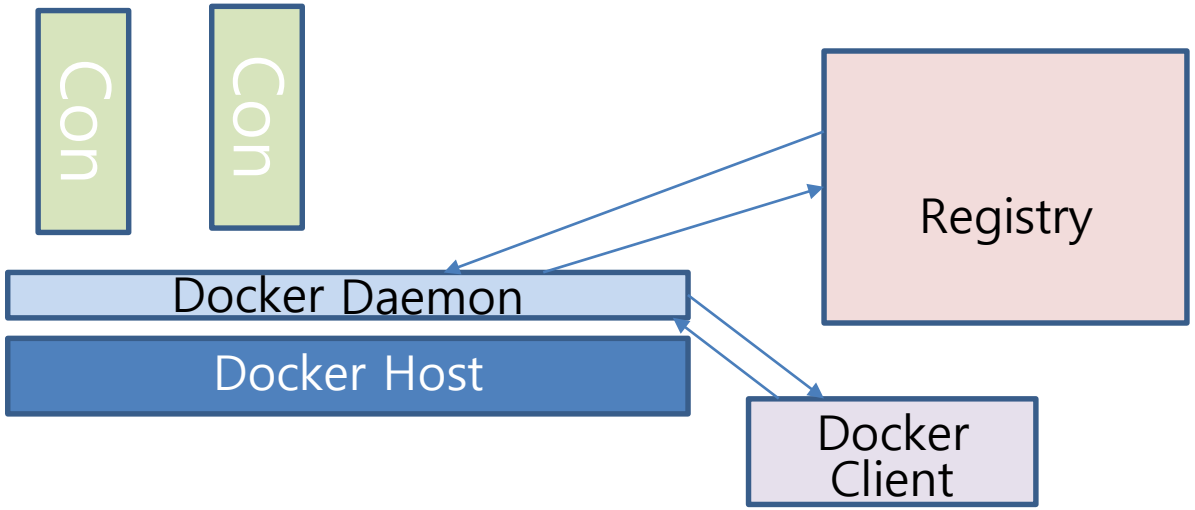


Image Layer

- Each instruction in a Dockerfile results in a new **image layer**, which can also be used to start a container. The new layer is created by starting a container using the image of the previous layer, executing the Dockerfile instruction and saving a new image.

- When a Dockerfile instruction successfully completes, the intermediate container will be deleted. Since each instruction results in a static image — essentially just a filesystem and some metadata—all running processes in the instruction will be stopped.

- If you want a service or process to start with the container, it must be launched from an **ENTRYPOINT** or **CMD** instruction.

Image Layer (cont.)

You can see the full set of layers that make up an image by running the docker history command. One example is:

~/cowsay# **docker history barryevans80/cowsay-dubjug**

IMAGE	CREATED	CREATED BY	SIZE	COMMENT
49e038393108	25 minutes ago	/bin/sh -c #(nop) COPY file:c22006eaeae75fd8…	103B	
5ed11d75f720	25 minutes ago	/bin/sh -c apt-get update && apt-get install…	85.4MB	
534160f2aa5d	25 minutes ago	/bin/sh -c #(nop) MAINTAINER Barry Evans…	0B	
0458a4468cbc	2 weeks ago	/bin/sh -c #(nop) CMD ["/bin/bash"]	0B	

Base Images

When creating your own images, you will need to decide which base image to start from:

- The best-case scenario is just to use an **existing image** and mount your configuration files and/or data into it. This is to be the case for common application software, such as databases and web servers, where there are official images available.

- Sometimes you really just need a small but **complete Linux distro**. The alpine image, which is only just over 5 MB in size but still has an extensive packager manager for easily installing applications and tools. The Debian images are second option.