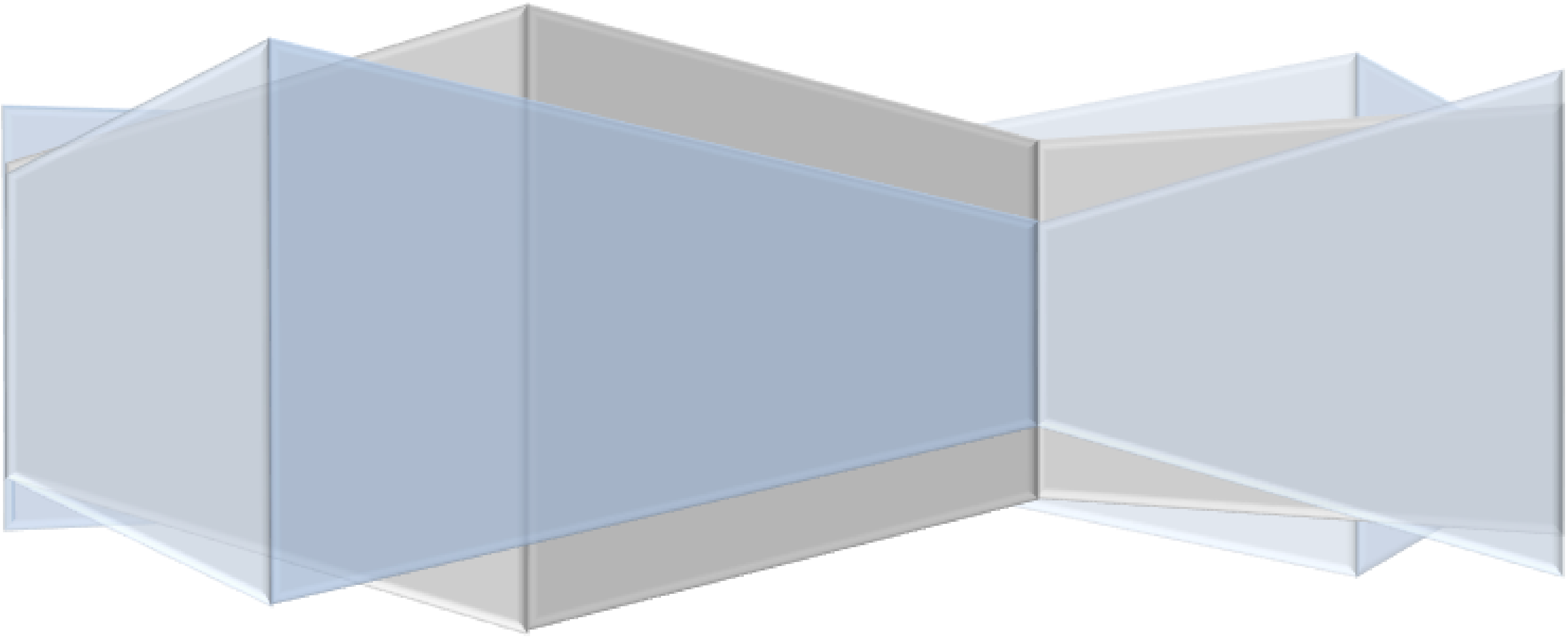
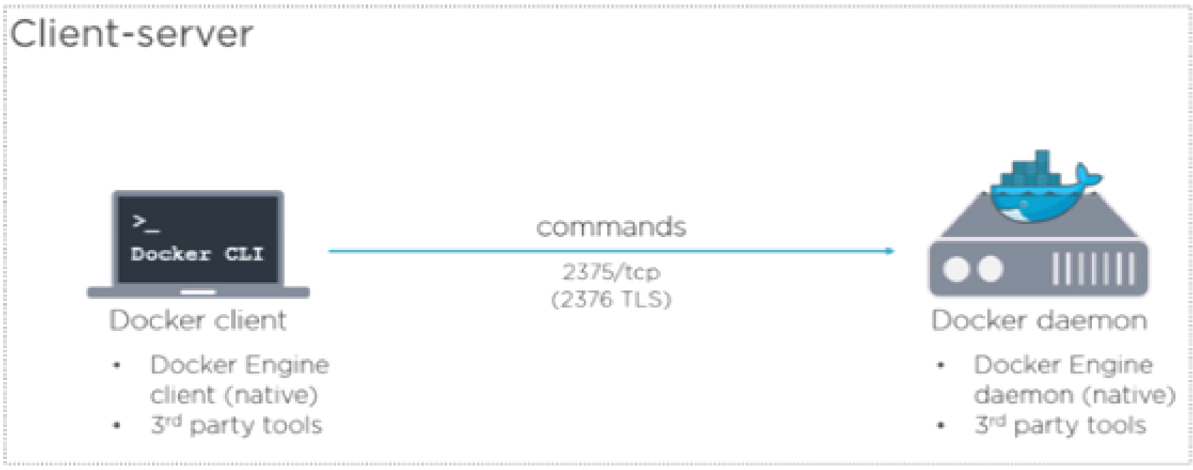
Using Docker

Ganesh Palnitkar



Docker is a Client-server architecture application with Client as an API and server as the Docker Daemon.



Client-server

Docker client

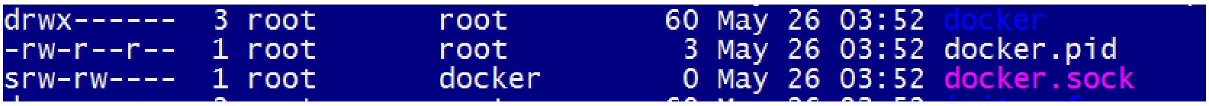
- Docker Engine client (native)
- 3rd party tools

Docker daemon

- Docker Engine daemon (native)
- 3rd party tools

```
$ ls -l /run
```

This shows that the docker is running on the UNIX socket.



```
drwx------   3 root        root          60 May 26 03:52 docker
-rw-r--r--   1 root        root           3 May 26 03:52 docker.pid
srw-rw----   1 root        docker         0 May 26 03:52 docker.sock
```

In order to make Docker run on TCP port run below command,

```
$ netstat -ntlp
```

This will show current programs listening on TCP ports.

Stop the docker service and restart it using below command to make it listen on a TCP port.

`$ docker -H 192.168.33.35:2375 -d &` …… in this command, we are making the service to start on TCP port in daemon mode.

This can also be set by editing docker.service

```
$ sudo systemctl edit docker.service
```

Add below text to the file,

```
[Service]
ExecStart=
ExecStart=/usr/bin/dockerd -H fd:// -H tcp://127.0.0.1:2375
```

Now, restart the Docker service,

```
$ systemctl daemon reload
```

```
$ systemctl restart docker
```

In order to connect to a docker host over TCP port from remote machine, set the environment variable on remote machine as,

```
export DOCKER_HOST="tcp://192.168.33.35:2375"
```

This connects to the remote docker running on 192.168.33.35 machine.

To set it back to local Linux socket,

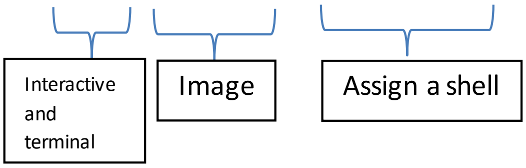
`export DOCKER_HOST=` … this will start listening docker back to local port.

We can also make the docker to run and listen on both Linux socket and TCP port.

```
$ docker -H 192.168.33.35:2375 -H unix:///var/run/docker.sock -d &
```
… this will run docker on both ports.

Docker Images are used for launching docker containers.

```
$ docker run -it fedora /bin/bash
```



`$ docker pull -a <image name>` , or,

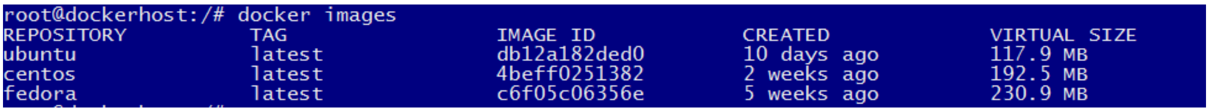
`$ docker pull <image name>` … this download image mentioned. We can view all available images on local machine, by using command,

`$ docker images <image name>` … this will list all downloaded images

Images are stored in Linux under `/var/lib/docker/aufs (storage driver)`

Docker Containers:

`$ docker images` …lists down all docker images available on the docker host machine.



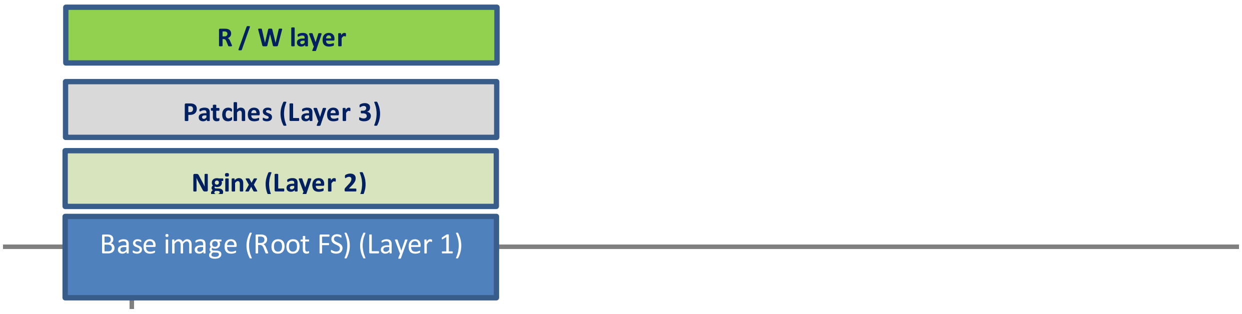
```
root@dockerhost:/# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
ubuntu              latest              db12a182ded0        10 days ago         117.9 MB
centos              latest              4beff0251382        2 weeks ago         192.5 MB
fedora              latest              c6f05c06356e        5 weeks ago         230.9 MB
```

We can exit container without killing the container by using keys `'ctrl + P + Q'` …

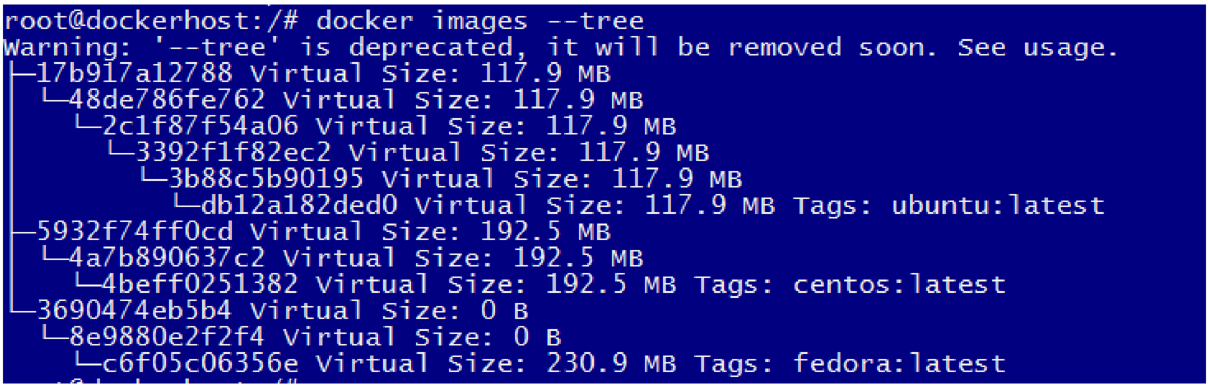
Using these images docker creates and runs a container when we run the docker run command.

Image Layers:

A docker container is formed using multiple images stacked on each other.
Union mount system helps to mount multiple filesystem components on to each other.



`$ docker images -tree` ... view of the image layers.



```
root@dockerhost:/# docker images --tree
Warning: '--tree' is deprecated, it will be removed soon. See usage.
├─17b917a12788 Virtual Size: 117.9 MB
│ └─48de786fe762 Virtual Size: 117.9 MB
│   └─2c1f87f54a06 Virtual Size: 117.9 MB
│     └─3392f1f82ec2 Virtual Size: 117.9 MB
│       └─3b88c5b90195 Virtual Size: 117.9 MB
│         └─db12a182ded0 Virtual Size: 117.9 MB Tags: ubuntu:latest
├─5932f74ff0cd Virtual Size: 192.5 MB
│ └─4a7b890637c2 Virtual Size: 192.5 MB
│   └─4beff0251382 Virtual Size: 192.5 MB Tags: centos:latest
└─3690474eb5b4 Virtual Size: 0 B
  └─8e9880e2f2f4 Virtual Size: 0 B
    └─c6f05c06356e Virtual Size: 230.9 MB Tags: fedora:latest
```

`$ docker history <image name>` ... this will also give detailed info about the docker images and layers.

Now try creating a new image from an existing container. First start a container with new changes, like,

```
$ docker run ubuntu /bin/bash -c "touch file1 | echo 'hello world' > file1"
```

Now run `$ docker ps -a`

Listing the container on the docker host can also be done using the command,

```
$ docker container ls --all --filter status=existed
```

```
$ docker rm $( docker container ls --all --filter STATUS=existed –format “{{ .ID }}”)
```

Run the command to create an image from the latest container.

```
$ docker commit <container id> <new-image name>
```

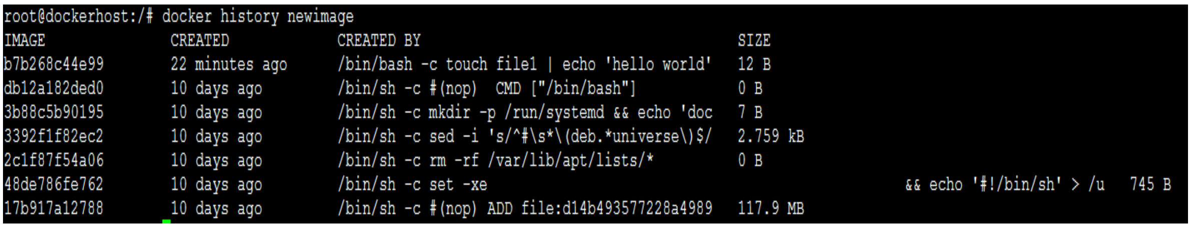
e.g.

```
$ docker commit d92a9d94aea8 newimage
```

This will create an image as ‘newimage’ and we can make use of this to create a new container.

The history command will give more info about the image layers.

```
$ docker history newimage
```



```
root@dockerhost:/# docker history newimage
IMAGE               CREATED             CREATED BY                                      SIZE
b7b268c44e99        22 minutes ago      /bin/bash -c touch file1 | echo 'hello world'   12 B
db12a182ded0        10 days ago         /bin/sh -c #(nop)  CMD ["/bin/bash"]            0 B
3b88c5b90195        10 days ago         /bin/sh -c mkdir -p /run/systemd && echo 'doc   7 B
3392f1f82ec2        10 days ago         /bin/sh -c sed -i 's/^#\s*\(deb.*universe\)$/   2.759 kB
2c1f87f54a06        10 days ago         /bin/sh -c rm -rf /var/lib/apt/lists/*          0 B
48de786fe762        10 days ago         /bin/sh -c set -xe                                                   && echo '#!/bin/sh' > /u   745 B
17b917a12788        10 days ago         /bin/sh -c #(nop) ADD file:d14b493577228a4989   117.9 MB
```

The layers shows above are the locked layers. When we use it to run a container it adds a writable layer which allows making changes to the files inside the locked layers.

Now to allow this image to be shared with others we can use below command. **Export image..**

`docker save –o <location to save image> <image name>`

`$ docker save –o /tmp/newimage newimage`

The image file is created as a tar file. Run tar command to check the file contents.

`$ tar –tf newimage` … this will show all layers and related file zipped in a the image file.

Now to make use of the newimage.tar image file. **Import Image…**

`$ docker load  -i newimage.tar …`

`$ docker images`

Now use `docker run` command to make use of the image.

<u>In most cases the container are run in the detached mode using the switch '–d'..</u>

The docker run commands comes with lot of options, run `$ docker run --help` to get more information about it.

IMP!!!! To get detailed information about a container, we can use command,

`$ docker inspect <container-id>`

Container Management

Docker Containers are started using `docker run` command. And can be stopped using `docker stop <container-id>.` A container can also be started using `docker start <container-id>.`

Docker container can also be stopped using command.

```
$ docker kill <container-id>
```

Docker container can be deleted using command,

```
$ docker rm <container-id>
```

The PID 1 always controls the docker container. When we kill the container we actually kill the PID 1.

!!! IMP !!! We can also use the `docker exec -it <container-id> /bin/bash` command to login to a running container and when we exit from the container shell, this does not kill the container.

We can use the `$ docker top` command to actually see all processes running inside the docker container.

Docker Hub

Public and Private repositories:

Account on the docker hub is similar to GITHUB account.

Add a tag to the existing image in order to push the image file to the Docker hub. This can be done as shown below.

```
$ docker tag <image-id> ganeshhp/helloworld:0.1.0
```

Once the image is tagged, we can then use the `docker push` command to push docker images to the remote docker repo.

```
$ docker push ganeshhp/helloworld:0.1.0
```

This will push only new layers of the images to the docker hub.

Docker registries can also be created on local server as private registries.

To create a local registry, Run below command to spin a local docker registry.

`$ docker run -d -p 5000:5000 registry` .... This will run a container in daemon mode with the network port 5000 on local server mapped to the network port 5000 on public server.

So we are actually going to start a container using the registry image on a local ubuntu server.

Docker Volumes:

Docker volumes allow sharing data outside the container. This way containers can access data outside the container.

```
$ docker run -it -v /test-volume --name=volcontainer ubuntu16.04 /bin/bash
```

```
CTRLPQ
```

Here, `–v` switch is used to specify the volume.

`--name` switch used to pass container name

Now try creating a container using above command and then run, the `$ docker inspect` command to see the volume folder on dockerhost.

This way we can share or move file from container to host machine.

Now this volume can be shared by other container as well. This is done by using option as `--volumes-from=<container-name>` switch.

```
$ docker run -it -volumes-from=volcontainer ubuntu /bin/bash
```

Once the container is started in interactive mode, check the file system for the volume folder.

This way we can map the volume from outside the container to the file system inside a container.

We can also map a folder from the host to a container . this can be done at the time of starting a container by running the command,

`$ docker run -it -v /data:/data ubuntu /bin/bash` .... This will create a folder if not present already on the host machine and map contents of it into the container. A file folder placed inside the /data folder in the container are also made available outside the container.

The same can be done from writing the `VOLUME /data` inside the Dockerfile as well. Only difference in this case is, the contents of the folder available on host machine are not mapped to the container, but vice-versa.

To delete the volume along with the container we have to use the –v switch with docker rm command as shown below,

```
$ docker rm –v <container-id>
```

Dockerfile

Dockerfile is used for building docker image. The name of the file is specific and has to be that way.

Dockerfile is written in ‘plain text’, has a ‘simple format’ and has ‘instruction to build the docker image’.

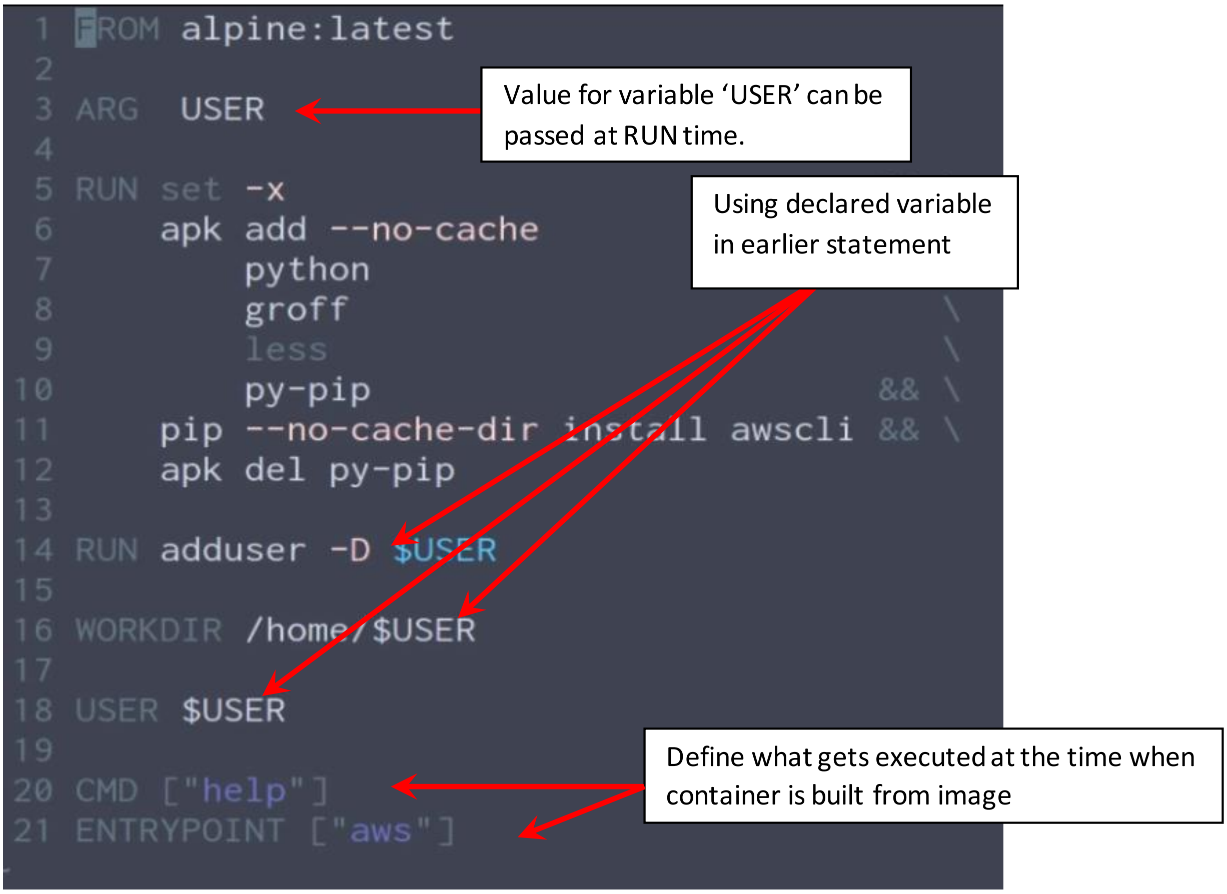
Location of the Dockerfile is important.

Create a folder and a file inside it named as Dockerfile.

```
$ mkdir dockerproj.
```

```
$ cd dockerproj
```

```
$ nano Dockerfile
```



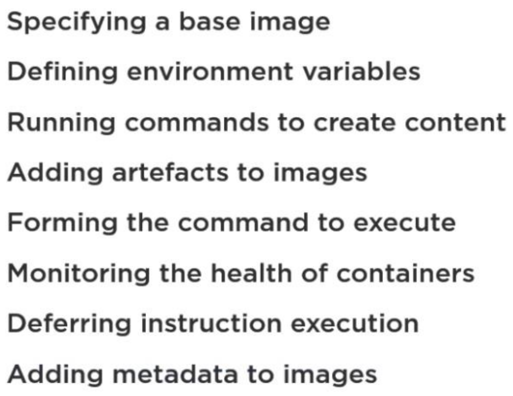
```
FROM alpine:latest

ARG  USER

RUN set -x
    apk add --no-cache
        python
        groff                                    \
        less                                     \
        py-pip                                && \
    pip --no-cache-dir install awscli        && \
    apk del py-pip

RUN adduser -D $USER

WORKDIR /home/$USER

USER $USER

CMD ["help"]
ENTRYPOINT ["aws"]
```

Value for variable ‘USER’ can be passed at RUN time.

Using declared variable in earlier statement

Define what gets executed at the time when container is built from image

Care to be taken while writing **Dockerfile**:



Specifying a base image

Defining environment variables

Running commands to create content

Adding artefacts to images

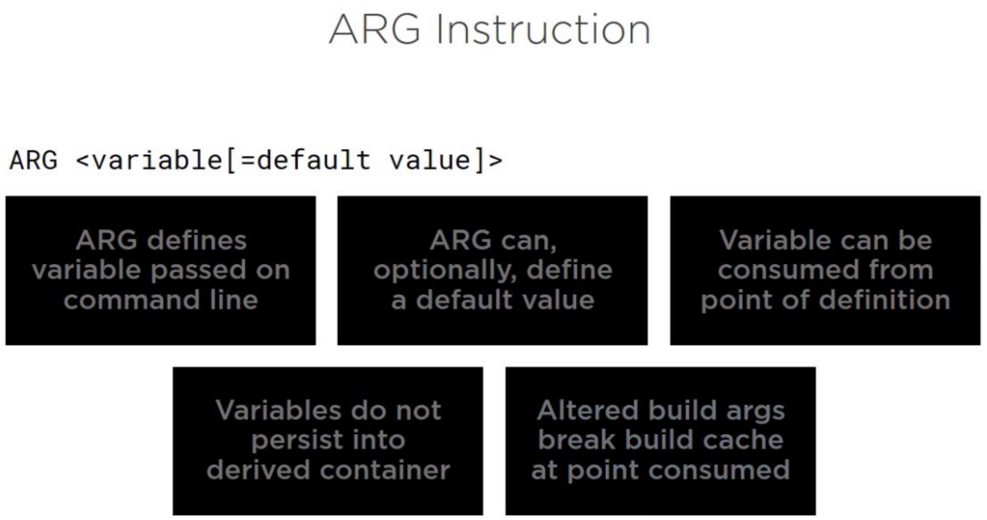
Forming the command to execute

Monitoring the health of containers

Deferring instruction execution

Adding metadata to images

ARG instruction in Dockerfile:



ARG Instruction

```
ARG <variable[=default value]>
```

ARG defines variable passed on command line

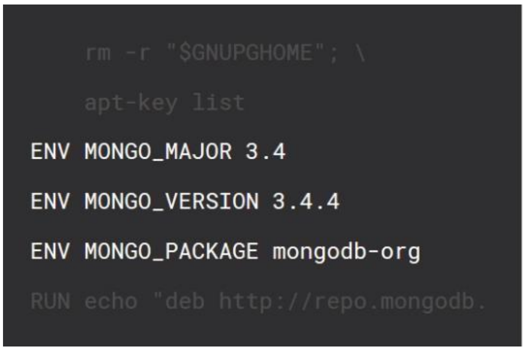
ARG can, optionally, define a default value

Variable can be consumed from point of definition

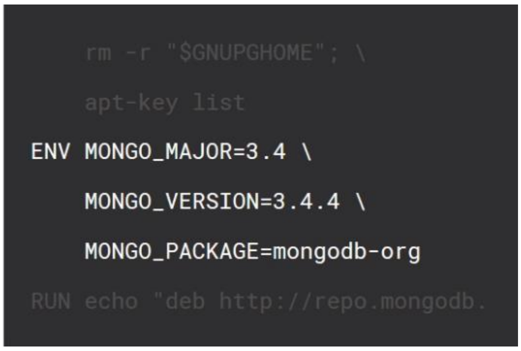
Variables do not persist into derived container

Altered build args break build cache at point consumed

ENV instruction:

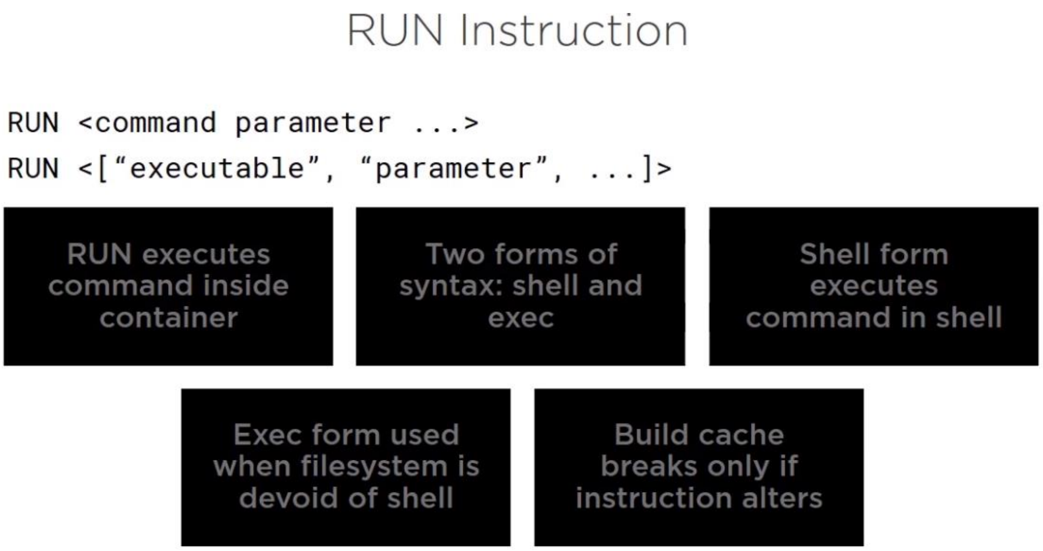


```
    rm -r "$GNUPGHOME"; \
    apt-key list
ENV MONGO_MAJOR 3.4
ENV MONGO_VERSION 3.4.4
ENV MONGO_PACKAGE mongodb-org
RUN echo "deb http://repo.mongodb.
```



```
    rm -r "$GNUPGHOME"; \
    apt-key list
ENV MONGO_MAJOR=3.4 \
    MONGO_VERSION=3.4.4 \
    MONGO_PACKAGE=mongodb-org
RUN echo "deb http://repo.mongodb.
```

RUN Instruction:



RUN Instruction

```
RUN <command parameter ...>
RUN <["executable", "parameter", ...]>
```

RUN executes command inside container

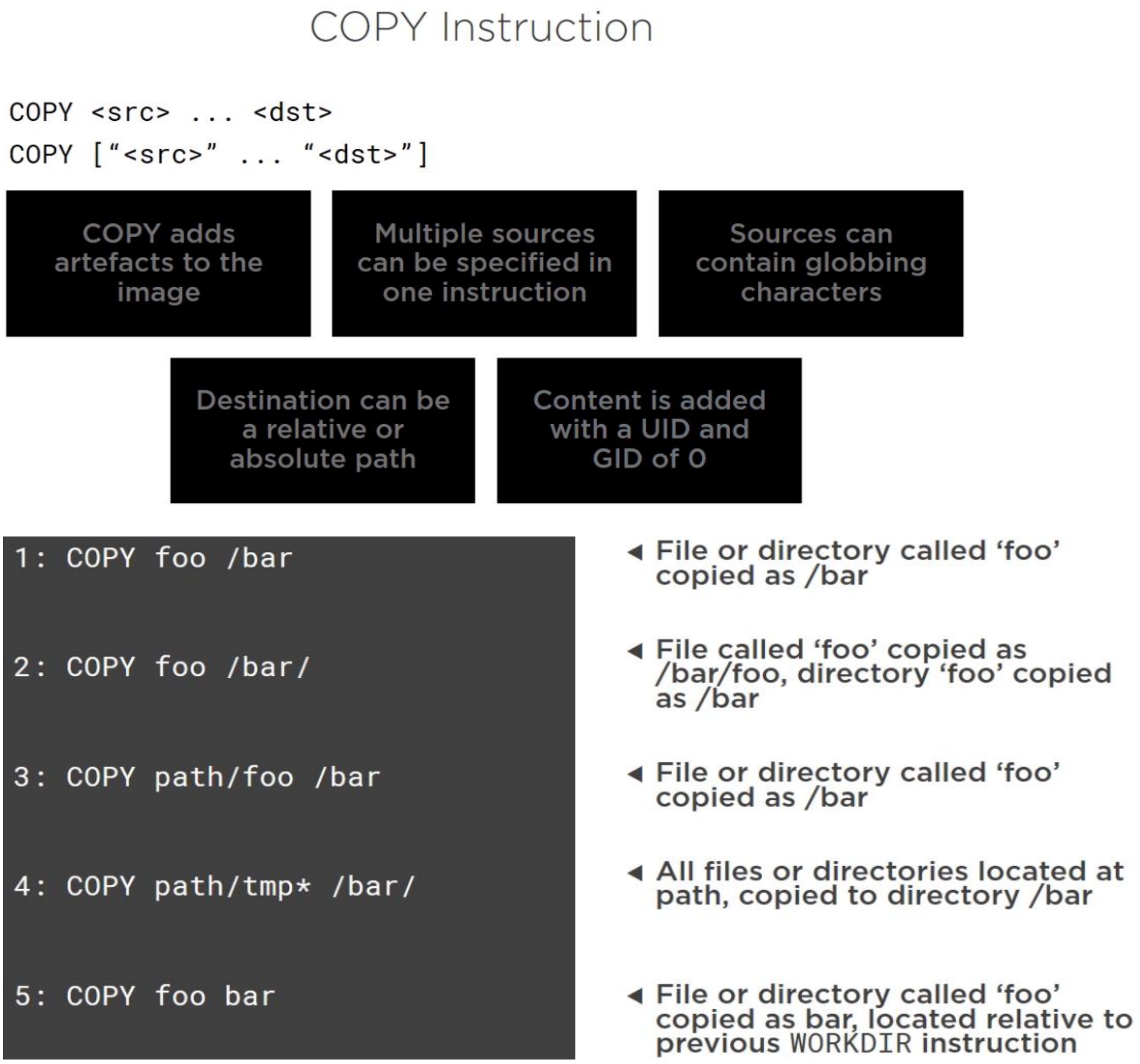
Two forms of syntax: shell and exec

Shell form executes command in shell

Exec form used when filesystem is devoid of shell

Build cache breaks only if instruction alters

Copy Instruction:



COPY Instruction

```
COPY <src> ... <dst>
COPY ["<src>" ... "<dst>"]
```

COPY adds artefacts to the image

Multiple sources can be specified in one instruction

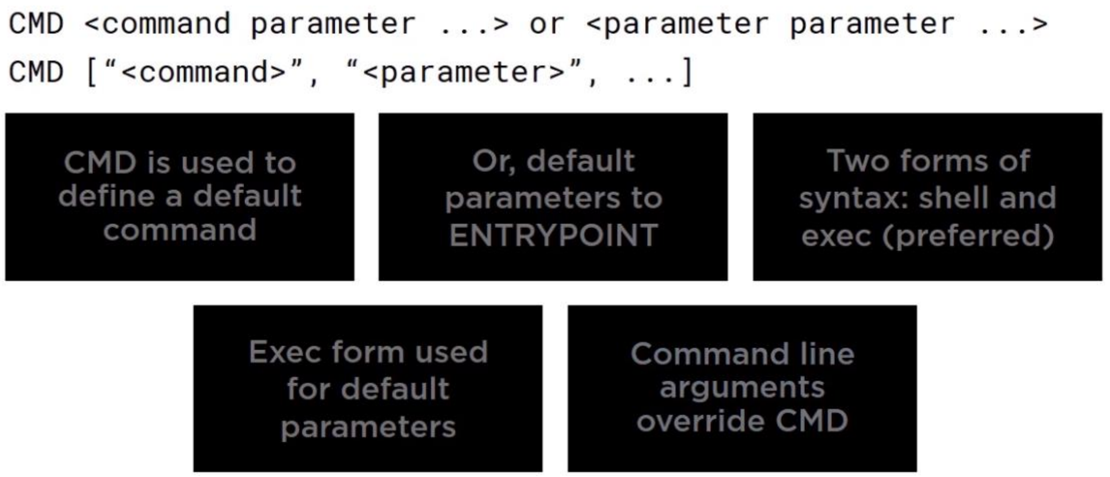
Sources can contain globbing characters

Destination can be a relative or absolute path

Content is added with a UID and GID of 0

`1: COPY foo /bar`	◄ File or directory called 'foo' copied as /bar
`2: COPY foo /bar/`	◄ File called 'foo' copied as /bar/foo, directory 'foo' copied as /bar
`3: COPY path/foo /bar`	◄ File or directory called 'foo' copied as /bar
`4: COPY path/tmp* /bar/`	◄ All files or directories located at path, copied to directory /bar
`5: COPY foo bar`	◄ File or directory called 'foo' copied as bar, located relative to previous `WORKDIR` instruction

CMD Instruction:



```
CMD <command parameter ...> or <parameter parameter ...>
CMD ["<command>", "<parameter>", ...]
```

CMD is used to define a default command

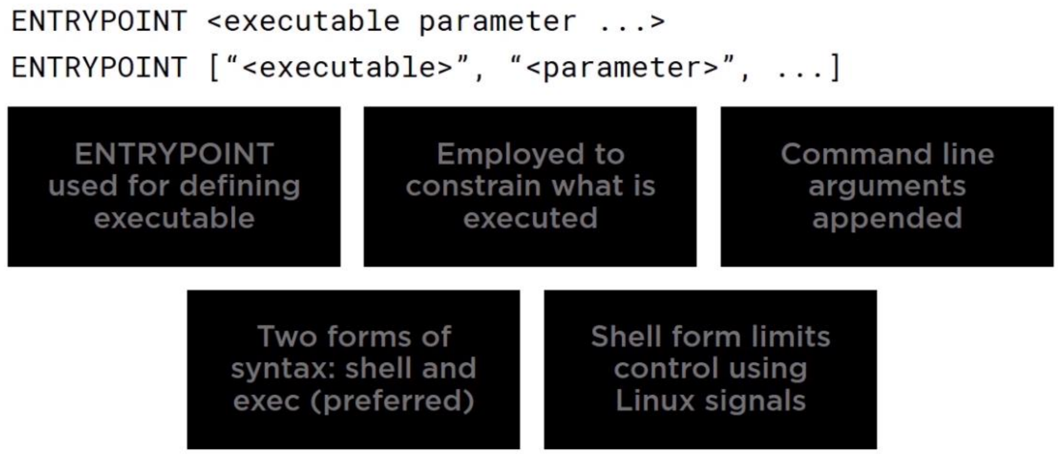
Or, default parameters to ENTRYPOINT

Two forms of syntax: shell and exec (preferred)

Exec form used for default parameters

Command line arguments override CMD

EXEC instruction:



```
ENTRYPOINT <executable parameter ...>
ENTRYPOINT ["<executable>", "<parameter>", ...]
```

ENTRYPOINT used for defining executable

Employed to constrain what is executed

Command line arguments appended

Two forms of syntax: shell and exec (preferred)

Shell form limits control using Linux signals

Below are the content of the Dockerfile

```
# ubuntubased container for a simple message.
# Each line starts with an instruction and its corresponding value.

FROM ubuntu:14.04
MAINTAINER ganesh@autofact.com
RUN apt-get update
# RUN apt-get install -y apache2
# RUN apt-get install -y ntp
CMD ["echo","Hello World"]

# RUN instructions are used to run commands against our images that
we are building.
# Every RUN instruction adds a new layer in the image.
```

Difference between COPY and ADD instruction.

COPY and ADD are both Dockerfile instructions that serve similar purposes. They let you copy files from a specific location into a Docker image.

COPY takes in a *src* and *destination*. It only lets you copy in a local file or directory from your host (the machine building the Docker image) into the Docker image itself.

ADD lets you do that too, but it also supports 2 other sources. First, you can use a URL instead of a local file / directory. Secondly, you can extract a tar file from the source directly into the destination.

In most cases if you're using a URL, you're downloading a zip file and are then using the RUN command to extract it. However, you might as well just use RUN with curl instead of ADD here so you chain everything into 1 RUN command to make a smaller Docker image.

If we want to add any file inside the image, the file has to be present inside the folder where the Dockerfile is located.
To add a file to the image use the `ADD` instruction.
Use below command to create the image using Dockerfile.

`$ docker build -t helloworld:0.1.0 .` – here 'helloworld' is the image name, '0.1.0' is the tag or version number and a '.' at the end is for specifying the Dockerfile is located in the same folder from where we are running the command.

The name of the image has to be in lowercase characters.

Once the image build is ready, we can use the image to run the container using the `docker run` command.

Each instruction is going to **add an image** layer to the image that we want to create.

One more Dockerfile

```
#running a webserver in container

FROM ubuntu
MAINTAINER ganesh@autofact.com
RUN apt-get update
RUN apt-get install -y apache2
RUN apt-get install -y apache2-utils
EXPOSE 80
CMD ["apachectl", "-D", "FOREGROUND"]
```

To create an image from the Dockerfile, use below command.

```
$ docker build -t="webserver" .   --- here webserver is the image
tag.
```

To test the container we can run the image created using the Dockerfile.

`$ docker run -d -p 80:80 <webserver>` .... This will run the container and start the Apache webserver on it.
How to reduce number of images in such case?

For this we can reduce the number of instructions. So the Dockerfile would look like as shown below,

```
FROM ubuntu
MAINTAINER ganesh@autofact.com
RUN apt-get update && apt-get install -y
     apache2 \
     apache2-utils \
     && apt-get clean \
     && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*
EXPOSE 80
CMD ["apache2ctl", "-D", "FOREGROUND"]
```

Build time instruction

Run time instruction

This will help reducing the number of layers as well as the size of image.

CMD instruction is a runtime instruction, whereas RUN is a build time instruction. There can only be one CMD instruction in one Dockerfile.

Build Cache:

When we create docker container using a Dockerfile, docker daemon saved the action in build cache and when we run the same instructions, it will use the information from cache and save the time to download or install everything again.

Try creating a docker container from a Dockerfile and rerun the `docker build` command again and note the difference.

Dockerfile and Layers:

Each instruction adds a layer to the docker image. In order to reduce number of layers in the image we can pack some of instructions in a single instruction.

Docker Networking

To understand how containers are created with a network IP address automatically, let's run the `ifconfig` on the host. Here we see that there's a `docker0` adaptor (n/w switch) running which manage the host to container network.

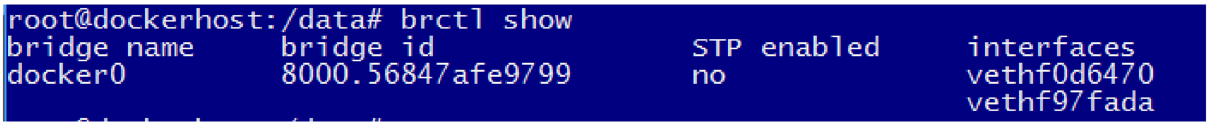
To get details about the interface / switch we can install the below package.

`$ apt-get install -y bridge-utils` or on Centos,

`$ yum install bridge-utils`

The 'bridge-utils' helps to view and manage software bridge on the host machine.

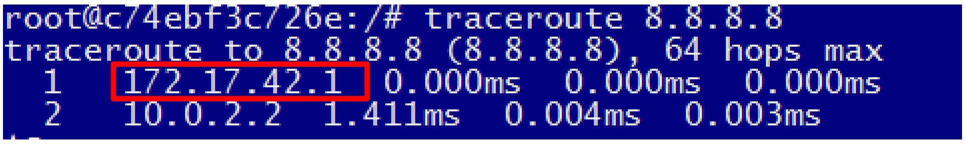
Use `$ brctl show` command to see the components that the bridge is managing.



```
root@dockerhost:/data# brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.56847afe9799       no              vethf0d6470
                                                        vethf97fada
```

Here we have started two container and both are seen in the network bridge utility.

If we run `traceroute` command inside the container we can see the gateway used which in the dockerhost machine.



```
root@c74ebf3c726e:/# traceroute 8.8.8.8
traceroute to 8.8.8.8 (8.8.8.8), 64 hops max
  1   172.17.42.1  0.000ms  0.000ms  0.000ms
  2   10.0.2.2  1.411ms  0.004ms  0.003ms
```

The network settings related file like, `resolve.conf` or `hosts` file for a container are available in the `/var/lib/docker/container/<container-id>` directory. We can make changes to the contents of these files in order to make the container alter network settings.

We can also expose network port on the container in the Dockerfile as we have earlier. Also at the run time we can map network port of host to a port on container by using below command.

```
$ docker run -it -p 5001:80 --name=webcontainer apache
```

…By this we can map port on host to a port on container.

Or we can also use the IP address along with the port number as shown below,

```
$ docker run -it -p 192.168.33.35:5002 :80 -name=web1 apache
```

To view which ports are mapped on a container, we can use the command,

```
$ docker port <container-id>
```

In the Dockerfile we can expose multiple ports as show below,

```
EXPOSE 80 1001 1002 1004 1005
```

Build a new image from the Docker file

```
$ docker build -t=”sample” .
```

Use the image to create container,

```
$ docker run -d -P -name=samplecont sample
```

Now run command to view how ports are mapped to the host.

```
$ docker port sample.
```

All exposed ports are mapped randomly to ports on docker host

In order to assign a specific range of IP addresses for the containers, we can use below commands. First stop the docker daemon,

```
$ service docker stop
$ ip link del docker0
```

… remove the docker bridge

Edit the docker config file at `/etc/default/docker` and update the contents with,

```
DOCKER_OPTS=--bip=10.2.15.1/24
```

– this will assign the bridge IP address.

Restart the docker service. And see the docker bridge ip. This way we can assign a specific range of IP addresses to the docker bridge which inturn gets assigned to the container started .

Docker Firewall:

If we look at the `iptables -L -v` we see that there are defaults firewall setting about the docker container communication.

By default the --`icc` value is set to `true` which mean that all communications are allowed across all containers.

By setting the value to `false` we can disallow inter-container-communication to stop.

Also, by setting the `--iptables` value to false, will disallow docker to interfere with `iptables` rules.

Update the file located at `/etc/default/docker` (docker conf file) with below line. `DOCKER_OPTS="—icc=true -iptables=false"` …. The `iptables` rule overrides the `icc` settings.

Linking containers:

We can link two containers using few commands. Here we will create a container with a name as shown below,

```
$ docker run -d --name=src <image-name>
```

`$ docker run --name=rcvr -link-src:src` .. here `link-src` is the container name to which we want to link our `'rcvr'` container. We also have to provide alias name for the `src` container.

Check the hosts file on the container and you will notice that the ipaddress of the src container gets added as an entry to the `src` alias name.

Log Maintenance:

Logs for docker daemon running on the docker host can be view as mentioned below. First stop the docker service and then run below command.

```
$ docker -d -l debug  &
```

The docker file in the default directory in /etc can be updated to run the docker in required log level as show below.

```
DOCKER_OPTS="--log-level=fatal"
```

Some best practices in writing the Dockerfile.

- One would usually want to test each step inside a container before writing a Dockerfile that can be then used enterprise wide.

Docker Events:

Get real time events from the server.

Only the last 1000 log events are returned. You can use filters to further limit the number of events returned.

Docker containers report the following events:

- `attach`
- `commit`
- `copy`
- `create`
- `destroy`
- `detach`
- `die`
- `exec_create`
- `exec_detach`
- `exec_die`
- `exec_start`
- `export`
- `health_status`
- `kill`
- `oom`
- `pause`
- `rename`
- `resize`
- `restart`
- `start`
- `stop`
- `top`
- `unpause`
- `update`

Docker images report the following events:

- `delete`
- `import`
- `load`
- `pull`
- `push`
- `save`
- `tag`
- `untag`

Docker volumes report the following events:

- `create`
- `destroy`
- `mount`
- `unmount`

Docker networks report the following events:

- `create`
- `connect`
- `destroy`
- `disconnect`
- `remove`