Docker is an open-source platform that allows developers to create, deploy, and run applications in containers. Containers are lightweight, standalone, and executable packages that contain everything needed to run an application, including code, libraries, and system tools.

There are several reasons why Docker is useful for software testing:

1. Consistent Environments: Docker provides a way to create and run applications in a consistent environment, which means that you can test your application in the same environment as the one it will be deployed in.

2. Isolation: Docker containers are isolated from each other and from the host system, which means that you can test your application without worrying about interfering with other applications or the host system.
3. Reproducibility: Docker allows you to create reproducible environments, which means that you can easily recreate a specific environment to reproduce a bug or to test a new feature.
4. Scalability: Docker makes it easy to scale your testing infrastructure up or down, depending on your needs. You can easily spin up new containers to test different parts of your application, or to test your application on different platforms.
5. Cost savings: Docker allows you to test your application on a smaller infrastructure footprint than traditional virtualization, which can save costs on hardware, maintenance, and energy.

Overall, Docker makes it easy to test your application in a controlled, consistent, and reproducible environment, which can help you find and fix bugs faster and ensure that your application works as intended in production.

1. docker --**version** -> To check Docker Version.

2. docker **pull** <Image-Name> -> To download Docker Image from Docker Hub.Pull an image or a repository from a registry

3. docker **ps** -> To check how many Containers are Up and Running.

4. docker **ps –a** -> To check how many Containers are Available, Up and Running.

5. docker **run -it -d** <Image-Name> -> To Create Docker Container from Docker Image.

6. docker **start** <Container-Id> -> To Start Container.

7. docker **restart** <Container-Id> -> To Restart Container.

8. docker **stop** <Container-Id> -> To Stop Container when in Running.

9. docker **rm** <Container-Id> -> To Delete Docker Container.

10. What are the commands for Dockerfile?

Ans: Dockerfile is a very important section, check the link **HERE** for all commands related to Dockerfile.

10. docker **rmi** <Image-Id> -> To Delete Docker Image.

11. docker **images** -> To check for Available Docker Images in System.

12. docker **exec –it <Container Id> bash** -> To Get into Container and take Control on it.

13. exit -> To Come Out from Container to Docker.

14. docker **kill** <Container Id> -> To Stop Container Forcefully.

15. docker **inspect** <Container Id> -> It will give complete information about Container.

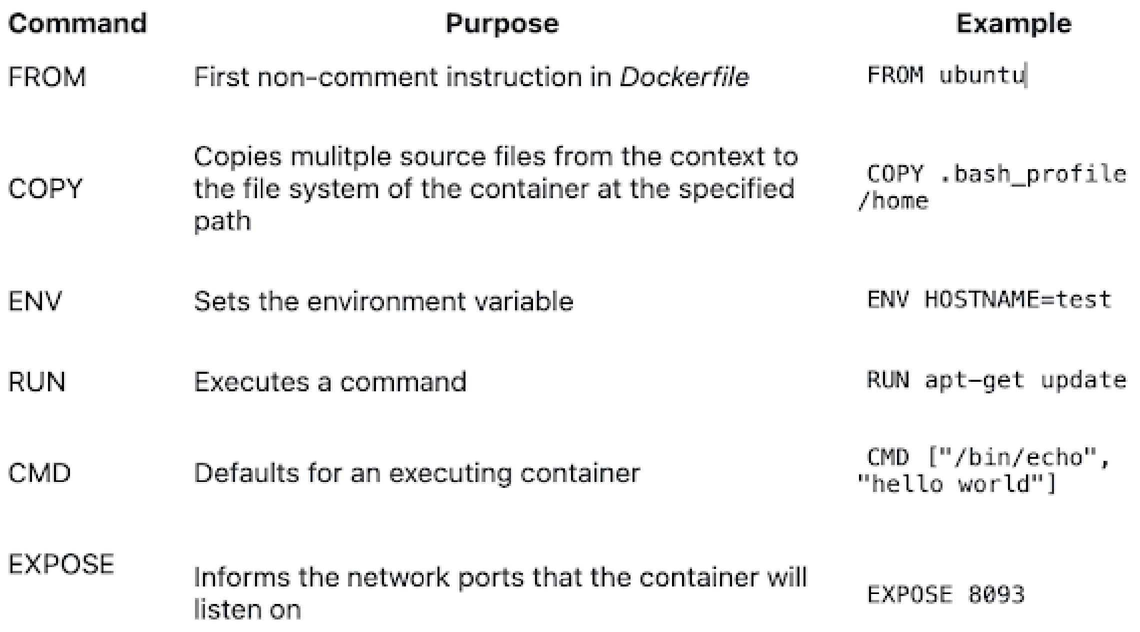
16. docker **image prune –a** -> It will delete images which doesn't have even a Single Container.

17. docker **run --rm** <Image-Name> -> Create a docker container and auto remove on exit

18. How to setup docker with selenium grid for cross browser testing?

Ans: CLICK HERE FOR DETAIL CODE

19. docker **system prune -a** : it will delete all the images, containers and networks which are not used to be active anymore.

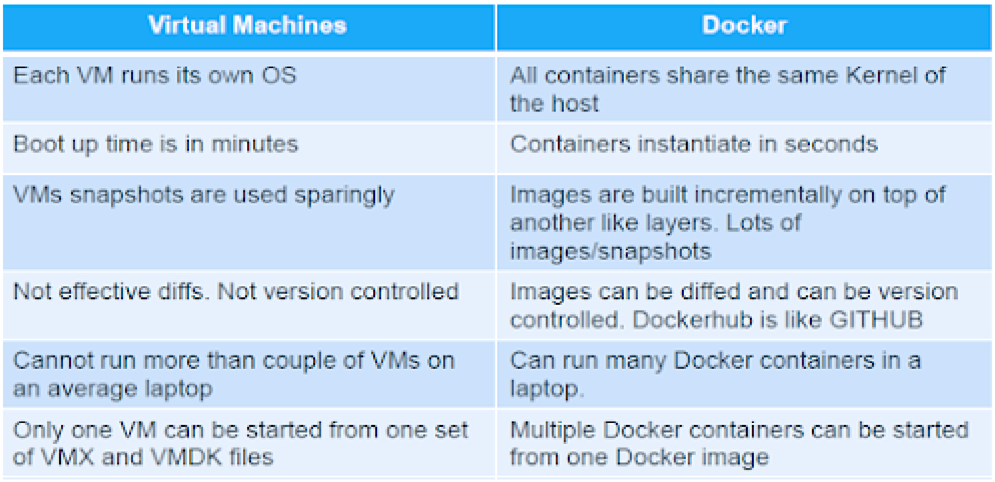


Command	Purpose	Example
FROM	First non-comment instruction in *Dockerfile*	`FROM ubuntu`
COPY	Copies mulitple source files from the context to the file system of the container at the specified path	`COPY .bash_profile /home`
ENV	Sets the environment variable	`ENV HOSTNAME=test`
RUN	Executes a command	`RUN apt-get update`
CMD	Defaults for an executing container	`CMD ["/bin/echo", "hello world"]`
EXPOSE	Informs the network ports that the container will listen on	`EXPOSE 8093`

What is the difference between VM and Doker?

Virtual machines have a full OS with its own memory management installed with the associated overhead of virtual device drivers. In a virtual machine, valuable resources are emulated for the guest OS and hypervisor, which makes it possible to run many instances of one or more operating systems in parallel on a single machine (or host). Every guest OS runs as an individual entity from the host system. Hence, we can look at it an independent full-fledge house where we don't share any resources

In the other hand, Docker containers are executed with the Docker engine rather than the hypervisor. Containers are therefore smaller than Virtual Machines and enable faster start up with better performance, less isolation and greater compatibility possible due to sharing of the host's kernel. Hence, it looks very similar to residental flats system where we share resources of the building.



Virtual Machines	Docker
Each VM runs its own OS	All containers share the same Kernel of the host
Boot up time is in minutes	Containers instantiate in seconds
VMs snapshots are used sparingly	Images are built incrementally on top of another like layers. Lots of images/snapshots
Not effective diffs. Not version controlled	Images can be diffed and can be version controlled. Dockerhub is like GITHUB
Cannot run more than couple of VMs on an average laptop	Can run many Docker containers in a laptop.
Only one VM can be started from one set of VMX and VMDK files	Multiple Docker containers can be started from one Docker image

How is Dockerfile different from Docker Compose?

A Dockerfile is a simple text file that contains the commands a user could call to assemble an image whereas Docker Compose is a tool for defining and running multi-container Docker applications.

Docker Compose define the services that make up your app in docker-compose.yml so they can be run together in an isolated environment. It get an app running in one command by just running docker-compose up.

Docker compose uses the Dockerfile if one add the build command to your project's docker-compose.yml. Your Docker workflow should be to build a suitable Dockerfile for each image you wish to create, then use compose to assemble the images using the build command.

What is the maximum number of containers you can run per host?

This really depends on your environment. The size of your applications as well as the amount of available resources (i.e like CPU) will all affect the number of containers that can be run in your environment. Containers unfortunately are not magical.

They can’t create new CPU from scratch. They do, however, provide a more efficient way of utilizing your resources. The containers themselves are super lightweight (remember, shared OS vs individual OS per container) and only last as long as the process they are running.

What is Docker Swarm?

Docker Swarm is native clustering for Docker. It turns a pool of Docker hosts into a single, virtual Docker host.

Docker Swarm serves the standard Docker API, any tool that already communicates with a Docker daemon can use Swarm to transparently scale to multiple hosts.

Does Docker Swarm do load balancing?

Yes, Docker Swarm does load balancing. Docker Swarm's load balancer runs on every node and is capable of balancing load requests across multiple containers and hosts.

1. Difference between CMD and ENTRYPOINT

TL;DR `CMD` will work for most of the cases.

Default entry point for a container is `/bin/sh`, the default shell.

Running a container as `docker container run -it ubuntu` uses that command and starts the default shell. The output is shown as:

```
> docker container run -it ubuntu

root@88976ddee107:/#
```

`ENTRYPOINT` allows to override the entry point to some other command, and even customize it. For example, a container can be started as:

```
> docker container run -it --entrypoint=/bin/cat ubuntu /etc/passwd

root:x:0:0:root:/root:/bin/bash

daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin

bin:x:2:2:bin:/bin:/usr/sbin/nologin

sys:x:3:3:sys:/dev:/usr/sbin/nologin

. . .
```

This command overrides the entry point to the container to `/bin/cat`. The argument(s) passed to the CLI are used by the entry point.

3. Difference between ADD and COPY

TL;DR `COPY` will work for most of the cases.

`ADD` has all capabilities of `COPY` and has the following additional features:

1. Allows tar file auto-extraction in the image, for example, `ADD app.tar.gz /opt/var/myapp`.
2. Allows files to be downloaded from a remote URL. However, the downloaded files will become part of the image. This causes the image size to bloat. So its recommended to use `curl` or `wget` to download the archive explicitly, extract, and remove the archive.

4. Import and export images

Docker images can be saved using `image save` command to a `.tar` file:

```
docker image save helloworld > helloworld.tar
```

These tar files can then be imported using `load` command:

```
docker image load -i helloworld.tar
```

`What is Dockerfile? Dockerfile`

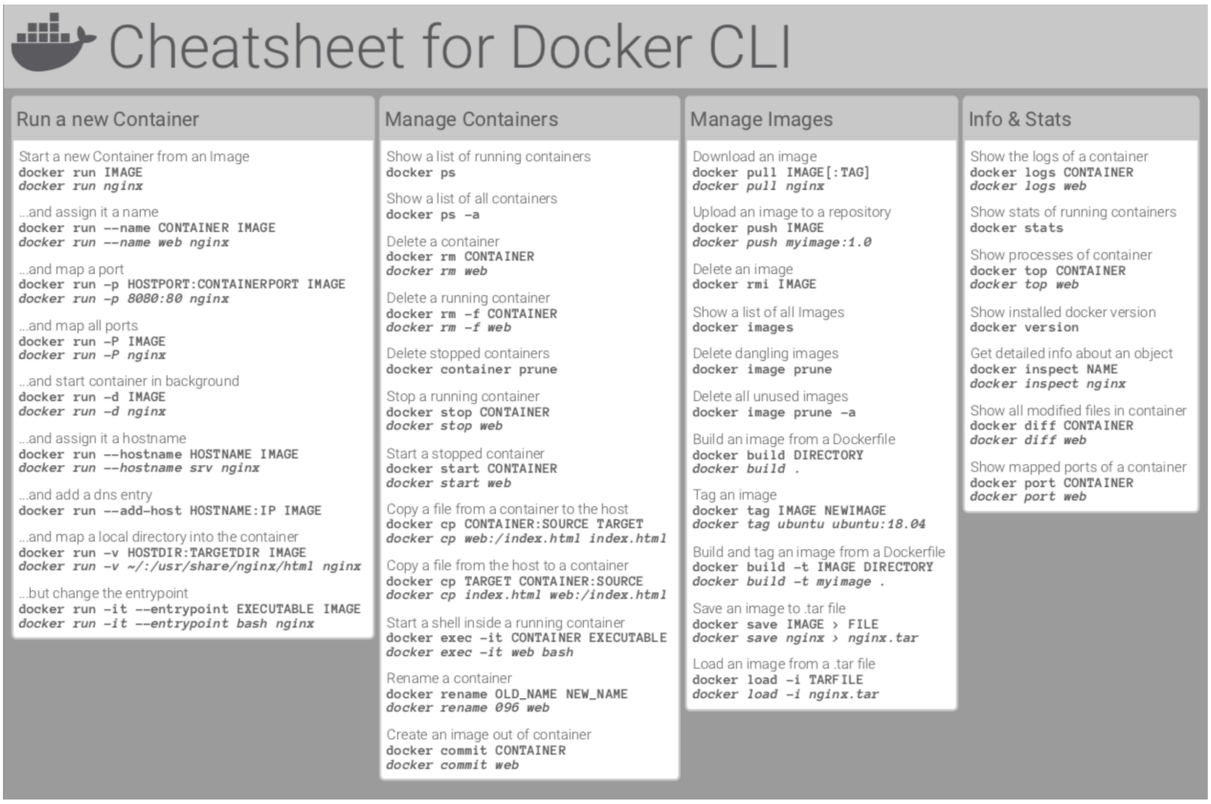
Docker builds images by reading instructions from a *Dockerfile*. A *Dockerfile* is a text document that contains all the commands a user could call on the command line to assemble an image. `docker image build` command uses this file and executes all the commands in succession to create an image.

`build` command is also passed a context that is used during image creation. This context can be a path on your local filesystem or a URL to a Git repository.

NEWLY ADDED

Basic Docker CLIs

Here’s the list of the basic Docker commands that works on both Docker Desktop as well as Docker Engine:



Cheatsheet for Docker CLI

Run a new Container

Start a new Container from an Image

```
docker run IMAGE
docker run nginx
```

...and assign it a name

```
docker run --name CONTAINER IMAGE
docker run --name web nginx
```

...and map a port

```
docker run -p HOSTPORT:CONTAINERPORT IMAGE
docker run -p 8080:80 nginx
```

...and map all ports

```
docker run -P IMAGE
docker run -P nginx
```

...and start container in background

```
docker run -d IMAGE
docker run -d nginx
```

...and assign it a hostname

```
docker run --hostname HOSTNAME IMAGE
docker run --hostname srv nginx
```

...and add a dns entry

```
docker run --add-host HOSTNAME:IP IMAGE
```

...and map a local directory into the container

```
docker run -v HOSTDIR:TARGETDIR IMAGE
docker run -v ~/:/usr/share/nginx/html nginx
```

...but change the entrypoint

```
docker run -it --entrypoint EXECUTABLE IMAGE
docker run -it --entrypoint bash nginx
```

Manage Containers

Show a list of running containers

```
docker ps
```

Show a list of all containers

```
docker ps -a
```

Delete a container

```
docker rm CONTAINER
docker rm web
```

Delete a running container

```
docker rm -f CONTAINER
docker rm -f web
```

Delete stopped containers

```
docker container prune
```

Stop a running container

```
docker stop CONTAINER
docker stop web
```

Start a stopped container

```
docker start CONTAINER
docker start web
```

Copy a file from a container to the host

```
docker cp CONTAINER:SOURCE TARGET
docker cp web:/index.html index.html
```

Copy a file from the host to a container

```
docker cp TARGET CONTAINER:SOURCE
docker cp index.html web:/index.html
```

Start a shell inside a running container

```
docker exec -it CONTAINER EXECUTABLE
docker exec -it web bash
```

Rename a container

```
docker rename OLD_NAME NEW_NAME
docker rename 096 web
```

Create an image out of container

```
docker commit CONTAINER
docker commit web
```

Manage Images

Download an image

```
docker pull IMAGE[:TAG]
docker pull nginx
```

Upload an image to a repository

```
docker push IMAGE
docker push myimage:1.0
```

Delete an image

```
docker rmi IMAGE
```

Show a list of all Images

```
docker images
```

Delete dangling images

```
docker image prune
```

Delete all unused images

```
docker image prune -a
```

Build an image from a Dockerfile

```
docker build DIRECTORY
docker build .
```

Tag an image

```
docker tag IMAGE NEWIMAGE
docker tag ubuntu ubuntu:18.04
```

Build and tag an image from a Dockerfile

```
docker build -t IMAGE DIRECTORY
docker build -t myimage .
```

Save an image to .tar file

```
docker save IMAGE > FILE
docker save nginx > nginx.tar
```

Load an image from a .tar file

```
docker load -i TARFILE
docker load -i nginx.tar
```

Info & Stats

Show the logs of a container

```
docker logs CONTAINER
docker logs web
```

Show stats of running containers

```
docker stats
```

Show processes of container

```
docker top CONTAINER
docker top web
```

Show installed docker version

```
docker version
```

Get detailed info about an object

```
docker inspect NAME
docker inspect nginx
```

Show all modified files in container

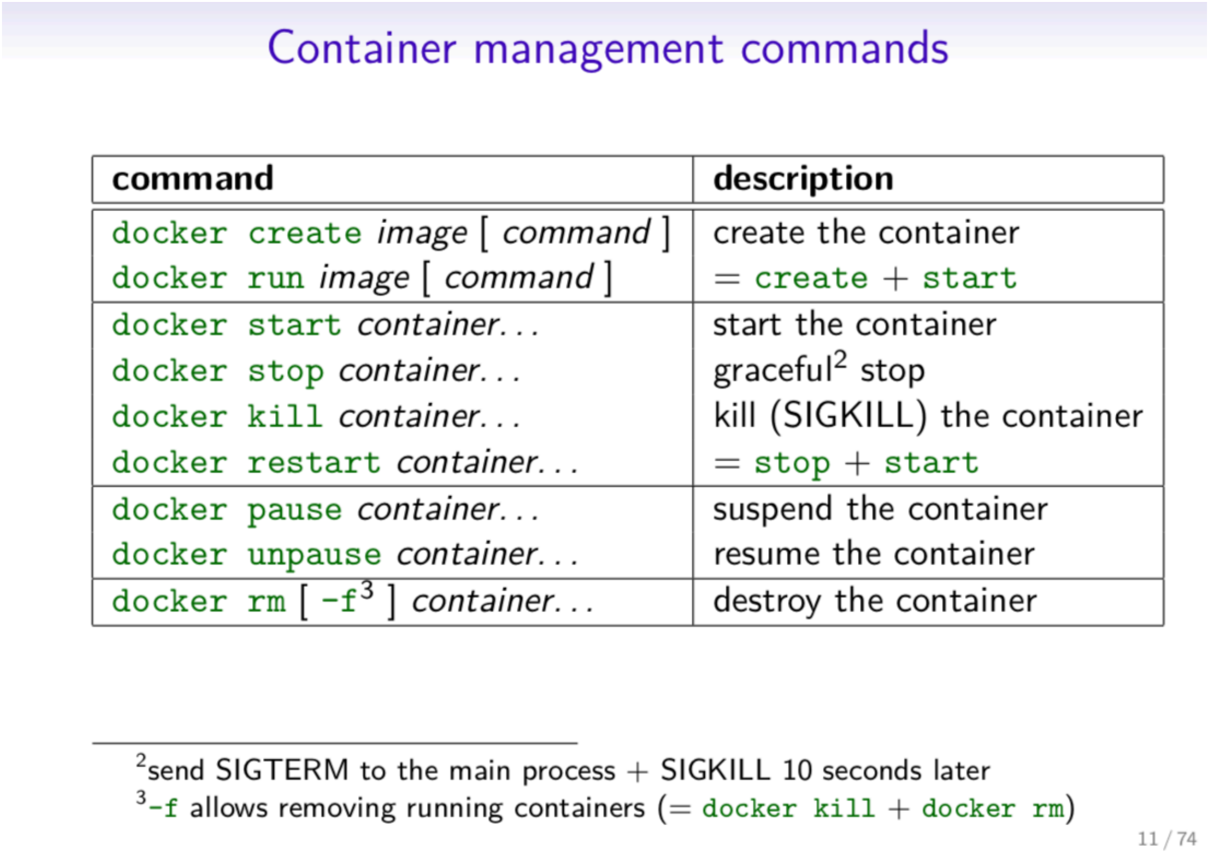
```
docker diff CONTAINER
docker diff web
```

Show mapped ports of a container

```
docker port CONTAINER
docker port web
```

Container Management CLIs

Here’s the list of the Docker commands that manages Docker images and containers flawlessly:



Container management commands

command	description
`docker create` *image* [*command*]	create the container
`docker run` *image* [*command*]	= `create` + `start`
`docker start` *container*...	start the container
`docker stop` *container*...	graceful[2] stop
`docker kill` *container*...	kill (SIGKILL) the container
`docker restart` *container*...	= `stop` + `start`
`docker pause` *container*...	suspend the container
`docker unpause` *container*...	resume the container
`docker rm` [`-f`[3]] *container*...	destroy the container

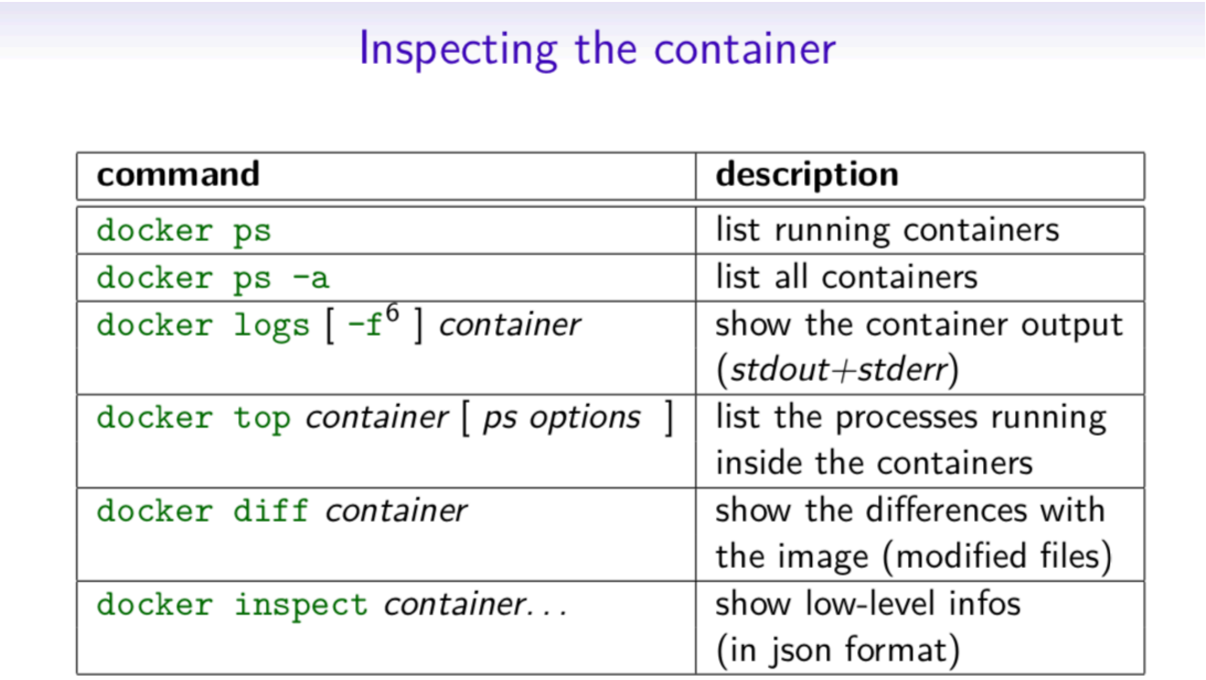
[2] send SIGTERM to the main process + SIGKILL 10 seconds later

[3] `-f` allows removing running containers (= `docker kill` + `docker rm`)

11 / 74

Inspecting The Container

Here’s the list of the basic Docker commands that helps you inspect the containers seamlessly:

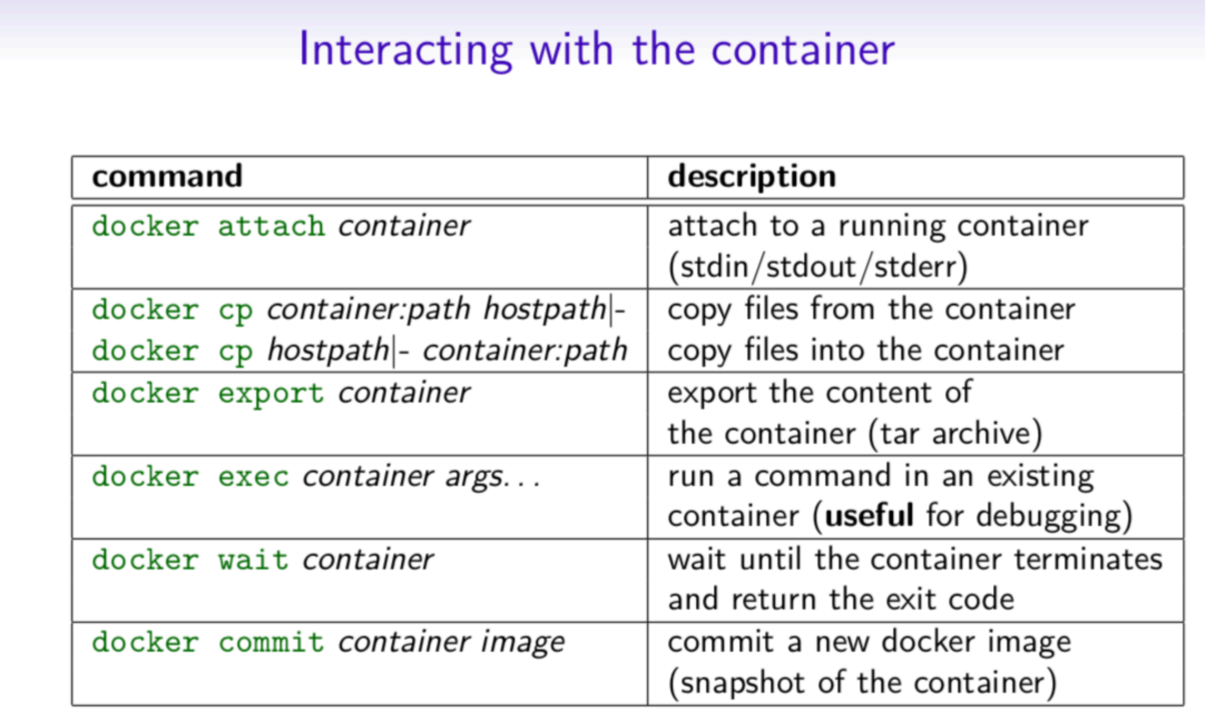


Inspecting the container

command	description
`docker ps`	list running containers
`docker ps -a`	list all containers
`docker logs` [`-f`[6]] *container*	show the container output (*stdout*+*stderr*)
`docker top` *container* [*ps options*]	list the processes running inside the containers
`docker diff` *container*	show the differences with the image (modified files)
`docker inspect` *container*...	show low-level infos (in json format)

Interacting with Container

Do you want to know how to access the containers? Check out these fundamental commands:



Interacting with the container

command	description
`docker attach` *container*	attach to a running container (stdin/stdout/stderr)
`docker cp` *container:path hostpath*|- `docker cp` *hostpath*|- *container:path*	copy files from the container copy files into the container
`docker export` *container*	export the content of the container (tar archive)
`docker exec` *container args...*	run a command in an existing container (**useful** for debugging)
`docker wait` *container*	wait until the container terminates and return the exit code
`docker commit` *container image*	commit a new docker image (snapshot of the container)

Image Management Commands

Here’s the list of Docker commands that helps you manage the Docker Images:

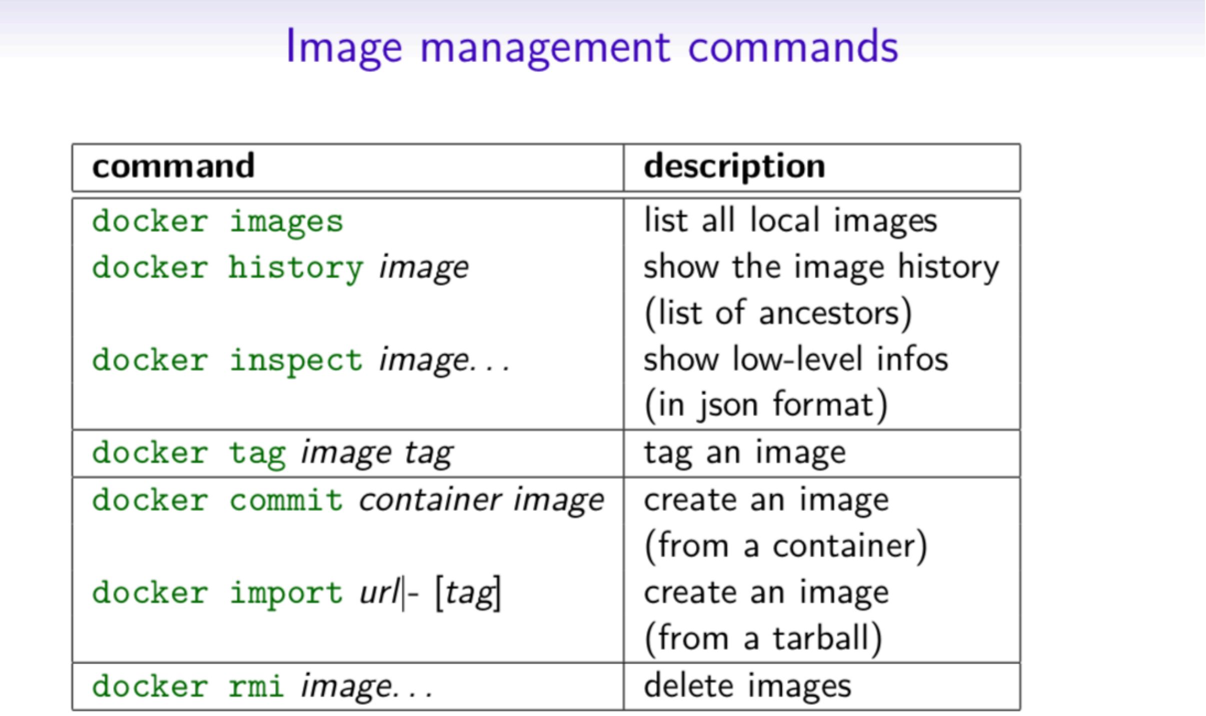


Image management commands

command	description
`docker images`	list all local images
`docker history` *image*	show the image history (list of ancestors)
`docker inspect` *image...*	show low-level infos (in json format)
`docker tag` *image tag*	tag an image
`docker commit` *container image*	create an image (from a container)
`docker import` *url*|- [*tag*]	create an image (from a tarball)
`docker rmi` *image...*	delete images

Image Transfer Commands

Here's the list of Docker image transfer commands:

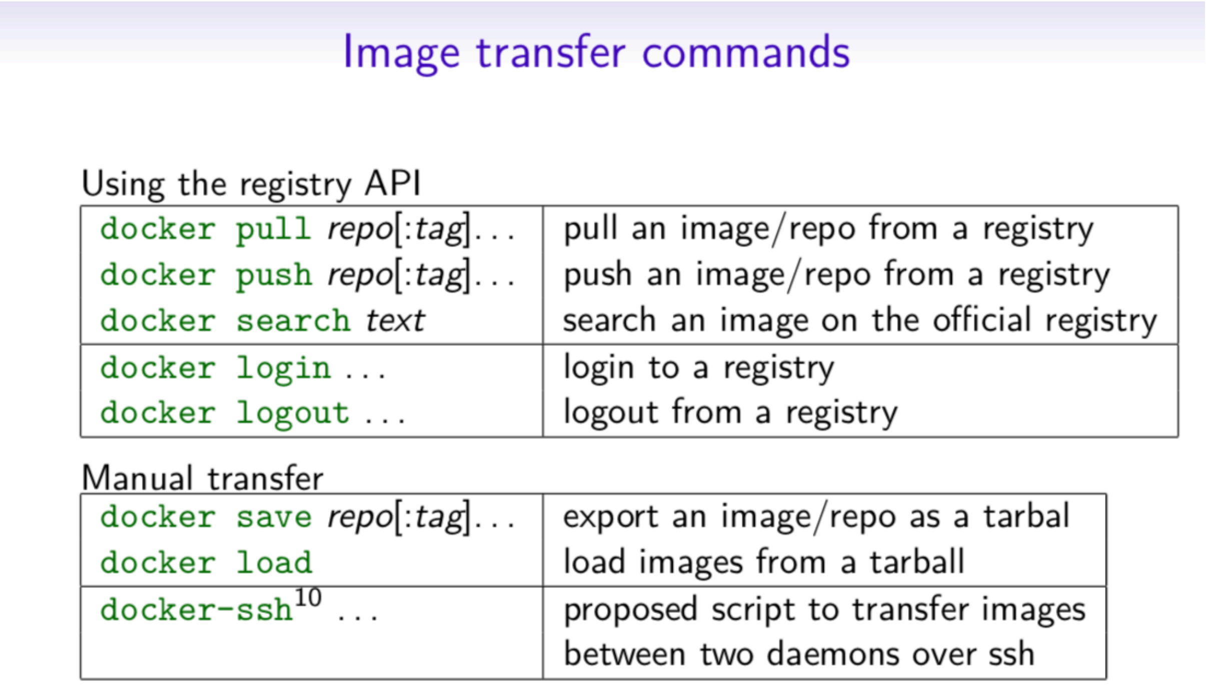


Image transfer commands

Using the registry API

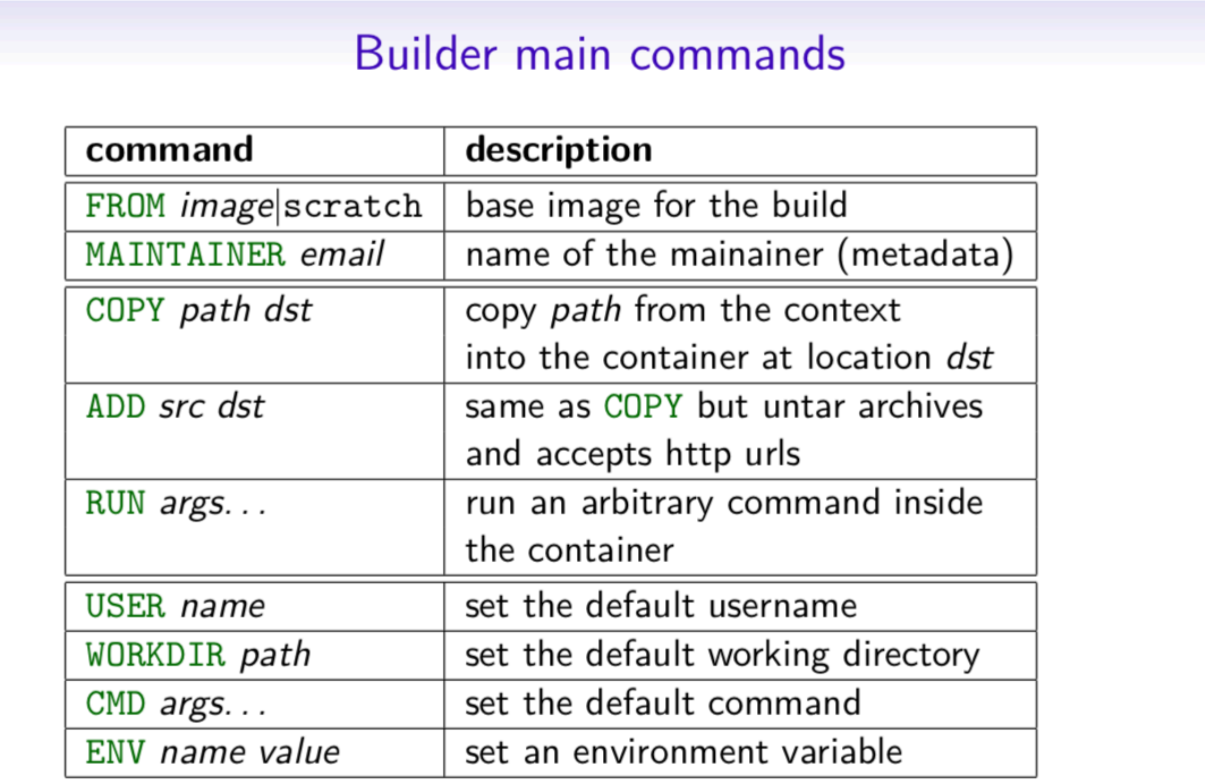
`docker pull` *repo*[:*tag*]...	pull an image/repo from a registry
`docker push` *repo*[:*tag*]...	push an image/repo from a registry
`docker search` *text*	search an image on the official registry
`docker login` ...	login to a registry
`docker logout` ...	logout from a registry

Manual transfer

`docker save` *repo*[:*tag*]...	export an image/repo as a tarbal
`docker load`	load images from a tarball
`docker-ssh`[10] ...	proposed script to transfer images between two daemons over ssh

Builder Main Commands

Want to know how to build Docker Image? Do check out the list of Image Build Commands:



Builder main commands

command	description
FROM *image*|scratch	base image for the build
MAINTAINER *email*	name of the mainainer (metadata)
COPY *path dst*	copy *path* from the context into the container at location *dst*
ADD *src dst*	same as COPY but untar archives and accepts http urls
RUN *args...*	run an arbitrary command inside the container
USER *name*	set the default username
WORKDIR *path*	set the default working directory
CMD *args...*	set the default command
ENV *name value*	set an environment variable

The Docker CLI

Manage images

`docker build`

```
docker build [options] .
  -t "app/container_name"    # name
```

Create an `image` from a Dockerfile.

`docker run`

```
docker run [options] IMAGE
  # see `docker create` for options
```

Run a command in an `image`.

Manage containers

`docker create`

```
docker create [options] IMAGE
  -a, --attach               # attach stdout/err
  -i, --interactive          # attach stdin (interactive)
  -t, --tty                  # pseudo-tty
      --name NAME            # name your image
  -p, --publish 5000:5000    # port map
      --expose 5432          # expose a port to linked
containers
  -P, --publish-all          # publish all ports
      --link container:alias # linking
  -v, --volume `pwd`:/app    # mount (absolute paths needed)
  -e, --env NAME=hello       # env vars
```

Example

```
$ docker create --name app_redis_1 \
  --expose 6379 \
  redis:3.0.2
```

Create a `container` from an `image`.

`docker exec`

```
docker exec [options] CONTAINER COMMAND
  -d, --detach        # run in background
  -i, --interactive   # stdin
  -t, --tty           # interactive
```

Example

```
$ docker exec app_web_1 tail logs/development.log
$ docker exec -t -i app_web_1 rails c
```

Run commands in a `container`.

`docker start`

```
docker start [options] CONTAINER
  -a, --attach        # attach stdout/err
  -i, --interactive   # attach stdin

docker stop [options] CONTAINER
```

Start/stop a `container`.

`docker ps`

```
$ docker ps
$ docker ps -a
$ docker kill $ID
```

Manage `container`s using ps/kill.

Images

`docker images`

```
$ docker images
  REPOSITORY   TAG        ID
  ubuntu       12.10      b750fe78269d
  me/myapp     latest     7b2431a8d968
```

```
$ docker images -a   # also show intermediate
```

Manages `images`.

`docker rmi`

```
docker rmi b750fe78269d
```

Deletes `images`.

Also see

- Getting Started *(docker.io)*

Dockerfile

Inheritance

```
FROM ruby:2.2.2
```

Variables

```
ENV APP_HOME /myapp
RUN mkdir $APP_HOME
```

Initialization

```
RUN bundle install
```

```
WORKDIR /myapp
```

```
VOLUME ["/data"]
# Specification for mount point
```

```
ADD file.xyz /file.xyz
COPY --chown=user:group host_file.xyz /path/container_file.xyz
```

Onbuild

```
ONBUILD RUN bundle install
# when used with another file
```

Commands

```
EXPOSE 5900
CMD    ["bundle", "exec", "rails", "server"]
```

Entrypoint

```
ENTRYPOINT ["executable", "param1", "param2"]
ENTRYPOINT command param1 param2
```

Configures a container that will run as an executable.

```
ENTRYPOINT exec top -b
```

This will use shell processing to substitute shell variables, and will ignore any `CMD` or `docker run` command line arguments.

Metadata

```
LABEL version="1.0"
```

```
LABEL "com.example.vendor"="ACME Incorporated"
LABEL com.example.label-with-value="foo"
```

```
LABEL description="This text illustrates \
that label-values can span multiple lines."
```

See also

- https://docs.docker.com/engine/reference/builder/

docker-compose

Basic example

```
# docker-compose.yml
version: '2'

services:
  web:
    build: .
    # build from Dockerfile
    context: ./Path
    dockerfile: Dockerfile
    ports:
     - "5000:5000"
    volumes:
     - .:/code
  redis:
    image: redis
```

Commands

```
docker-compose start
docker-compose stop

docker-compose pause
docker-compose unpause

docker-compose ps
docker-compose up
docker-compose down
```

Reference

Building

```
web:
  # build from Dockerfile
  build: .

 # build from custom Dockerfile
  build:
    context: ./dir
    dockerfile: Dockerfile.dev

 # build from image
  image: ubuntu
  image: ubuntu:14.04
  image: tutum/influxdb
  image: example-registry:4000/postgresql
  image: a4bc65fd

```

Ports

```
 ports:
    - "3000"
    - "8000:80"  # guest:host

 # expose ports to linked services (not to host)
  expose: ["3000"]

```

Commands

```
 # command to execute
  command: bundle exec thin -p 3000
  command: [bundle, exec, thin, -p, 3000]

 # override the entrypoint
  entrypoint: /app/start.sh
  entrypoint: [php, -d, vendor/bin/phpunit]

```

Environment variables

```
 # environment vars
  environment:
    RACK_ENV: development
  environment:
    - RACK_ENV=development

 # environment vars from file
  env_file: .env
  env_file: [.env, .development.env]

```

Dependencies

```
 # makes the `db` service available as the hostname `database`
  # (implies depends_on)
  links:
    - db:database
    - redis

# make sure `db` is alive before starting
  depends_on:
    - db

```

Other options

```
# make this service extend another
  extends:
    file: common.yml  # optional
    service: webapp

volumes:
    - /var/lib/mysql
    - ./_data:/var/lib/mysql

```

Advanced features

Labels

```
services:
  web:
    labels:
      com.example.description: "Accounting web app"

```

DNS servers

```
services:
  web:
    dns: 8.8.8.8
    dns:
      - 8.8.8.8
      - 8.8.4.4

```

Devices

```
services:
  web:
    devices:
    - "/dev/ttyUSB0:/dev/ttyUSB0"
```

External links

```
services:
  web:
    external_links:
      - redis_1
      - project_db_1:mysql
```

Hosts

```
services:
  web:
    extra_hosts:
      - "somehost:192.168.1.100"
```

services

To view list of all the services running in swarm

```
docker service ls
```

To see all running services

```
docker stack services stack_name
```

to see all services logs

```
docker service logs stack_name service_name
```

To scale services quickly across qualified node

```
docker service scale stack_name_service_name=replicas
```

clean up

To clean or prune unused (dangling) images

```
docker image prune
```

To remove all images which are not in use containers , add - a

```
docker image prune -a
```

To prune your entire system

```
docker system prune
```

To leave swarm

```
docker swarm leave
```

To remove swarm (deletes all volume data and database info)

```
docker stack rm stack_name
```

To kill all running containers

```
docker kill $(docker ps -q )
```

Docker Security

Docker Scout

Command line tool for Docker Scout:

```
docker scout
```

Analyzes a software artifact for vulnerabilities

```
docker scout cves [OPTIONS] IMAGE|DIRECTORY|ARCHIVE
```

Display vulnerabilities from a docker save tarball

```
docker save redis > redis.tar
```

Display vulnerabilities from an OCI directory

```
skopeo copy --override-os linux docker://alpine oci:redis
```

Export vulnerabilities to a SARIF JSON file

```
docker scout cves --format sarif --output redis.sarif.json redis
```

Comparing two images

```
docker scout compare --to redis:6.0 redis:6-bullseye
```

Displaying the Quick Overview of an Image

```
docker scout quickview redis:6.0
```