Docker Cheat Sheet

NOTE: This used to be a gist that continually expanded. It's now a GitHub project because it's considerably easier for other people to edit, fix and expand on Docker using Github. Just click README.md, and then on the "writing pen" icon on the right to edit.

Why

"With Docker, developers can build any app in any language using any toolchain. "Dockerized" apps are completely portable and can run anywhere - colleagues' OS X and Windows laptops, QA servers running Ubuntu in the cloud, and production data center VMs running Red Hat.

Developers can get going quickly by starting with one of the 13,000+ apps available on Docker Hub. Docker manages and tracks changes and dependencies, making it easier for sysadmins to understand how the apps that developers build work. And with Docker Hub, developers can automate their build pipeline and share artifacts with collaborators through public or private repositories.

Docker helps developers build and ship higher-quality applications, faster." – What is Docker

Prerequisites

I use Oh My Zsh with the Docker plugin for autocompletion of docker commands. YMMV.

Linux

The 3.10.x kernel is the minimum requirement for Docker.

MacOS

10.8 “Mountain Lion” or newer is required.

Installation

Linux

Quick and easy install script provided by Docker:

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

If you’re not willing to run a random shell script, please see the installation instructions for your distribution.

If you are a complete Docker newbie, you should follow the series of tutorials now.

Mac OS X

Download and install Docker Toolbox. If that doesn’t work, see the installation instructions.

Docker used to use boot2docker, but you should be using docker machine now. The Docker website has instructions on how to upgrade. If you have an existing docker instance, you can also install the Docker Machine binaries directly.

Once you’ve installed Docker Toolbox, install a VM with Docker Machine using the VirtualBox provider:

```
docker-machine create --driver=virtualbox default
docker-machine ls
eval "$(docker-machine env default)"
```

Then start up a container:

```
docker run hello-world
```

That’s it, you have a running Docker container.

If you are a complete Docker newbie, you should probably follow the series of tutorials now.

Containers

Your basic isolated Docker process. Containers are to Virtual Machines as threads are to processes. Or you can think of them as chroots on steroids.

Lifecycle

- `docker create` creates a container but does not start it.
- `docker run` creates and starts a container in one operation.
- `docker rm` deletes a container.

If you want a transient container, `docker run --rm` will remove the container after it stops.

If you want to map a directory on the host to a docker container, `docker run -v $HOSTDIR:$DOCKERDIR`. Also see Volumes.

If you want to remove also the volumes associated with the container, the deletion of the container must include the -v switch like in `docker rm -v`.

Starting and Stopping

- `docker start` starts a container so it is running.
- `docker stop` stops a running container.
- `docker restart` stops and starts a container.
- `docker pause` pauses a running container, "freezing" it in place.
- `docker unpause` will unpause a running container.
- `docker wait` blocks until running container stops.
- `docker kill` sends a SIGKILL to a running container.
- `docker attach` will connect to a running container.

If you want to integrate a container with a host process manager, start the daemon with `-r=false` then use `docker start -a`.

If you want to expose container ports through the host, see the exposing ports section.

Restart policies on crashed docker instances are covered here.

Info

- `docker ps` shows running containers.
- `docker logs` gets logs from container.
- `docker inspect` looks at all the info on a container (including IP address).
- `docker events` gets events from container.
- `docker port` shows public facing port of container.
- `docker top` shows running processes in container.
- `docker stats` shows containers' resource usage statistics.
- `docker diff` shows changed files in the container's FS.

`docker ps -a` shows running and stopped containers.

Import / Export

- `docker cp` copies files or folders between a container and the local filesystem…
- `docker export` turns container filesystem into tarball archive stream to STDOUT.

Executing Commands

- `docker exec` to execute a command in container.

To enter a running container, attach a new shell process to a running container called foo, use: `docker exec -it foo /bin/bash`.

Images

Images are just templates for docker containers.

Lifecycle

- `docker images` shows all images.
- `docker import` creates an image from a tarball.
- `docker build` creates image from Dockerfile.
- `docker commit` creates image from a container, pausing it temporarily if it is running.
- `docker rmi` removes an image.
- `docker load` loads an image from a tar archive as STDIN, including images and tags (as of 0.7).
- `docker save` saves an image to a tar archive stream to STDOUT with all parent layers, tags & versions (as of 0.7).

Info

- `docker history` shows history of image.
- `docker tag` tags an image to a name (local or registry).

Cleaning up

While you can use the `docker rmi` command to remove specific images, there's a tool called docker-gc that will clean up images that are no longer used by any containers in a safe manner.

Networks

Docker has a networks feature. Not much is known about it, so this is a good place to expand the cheat sheet. There is a note saying that it's a good way to configure docker containers to talk to each other without using ports. See working with networks for more details.

Lifecycle

- `docker network create`
- `docker network rm`

Info

- `docker network ls`
- `docker network inspect`

Connection

- `docker network connect`
- `docker network disconnect`

Registry & Repository

A repository is a *hosted* collection of tagged images that together create the file system for a container.

A registry is a *host* – a server that stores repositories and provides an HTTP API for managing the uploading and downloading of repositories.

Docker.com hosts its own index to a central registry which contains a large number of repositories. Having said that, the central docker registry does not do a good job of verifying images and should be avoided if you're worried about security.

- `docker login` to login to a registry.
- `docker search` searches registry for image.

- `docker pull` pulls an image from registry to local machine.
- `docker push` pushes an image to the registry from local machine.

Run local registry

Registry implementation has an official image for basic setup that can be launched with `docker run -p 5000:5000 registry` Note that this installation does not have any authorization controls. You may use option `-P -p 127.0.0.1:5000:5000` to limit connections to localhost only. In order to push to this repository tag image with `repositoryHostName:5000/imageName` then push this tag.

Dockerfile

The configuration file. Sets up a Docker container when you run `docker build` on it. Vastly preferable to `docker commit`. If you use jEdit, I've put up a syntax highlighting module for Dockerfile you can use. You may also like to try the tools section.

Instructions

- .dockerignore
- FROM Sets the Base Image for subsequent instructions.
- MAINTAINER Set the Author field of the generated images…
- RUN execute any commands in a new layer on top of the current image and commit the results.
- CMD provide defaults for an executing container.
- EXPOSE informs Docker that the container listens on the specified network ports at runtime. NOTE: does not actually make ports accessible.
- ENV sets environment variable.
- ADD copies new files, directories or remote file to container. Invalidates caches. Avoid `ADD` and use `COPY` instead.
- COPY copies new files or directories to container.
- ENTRYPOINT configures a container that will run as an executable.
- VOLUME creates a mount point for externally mounted volumes or other containers.
- USER sets the user name for following RUN / CMD / ENTRYPOINT commands.
- WORKDIR sets the working directory.

- ARG defines a build-time variable.
- ONBUILD adds a trigger instruction when the image is used as the base for another build.
- STOPSIGNAL sets the system call signal that will be sent to the container to exit.
- LABEL apply key/value metadata to your images, containers, or daemons.

Tutorial

- Flux7's Dockerfile Tutorial

Examples

- Examples
- Best practices for writing Dockerfiles
- Michael Crosby has some more Dockerfiles best practices / take 2.
- Building Good Docker Images / Building Better Docker Images
- Managing Container Configuration with Metadata

Layers

The versioned filesystem in Docker is based on layers. They're like git commits or changesets for filesystems.

Note that if you're using aufs as your filesystem, Docker does not always remove data volumes containers layers when you delete a container! See PR 8484 for more details.

Links

Links are how Docker containers talk to each other through TCP/IP ports. Linking into Redis and Atlassian show worked examples. You can also (in 0.11) resolve links by hostname.

NOTE: If you want containers to ONLY communicate with each other through links, start the docker daemon with `-icc=false` to disable inter process communication.

If you have a container with the name CONTAINER (specified by `docker run --name`

`CONTAINER`) and in the Dockerfile, it has an exposed port:

```
EXPOSE 1337
```

Then if we create another container called LINKED like so:

```
docker run -d --link CONTAINER:ALIAS --name LINKED user/wordpress
```

Then the exposed ports and aliases of CONTAINER will show up in LINKED with the following environment variables:

```
$ALIAS_PORT_1337_TCP_PORT
$ALIAS_PORT_1337_TCP_ADDR
```

And you can connect to it that way.

To delete links, use `docker rm --link`.

If you want to link across docker hosts then you should look at Swarm. This link on stackoverflow provides some good information on different patterns for linking containers across docker hosts.

Volumes

Docker volumes are free-floating filesystems. They don't have to be connected to a particular container. You should use volumes mounted from data-only containers for portability.

Lifecycle

- `docker volume create`
- `docker volume rm`

Info

- `docker volume ls`
- `docker volume inspect`

Volumes are useful in situations where you can't use links (which are TCP/IP only). For instance, if you need to have two docker instances communicate by leaving stuff on the filesystem.

You can mount them in several docker containers at once, using `docker run --volumes-from`.

Because volumes are isolated filesystems, they are often used to store state from computations between transient containers. That is, you can have a stateless and transient container run from a recipe, blow it away, and then have a second instance of the transient container pick up from where the last one left off.

See advanced volumes for more details. Container42 is also helpful.

As of 1.3, you can map MacOS host directories as docker volumes through boot2docker:

```
docker run -v /Users/wsargent/myapp/src:/src
```

You can also use remote NFS volumes if you're feeling brave.

You may also consider running data-only containers as described here to provide some data portability.

Exposing ports

Exposing incoming ports through the host container is fiddly but doable.

This is done by mapping the container port to the host port (only using localhost interface) using `-p`:

```
docker run -p 127.0.0.1:$HOSTPORT:$CONTAINERPORT --name CONTAINER -t someimage
```

You can tell Docker that the container listens on the specified network ports at runtime by using EXPOSE:

```
EXPOSE <CONTAINERPORT>
```

But note that EXPOSE does not expose the port itself, only `-p` will do that.

If you're running Docker in Virtualbox, you then need to forward the port there as well, using forwarded_port. It can be useful to define something in Vagrantfile to expose a range of ports so that you can dynamically map them:

```
Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  ...

  (49000..49900).each do |port|
    config.vm.network :forwarded_port, :host => port, :guest => port
  end

  ...
end
```

If you forget what you mapped the port to on the host container, use `docker port` to show it:

```
docker port CONTAINER $CONTAINERPORT
```

Best Practices

This is where general Docker best practices and war stories go:

- The Rabbit Hole of Using Docker in Automated Tests
- Bridget Kromhout has a useful blog post on running Docker in production at Dramafever.
- There's also a best practices blog post from Lyst.

- A Docker Dev Environment in 24 Hours!
- Building a Development Environment With Docker
- Discourse in a Docker Container

Security

This is where security tips about Docker go. The security page goes into more detail.

First things first: Docker runs as root. If you are in the `docker` group, you effectively have root access. If you expose the docker unix socket to a container, you are giving the container root access to the host. Docker should not be your only defense.

Security Tips

For greatest security, you want to run Docker inside a virtual machine, or on a host. This is straight from the Docker Security Team Lead – slides / notes. Then, run with AppArmor / seccomp / SELinux / grsec etc to limit the container permissions.

Docker image ids are sensitive information and should not be exposed to the outside world. Treat them like passwords.

See the Docker Security Cheat Sheet by Thomas Sjögren: some good stuff about container hardening in there.

Check out the docker bench security script, download the white papers and subscribe to the mailing lists (unfortunately Docker does not have a unique mailing list, only dev / user).

You should start off by using a kernel with unstable patches for grsecurity / pax compiled in, such as Alpine Linux. If you are using grsecurity in production, you should spring for commercial support for the stable patches, same as you would do for RedHat. It's $200 a month, which is nothing to your devops budget.

From the Docker Security Cheat Sheet (it's in PDF which makes it hard to use, so copying below) by Container Solutions:

Turn off interprocess communication with:

```
docker -d --icc=false --iptables
```

Set the container to be read-only:

```
docker run --read-only
```

Verify images with a hashsum:

```
docker pull debian@sha256:a25306f3850e1bd44541976aa7b5fd0a29be
```

Set volumes to be read only:

```
docker run -v $(pwd)/secrets:/secrets:ro debian
```

Set memory and CPU sharing:

```
docker -c 512 -mem 512m
```

Define and run a user in your Dockerfile so you don't run as root inside the container:

```
RUN groupadd -r user && useradd -r -g user user
USER user
```

Security Videos

- Using Docker Safely
- Securing your applications using Docker
- Container security: Do containers actually contain?

Security Roadmap

The Docker roadmap talks about seccomp support. There is an AppArmor policy generator called bane, and they're working on security profiles. There's also work on user namespaces which just made it out of experimental.

Tips

Sources:

- 15 Docker Tips in 5 minutes

Last Ids

```
alias dl='docker ps -l -q'
docker run ubuntu echo hello world
docker commit `dl` helloworld
```

Commit with command (needs Dockerfile)

```
docker commit -run='{"Cmd":["postgres", "-too -many -opts"]}' `dl` postgres
```

Get IP address

```
docker inspect `dl` | grep IPAddress | cut -d '"' -f 4
```

or

```
wget http://stedolan.github.io/jq/download/source/jq-1.3.tar.gz
tar xzvf jq-1.3.tar.gz
cd jq-1.3
./configure && make && sudo make install
docker inspect `dl` | jq -r '.[0].NetworkSettings.IPAddress'
```

or using a go template

```
docker inspect -f '{{ .NetworkSettings.IPAddress }}' <container_name>
```

Get port mapping

```
docker inspect -f '{{range $p, $conf := .NetworkSettings.Ports}} {{$p}} -> {{(index $conf 0).HostPort}} {{end}}' <containername>
```

Find containers by regular expression

```
for i in $(docker ps -a | grep "REGEXP_PATTERN" | cut -f1 -d" "); do echo $i; done`
```

Get Environment Settings

```
docker run --rm ubuntu env
```

Kill running containers

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

Delete old containers

```
docker ps -a | grep 'weeks ago' | awk '{print $1}' | xargs docker rm
```

Delete stopped containers

```
docker rm -v `docker ps -a -q -f status=exited`
```

Delete dangling images

```
docker rmi $(docker images -q -f dangling=true)
```

Delete all images

```
docker rmi $(docker images -q)
```

Delete dangling volumes

As of Docker 1.9:

```
docker volume rm $(docker volume ls -q -f dangling=true)
```

In 1.9.0, the filter `dangling=false` does *not* work - it is ignored and will list all volumes.

Show image dependencies

```
docker images -viz | dot -Tpng -o docker.png
```

Slimming down Docker containers Intercity Blog

- Cleaning APT in a RUN layer

  This should be done in the same layer as other apt commands.

  Otherwise, the previous layers still persist the original information and your images will still be fat.

```
RUN {apt commands} \
 && apt-get clean \
 && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*
```

- Flatten an image

```
ID=$(docker run -d image-name /bin/bash)
docker export $ID | docker import – flat-image-name
```

- For backup

```
ID=$(docker run -d image-name /bin/bash)
(docker export $ID | gzip -c > image.tgz)
gzip -dc image.tgz | docker import - flat-image-name
```

Monitor system resource utilization for running containers

To check the CPU, memory, and network i/o usage of a single container, you can use:

```
docker stats <container>
```

For all containers listed by id:

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

For all containers listed by name:

```
docker stats $(docker ps --format '{{.Names}}')
```