**Table of Contents**

* [Why Docker](https://github.com/wsargent/docker-cheat-sheet#why-docker)
* [Prerequisites](https://github.com/wsargent/docker-cheat-sheet#prerequisites)
* [Installation](https://github.com/wsargent/docker-cheat-sheet#installation)
* [Containers](https://github.com/wsargent/docker-cheat-sheet#containers)
* [Images](https://github.com/wsargent/docker-cheat-sheet#images)
* [Networks](https://github.com/wsargent/docker-cheat-sheet#networks)
* [Registry and Repository](https://github.com/wsargent/docker-cheat-sheet#registry--repository)
* [Dockerfile](https://github.com/wsargent/docker-cheat-sheet#dockerfile)
* [Layers](https://github.com/wsargent/docker-cheat-sheet#layers)
* [Links](https://github.com/wsargent/docker-cheat-sheet#links)
* [Volumes](https://github.com/wsargent/docker-cheat-sheet#volumes)
* [Exposing Ports](https://github.com/wsargent/docker-cheat-sheet#exposing-ports)
* [Best Practices](https://github.com/wsargent/docker-cheat-sheet#best-practices)
* [Security](https://github.com/wsargent/docker-cheat-sheet#security)
* [Tips](https://github.com/wsargent/docker-cheat-sheet#tips)
* [Contributing](https://github.com/wsargent/docker-cheat-sheet#contributing)

**Why Docker**

"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](https://www.docker.com/what-docker#copy1)

**Prerequisites**

I use [Oh My Zsh](https://github.com/robbyrussell/oh-my-zsh) with the [Docker plugin](https://github.com/robbyrussell/oh-my-zsh/wiki/Plugins" \l "docker) for autocompletion of docker commands. YMMV.

**Linux**

The 3.10.x kernel is [the minimum requirement](https://docs.docker.com/engine/installation/binaries/#check-kernel-dependencies) 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](https://docs.docker.com/engine/installation/linux/) instructions for your distribution.

If you are a complete Docker newbie, you should follow the [series of tutorials](https://docs.docker.com/engine/getstarted/) now.

**Mac OS X**

Download and install [Docker Toolbox](https://docs.docker.com/toolbox/overview/). [Docker For Mac](https://docs.docker.com/docker-for-mac/) is nice, but it's not quite as finished as the VirtualBox install. [See the comparison](https://docs.docker.com/docker-for-mac/docker-toolbox/).

**NOTE** If you have an existing docker toolbox, you might think you can upgrade [Docker Machine](https://docs.docker.com/machine/install-machine/) binaries directly (either from URL or docker-machine upgrade default) and it will take care of itself. This is not going to help -- docker-machinewill be 1.10.3 while docker is still 1.8.3 or whatever your previous version is.

You are much better off using Docker Toolbox DMG file to upgrade, which will take care of all the binaries at once.

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](https://docs.docker.com/engine/getstarted/) now.

**Containers**

[Your basic isolated Docker process](http://etherealmind.com/basics-docker-containers-hypervisors-coreos/). Containers are to Virtual Machines as threads are to processes. Or you can think of them as chroots on steroids.

**Lifecycle**

* [docker create](https://docs.docker.com/engine/reference/commandline/create) creates a container but does not start it.
* [docker rename](https://docs.docker.com/engine/reference/commandline/rename/) allows the container to be renamed.
* [docker run](https://docs.docker.com/engine/reference/commandline/run) creates and starts a container in one operation.
* [docker rm](https://docs.docker.com/engine/reference/commandline/rm) deletes a container.
* [docker update](https://docs.docker.com/engine/reference/commandline/update/) updates a container's resource limits.

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](https://github.com/wsargent/docker-cheat-sheet/#volumes).

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

There's also a [logging driver](https://docs.docker.com/engine/admin/logging/overview/) available for individual containers in docker 1.10. To run docker with a custom log driver (i.e., to syslog), use docker run --log-driver=syslog.

**Starting and Stopping**

* [docker start](https://docs.docker.com/engine/reference/commandline/start) starts a container so it is running.
* [docker stop](https://docs.docker.com/engine/reference/commandline/stop) stops a running container.
* [docker restart](https://docs.docker.com/engine/reference/commandline/restart) stops and starts a container.
* [docker pause](https://docs.docker.com/engine/reference/commandline/pause/) pauses a running container, "freezing" it in place.
* [docker unpause](https://docs.docker.com/engine/reference/commandline/unpause/) will unpause a running container.
* [docker wait](https://docs.docker.com/engine/reference/commandline/wait) blocks until running container stops.
* [docker kill](https://docs.docker.com/engine/reference/commandline/kill) sends a SIGKILL to a running container.
* [docker attach](https://docs.docker.com/engine/reference/commandline/attach) will connect to a running container.

If you want to integrate a container with a [host process manager](https://docs.docker.com/engine/admin/host_integration/), 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](https://github.com/wsargent/docker-cheat-sheet#exposing-ports) section.

Restart policies on crashed docker instances are [covered here](http://container42.com/2014/09/30/docker-restart-policies/).

**CPU Constraints**

You can limit CPU, either using a percentage of all CPUs, or by using specific cores.

For example, you can tell the [cpu-shares](https://docs.docker.com/engine/reference/run/#/cpu-share-constraint) setting. The setting is a bit strange -- 1024 means 100% of the CPU, so if you want the container to take 50% of all CPU cores, you should specify 512. See <https://goldmann.pl/blog/2014/09/11/resource-management-in-docker/#_cpu> for more:

docker run -ti --c 512 agileek/cpuset-test

You can also only use some CPU cores using [cpuset-cpus](https://docs.docker.com/engine/reference/run/#/cpuset-constraint). See <https://agileek.github.io/docker/2014/08/06/docker-cpuset/> for details and some nice videos:

docker run -ti --cpuset-cpus=0,4,6 agileek/cpuset-test

Note that Docker can still **see** all of the CPUs inside the container -- it just isn't using all of them. See <https://github.com/docker/docker/issues/20770> for more details.

**Memory Constraints**

You can also set [memory constraints](https://docs.docker.com/engine/reference/run/#/user-memory-constraints) on Docker:

docker run -it -m 300M ubuntu:14.04 /bin/bash

**Capabilities**

Linux capabilities can be set by using cap-add and cap-drop. See <https://docs.docker.com/engine/reference/run/#/runtime-privilege-and-linux-capabilities> for details. This should be used for greater security.

To mount a FUSE based filesystem, you need to combine both --cap-add and --device:

docker run --rm -it --cap-add SYS\_ADMIN --device /dev/fuse sshfs

Give access to a single device:

docker run -it --device=/dev/ttyUSB0 debian bash

Give access to all devices:

docker run -it --privileged -v /dev/bus/usb:/dev/bus/usb debian bash

more info about privileged containers [here](https://docs.docker.com/engine/reference/run/#/runtime-privilege-and-linux-capabilities)

**Info**

* [docker ps](https://docs.docker.com/engine/reference/commandline/ps) shows running containers.
* [docker logs](https://docs.docker.com/engine/reference/commandline/logs) gets logs from container. (You can use a custom log driver, but logs is only available for json-file and journald in 1.10).
* [docker inspect](https://docs.docker.com/engine/reference/commandline/inspect) looks at all the info on a container (including IP address).
* [docker events](https://docs.docker.com/engine/reference/commandline/events) gets events from container.
* [docker port](https://docs.docker.com/engine/reference/commandline/port) shows public facing port of container.
* [docker top](https://docs.docker.com/engine/reference/commandline/top) shows running processes in container.
* [docker stats](https://docs.docker.com/engine/reference/commandline/stats) shows containers' resource usage statistics.
* [docker diff](https://docs.docker.com/engine/reference/commandline/diff) shows changed files in the container's FS.

docker ps -a shows running and stopped containers.

docker stats --all shows a running list of containers.

**Import / Export**

* [docker cp](https://docs.docker.com/engine/reference/commandline/cp) copies files or folders between a container and the local filesystem.
* [docker export](https://docs.docker.com/engine/reference/commandline/export) turns container filesystem into tarball archive stream to STDOUT.

**Executing Commands**

* [docker exec](https://docs.docker.com/engine/reference/commandline/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](https://docs.docker.com/engine/understanding-docker/#how-does-a-docker-image-work).

**Lifecycle**

* [docker images](https://docs.docker.com/engine/reference/commandline/images) shows all images.
* [docker import](https://docs.docker.com/engine/reference/commandline/import) creates an image from a tarball.
* [docker build](https://docs.docker.com/engine/reference/commandline/build) creates image from Dockerfile.
* [docker commit](https://docs.docker.com/engine/reference/commandline/commit) creates image from a container, pausing it temporarily if it is running.
* [docker rmi](https://docs.docker.com/engine/reference/commandline/rmi) removes an image.
* [docker load](https://docs.docker.com/engine/reference/commandline/load) loads an image from a tar archive as STDIN, including images and tags (as of 0.7).
* [docker save](https://docs.docker.com/engine/reference/commandline/save) saves an image to a tar archive stream to STDOUT with all parent layers, tags & versions (as of 0.7).

**Info**

* [docker history](https://docs.docker.com/engine/reference/commandline/history) shows history of image.
* [docker tag](https://docs.docker.com/engine/reference/commandline/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](https://github.com/spotify/docker-gc) that will clean up images that are no longer used by any containers in a safe manner.

**Load/Save image**

Load an image from file:

docker load < my\_image.tar.gz

Save an existing image:

docker save my\_image:my\_tag | gzip > my\_image.tar.gz

**Import/Export container**

Import a container as an image from file:

cat my\_container.tar.gz | docker import - my\_image:my\_tag

Export an existing container:

docker export my\_container | gzip > my\_container.tar.gz

**Difference between loading a saved image and importing an exported container as an image ?**

Loading an image using the load command creates a new image including its history.  
Importing a container as an image using the import command creates a new image excluding the history which results in a smaller image size compared to loading an image.

**Networks**

Docker has a [networks](https://docs.docker.com/engine/userguide/networking/) 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](https://docs.docker.com/engine/userguide/networking/work-with-networks/) for more details.

**Lifecycle**

* [docker network create](https://docs.docker.com/engine/reference/commandline/network_create/)
* [docker network rm](https://docs.docker.com/engine/reference/commandline/network_rm/)

**Info**

* [docker network ls](https://docs.docker.com/engine/reference/commandline/network_ls/)
* [docker network inspect](https://docs.docker.com/engine/reference/commandline/network_inspect/)

**Connection**

* [docker network connect](https://docs.docker.com/engine/reference/commandline/network_connect/)
* [docker network disconnect](https://docs.docker.com/engine/reference/commandline/network_disconnect/)

You can specify a [specific IP address for a container](https://blog.jessfraz.com/post/ips-for-all-the-things/):

# create a new bridge network with your subnet and gateway for your ip block

docker network create --subnet 203.0.113.0/24 --gateway 203.0.113.254 iptastic

# run a nginx container with a specific ip in that block

$ docker run --rm -it --net iptastic --ip 203.0.113.2 nginx

# curl the ip from any other place (assuming this is a public ip block duh)

$ curl 203.0.113.2

**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](https://docs.docker.com/engine/tutorials/dockerrepos/).

Docker.com hosts its own [index](https://hub.docker.com/) 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](https://titanous.com/posts/docker-insecurity) and should be avoided if you're worried about security.

* [docker login](https://docs.docker.com/engine/reference/commandline/login) to login to a registry.
* [docker logout](https://docs.docker.com/engine/reference/commandline/logout) to logout from a registry.
* [docker search](https://docs.docker.com/engine/reference/commandline/search) searches registry for image.
* [docker pull](https://docs.docker.com/engine/reference/commandline/pull) pulls an image from registry to local machine.
* [docker push](https://docs.docker.com/engine/reference/commandline/push) pushes an image to the registry from local machine.

**Run local registry**

You can run a local registry by using the [docker distribution](https://github.com/docker/distribution) project and looking at the [local deploy](https://github.com/docker/docker.github.io/blob/master/registry/deploying.md) instructions.

Also see the [mailing list](https://groups.google.com/a/dockerproject.org/forum/#!forum/distribution).

**Dockerfile**

[The configuration file](https://docs.docker.com/engine/reference/builder/). Sets up a Docker container when you run docker build on it. Vastly preferable to docker commit.

Here are some common text editors and their syntax highlighting modules you could use to create Dockerfiles:

* If you use [jEdit](http://jedit.org/), I've put up a syntax highlighting module for [Dockerfile](https://github.com/wsargent/jedit-docker-mode) you can use.
* [Sublime Text 2](https://packagecontrol.io/packages/Dockerfile%20Syntax%20Highlighting)
* [Atom](https://atom.io/packages/language-docker)
* [Vim](https://github.com/ekalinin/Dockerfile.vim)
* [Emacs](https://github.com/spotify/dockerfile-mode)
* [TextMate](https://github.com/docker/docker/tree/master/contrib/syntax/textmate)
* For a most comprehensive list of editors and IDEs, check [Docker meets the IDE](https://domeide.github.io/)

**Instructions**

* [.dockerignore](https://docs.docker.com/engine/reference/builder/#dockerignore-file)
* [FROM](https://docs.docker.com/engine/reference/builder/#from) Sets the Base Image for subsequent instructions.
* [MAINTAINER](https://docs.docker.com/engine/reference/builder/#maintainer) Set the Author field of the generated images..
* [RUN](https://docs.docker.com/engine/reference/builder/#run) execute any commands in a new layer on top of the current image and commit the results.
* [CMD](https://docs.docker.com/engine/reference/builder/#cmd) provide defaults for an executing container.
* [EXPOSE](https://docs.docker.com/engine/reference/builder/#expose) informs Docker that the container listens on the specified network ports at runtime. NOTE: does not actually make ports accessible.
* [ENV](https://docs.docker.com/engine/reference/builder/#env) sets environment variable.
* [ADD](https://docs.docker.com/engine/reference/builder/#add) copies new files, directories or remote file to container. Invalidates caches. Avoid ADD and use COPY instead.
* [COPY](https://docs.docker.com/engine/reference/builder/#copy) copies new files or directories to container.
* [ENTRYPOINT](https://docs.docker.com/engine/reference/builder/#entrypoint) configures a container that will run as an executable.
* [VOLUME](https://docs.docker.com/engine/reference/builder/#volume) creates a mount point for externally mounted volumes or other containers.
* [USER](https://docs.docker.com/engine/reference/builder/#user) sets the user name for following RUN / CMD / ENTRYPOINT commands.
* [WORKDIR](https://docs.docker.com/engine/reference/builder/#workdir) sets the working directory.
* [ARG](https://docs.docker.com/engine/reference/builder/#arg) defines a build-time variable.
* [ONBUILD](https://docs.docker.com/engine/reference/builder/#onbuild) adds a trigger instruction when the image is used as the base for another build.
* [STOPSIGNAL](https://docs.docker.com/engine/reference/builder/#stopsignal) sets the system call signal that will be sent to the container to exit.
* [LABEL](https://docs.docker.com/engine/userguide/labels-custom-metadata/) apply key/value metadata to your images, containers, or daemons.

**Tutorial**

* [Flux7's Dockerfile Tutorial](http://flux7.com/blogs/docker/docker-tutorial-series-part-3-automation-is-the-word-using-dockerfile/)

**Examples**

* [Examples](https://docs.docker.com/engine/reference/builder/#dockerfile-examples)
* [Best practices for writing Dockerfiles](https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/)
* [Michael Crosby](http://crosbymichael.com/) has some more [Dockerfiles best practices](http://crosbymichael.com/dockerfile-best-practices.html) / [take 2](http://crosbymichael.com/dockerfile-best-practices-take-2.html).
* [Building Good Docker Images](http://jonathan.bergknoff.com/journal/building-good-docker-images) / [Building Better Docker Images](http://jonathan.bergknoff.com/journal/building-better-docker-images)
* [Managing Container Configuration with Metadata](https://speakerdeck.com/garethr/managing-container-configuration-with-metadata)

**Layers**

The versioned filesystem in Docker is based on layers. They're like [git commits or changesets for filesystems](https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/).

Note that if you're using [aufs](https://en.wikipedia.org/wiki/Aufs) as your filesystem, Docker does not always remove data volumes containers layers when you delete a container! See [PR 8484](https://github.com/docker/docker/pull/8484) for more details.

**Links**

Links are how Docker containers talk to each other [through TCP/IP ports](https://docs.docker.com/engine/userguide/networking/default_network/dockerlinks/). [Linking into Redis](https://docs.docker.com/engine/examples/running_redis_service/) and [Atlassian](https://blogs.atlassian.com/2013/11/docker-all-the-things-at-atlassian-automation-and-wiring/) show worked examples. You can also resolve [links by hostname](https://docs.docker.com/engine/userguide/networking/default_network/dockerlinks/#/updating-the-etchosts-file).

NOTE: If you want containers to ONLY communicate with each other through links, start the docker daemon with -icc=falseto 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.

Generally, linking between docker services is a subset of "service discovery", a big problem if you're planning to use Docker at scale in production. Please read [The Docker Ecosystem: Service Discovery and Distributed Configuration Stores](https://www.digitalocean.com/community/tutorials/the-docker-ecosystem-service-discovery-and-distributed-configuration-stores) for more info.

**Volumes**

Docker volumes are [free-floating filesystems](https://docs.docker.com/engine/tutorials/dockervolumes/). They don't have to be connected to a particular container. You should use volumes mounted from [data-only containers](https://medium.com/@ramangupta/why-docker-data-containers-are-good-589b3c6c749e) for portability.

**Lifecycle**

* [docker volume create](https://docs.docker.com/engine/reference/commandline/volume_create/)
* [docker volume rm](https://docs.docker.com/engine/reference/commandline/volume_rm/)

**Info**

* [docker volume ls](https://docs.docker.com/engine/reference/commandline/volume_ls/)
* [docker volume inspect](https://docs.docker.com/engine/reference/commandline/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](http://crosbymichael.com/advanced-docker-volumes.html) for more details. Container42 is [also helpful](http://container42.com/2014/11/03/docker-indepth-volumes/).

You can [map MacOS host directories as docker volumes](https://docs.docker.com/engine/tutorials/dockervolumes/#mount-a-host-directory-as-a-data-volume):

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

You can use remote NFS volumes if you're [feeling brave](https://docs.docker.com/engine/tutorials/dockervolumes/#/mount-a-shared-storage-volume-as-a-data-volume).

You may also consider running data-only containers as described [here](http://container42.com/2013/12/16/persistent-volumes-with-docker-container-as-volume-pattern/) to provide some data portability.

**Exposing ports**

Exposing incoming ports through the host container is [fiddly but doable](https://docs.docker.com/engine/reference/run/#expose-incoming-ports).

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](https://docs.docker.com/engine/reference/builder/#expose):

EXPOSE <CONTAINERPORT>

Note that EXPOSE does not expose the port itself -- only -p will do that. To expose the container's port on your localhost's port:

iptables -t nat -A DOCKER -p tcp --dport <LOCALHOSTPORT> -j DNAT --to-destination <CONTAINERIP>:<PORT>

If you're running Docker in Virtualbox, you then need to forward the port there as well, using [forwarded\_port](https://docs.vagrantup.com/v2/networking/forwarded_ports.html). Define a range of ports in your Vagrantfile like this so 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

**Security**

This is where security tips about Docker go. The Docker [security](https://docs.docker.com/engine/security/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](http://reventlov.com/advisories/using-the-docker-command-to-root-the-host). If you expose the docker unix socket to a container, you are giving the container [root access to the host](https://www.lvh.io/posts/dont-expose-the-docker-socket-not-even-to-a-container.html).

Docker should not be your only defense. You should secure and harden it.

For an understanding of what containers leave exposed, you should read is [Understanding and Hardening Linux Containers](https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2016/april/ncc_group_understanding_hardening_linux_containers-10pdf) by [Aaron Grattafiori](https://twitter.com/dyn___). This is a complete and comprehensive guide to the issues involved with containers, with a plethora of links and footnotes leading on to yet more useful content. The security tips following are useful if you've already hardened containers in the past, but are not a substitute for understanding.

**Security Tips**

For greatest security, you want to run Docker inside a virtual machine. This is straight from the Docker Security Team Lead -- [slides](http://www.slideshare.net/jpetazzo/linux-containers-lxc-docker-and-security) / [notes](http://www.projectatomic.io/blog/2014/08/is-it-safe-a-look-at-docker-and-security-from-linuxcon/). Then, run with AppArmor / seccomp / SELinux / grsec etc to [limit the container permissions](http://linux-audit.com/docker-security-best-practices-for-your-vessel-and-containers/). See the [Docker 1.10 security features](https://blog.docker.com/2016/02/docker-engine-1-10-security/) for more details.

Docker image ids are [sensitive information](https://medium.com/@quayio/your-docker-image-ids-are-secrets-and-its-time-you-treated-them-that-way-f55e9f14c1a4) and should not be exposed to the outside world. Treat them like passwords.

See the [Docker Security Cheat Sheet](https://github.com/konstruktoid/Docker/blob/master/Security/CheatSheet.adoc) by [Thomas Sjögren](https://github.com/konstruktoid): some good stuff about container hardening in there.

Check out the [docker bench security script](https://github.com/docker/docker-bench-security), download the [white papers](https://blog.docker.com/2015/05/understanding-docker-security-and-best-practices/) and subscribe to the [mailing lists](https://www.docker.com/docker-security) (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](https://en.wikipedia.org/wiki/Alpine_Linux). If you are using grsecurity in production, you should spring for [commercial support](https://grsecurity.net/business_support.php) for the [stable patches](https://grsecurity.net/announce.php), same as you would do for RedHat. It's $200 a month, which is nothing to your devops budget.

Since docker 1.11 you can easily limit the number of active processes running inside a container to prevent fork bombs. This requires a linux kernel >= 4.3 with CGROUP\_PIDS=y to be in the kernel configuration.

docker run --pids-limit=64

Also available since docker 1.11 is the ability to prevent processes from gaining new privileges. This feature have been in the linux kernel since version 3.5. You can read more about it in [this](http://www.projectatomic.io/blog/2016/03/no-new-privs-docker/) blog post.

docker run --security-opt=no-new-privileges

From the [Docker Security Cheat Sheet](http://container-solutions.com/content/uploads/2015/06/15.06.15_DockerCheatSheet_A2.pdf) (it's in PDF which makes it hard to use, so copying below) by [Container Solutions](http://container-solutions.com/is-docker-safe-for-production/):

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

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

**User Namespaces**

There's also work on [user namespaces](https://s3hh.wordpress.com/2013/07/19/creating-and-using-containers-without-privilege/) -- it is in 1.10 but is not enabled by default.

To enable user namespaces ("remap the userns") in Ubuntu 15.10, [follow the blog example](https://raesene.github.io/blog/2016/02/04/Docker-User-Namespaces/).

**Security Videos**

* [Using Docker Safely](https://youtu.be/04LOuMgNj9U)
* [Securing your applications using Docker](https://youtu.be/KmxOXmPhZbk)
* [Container security: Do containers actually contain?](https://youtu.be/a9lE9Urr6AQ)
* [Linux Containers: Future or Fantasy?](https://www.youtube.com/watch?v=iN6QbszB1R8)

**Security Roadmap**

The Docker roadmap talks about [seccomp support](https://github.com/docker/docker/blob/master/ROADMAP.md" \l "11-security). There is an AppArmor policy generator called [bane](https://github.com/jfrazelle/bane), and they're working on [security profiles](https://github.com/docker/docker/issues/17142).

**Tips**

Sources:

* [15 Docker Tips in 5 minutes](http://sssslide.com/speakerdeck.com/bmorearty/15-docker-tips-in-5-minutes)
* [CodeFresh Everyday Hacks Docker](https://codefresh.io/blog/everyday-hacks-docker/)

**Prune**

The new [Data Management Commands](https://github.com/docker/docker/pull/26108) have landed as of Docker 1.13:

* docker system prune
* docker volume prune
* docker network prune
* docker container prune
* docker image prune

**df**

docker system df presents a summary of the space currently used by different docker objects.

**Heredoc Docker Container**

docker build -t htop - << EOF

FROM alpine

RUN apk --no-cache add htop

EOF

**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 install [jq](https://stedolan.github.io/jq/):

docker inspect $(dl) | jq -r '.[0].NetworkSettings.IPAddress'

or using a [go template](https://docs.docker.com/engine/reference/commandline/inspect):

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 (see**[**Intercity Blog**](http://bit.ly/1Wwo61N)**)**

* 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}}')

For all containers listed by image:

docker ps -a -f ancestor=ubuntu