Prerequisites

Linux

- The 3.10.x kernel or newer

MacOS

- 10.8 “Mountain Lion” or newer

Installation

Linux

Install script provided by Docker:

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

If unwilling to run a random shell script, please see the installation instructions for your distribution.

Mac OS X

Download and install **Docker Toolbox**. If this fails, see the installation instructions.

If you have an existing Docker Toolbox, you may think you can upgrade Docker Machine binaries directly (either from URL or `docker-machine upgrade default`). This does not work. Your Docker Machine will be 1.10.3, while Docker itself remains at its previous version.

Instead, use the Docker Toolbox DMG file to upgrade; this takes 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.

Containers

Your basic isolated Docker process. Containers are to virtual machines, as threads are to processes. Or you can think of them as larger-than-life `chroot` environments.

Lifecycle

- `docker create` • Creates a container but does not start it
- `docker rename` • Allows the container to be renamed
- `docker run` • Creates and starts a container in one operation
- `docker rm` • Deletes a container
- `docker update` • Updates a container's resource limits
- `docker run --rm` • Removes container when stopped
- `docker run -v $HOSTDIR:$DOCKERDIR` • Maps a directory on the host to the Docker container; see also• Volumes
- `docker rm -v` • Removes volumes associated with container
- `docker run --log-driver=syslog` • Runs Docker with custom log driver

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` • Unpauses a running container
- `docker wait` • Blocks until running container stops
- `docker kill` • Sends a SIGKILL to a running container
- `docker attach` • Connects 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.

Information on Docker Containers, Processes and Performance

- `docker ps` • Shows running containers
- `docker logs` • Gets logs from container; you can use a custom log driver, but logs are only available for `json-file` and `journald` in 1.10
- `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 filesystem
- `docker ps -a` • Shows running and stopped containers
- `docker stats --all` • Shows a running list of containers

Import / Export (Backup / Restore)

- `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` • Executes 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 templates that Docker containers are based on. They are the foundational layer from which your container is launched, and your changes then become independent from it (as another layer).

Lifecycle of Containers (Create, Run, Build, Commit)

- `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 and 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-gci` that will clean up images that are no longer used by any containers in a safe manner.

Images Created by Redirection

Load an image from file:

```
docker load < my_image.tar.gz
```

Save an existing image:

```
docker save my_image•my_tag > 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 > my_container.tar.gz
```

Differences 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.

Private and Public Registries/Repositories

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 (the Docker Hub) which contains a large number of repositories.

- **`docker login`** • Logs into a registry
- **`docker logout`** • Logs out from 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

Running a Local Registry

You can run a local registry by using the docker distribution project and looking at the local deployment instructions.

Dockerfile

The configuration file. Sets up a Docker container when you run `docker build` on it.

- Sections/Directives in a Dockerfile:
  - » **`.dockerignore`** • Files and directories to be ignored during the `build -t` of the Dockerfile
  - » **`FROM`** • Sets the base image for subsequent instructions
  - » **`MAINTAINER`** • Sets the Author field of the generated images
  - » **`RUN`** • Executes any commands in a new layer on top of the current image and commits the results
  - » **`CMD`** • Provides defaults for an executing container
  - » **`EXPOSE`** • Informs Docker that the container listens on the specified network ports at

runtime; does not make ports accessible

» **ENV** • Sets environment variables

» **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 username 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

Layers

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

Links Between Containers

Links are how Docker containers talk to each other through TCP/IP ports. As of 0.11, you can resolve links by hostname.

If you want containers only to communicate with each other through links, start the docker daemon with `-icc=false` to disable interprocess 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 8080
```

Then if we create another container called *LINKED*:

```
docker run -d --link CONTAINER•ALIAS --name LINKED user/example
```

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

```
$ALIAS_PORT_8080_TCP_PORT
$ALIAS_PORT_8080_TCP_ADDR
```

And you can connect to it that way.

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

Volumes

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

Docker Volume Lifecycle

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

Volume Information

- **`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 items 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 states from computations between transient containers. That is, you can have a stateless and transient container run from a recipe/playbook/manifest, blow it away, and then have a second instance of the transient container pick up from where the last one left off.

Exposing Ports

This is done by mapping the container port to the host port (only using *localhost* interface, for example) 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>
```

EXPOSE does not expose the port itself, only **-p** will do that. To expose the container's port on your localhost's port, see above, and add the appropriate ports to your firewall ruleset, as needed.

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

```
docker port CONTAINER $CONTAINERPORT
```

Security Considerations

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. You should secure and harden it.

Security Tips

For the greatest security, you want to run Docker inside a virtual machine. Then, run with AppArmor/`seccomp`/SELinux/`grsec`, etc. to limit the container permissions. See the Docker 1.10 security features for more details.

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

Since Docker 1.11 you can easily limit the number of active processes running inside a container to prevent fork bombs. This requires Linux kernel 4.3 or higher with `CGROUP_PIDS=y` in the kernel configuration.

```
docker run --pids-limit=64
```

Also available since Docker 1.11 is the ability to prevent processes to gain new privileges. This feature is in the Linux kernel since version 3.5.

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

Turn off interprocess communication:

```
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
```