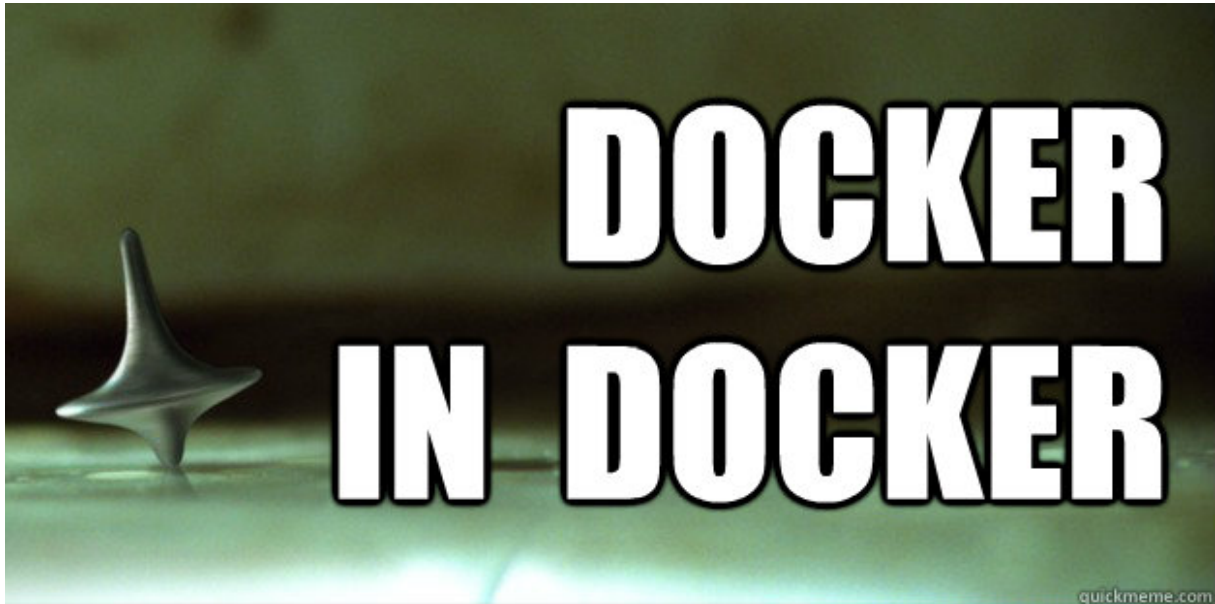

Docker-in-Docker

This recipe lets you run Docker within Docker.



There is only one requirement: your Docker version should support the `--privileged` flag.

A word of warning

If you came here because you would like to run a testing system like Jenkins in a container, and want that container to spin up more containers, then please read this blog post first. Thank you!

Another word of warning

This work is now obsolete, thanks to the combined efforts of some amazing people like @jfrazelle and @tianon, who also are black belts in the art of putting IKEA furniture together.

If you want to run Docker-in-Docker today, all you need to do is:

```
1 docker run --privileged -d docker:dind
```

... And that's it; you get Docker running in Docker, thanks to the official Docker image, in its "Docker-in-Docker" flavor. You can then connect to this Docker instance by starting another Docker container linking to the first one (which is a pretty amazing thing to do).

For more details about the `docker:dind` official image, explanations about how to use it, customize it to use specific storage drivers, and other tidbits of useful knowledge, check its documentation on the Docker Hub.

If you read past this paragraph ...

... Then you're probably an archaeologist, a masochist, or both.

Seriously, though: the information below is here mostly for historical value, or if you want to understand how those things work under the hood.

You've been warned!

Quickstart

Build the image:

```
1 docker build -t dind .
```

Run Docker-in-Docker and get a shell where you can play, and docker daemon logs to stdout:

```
1 docker run --privileged -t -i dind
```

Run Docker-in-Docker and get a shell where you can play, but docker daemon logs into `/var/log/docker.log`:

```
1 docker run --privileged -t -i -e LOG=file dind
```

Run Docker-in-Docker and expose the inside Docker to the outside world:

```
1 docker run --privileged -d -p 4444 -e PORT=4444 dind
```

Note: when started with the `PORT` environment variable, the image will just the Docker daemon and expose it over said port. When started *without* the `PORT` environment variable, the image will run the Docker daemon in the background and execute a shell for you to play.

Daemon configuration

You can use the `DOCKER_DAEMON_ARGS` environment variable to configure the docker daemon with any extra options:

```
1 docker run --privileged -d -e DOCKER_DAEMON_ARGS="-D" dind
```

It didn't work!

If you get a weird permission message, check the output of `dmesg`: it could be caused by AppArmor. In that case, try again, adding an extra flag to kick AppArmor out of the equation:

```
1 docker run --privileged --lxc-conf="lxc.aa_profile=unconfined" -t -i dind
```

If you get the warning:

```
1 WARNING: the 'devices' cgroup should be in its own hierarchy.
```

When starting up dind, you can get around this by shutting down docker and running:

```
1 # /etc/init.d/lxc stop
2 # umount /sys/fs/cgroup/
3 # mount -t cgroup devices 1 /sys/fs/cgroup
```

If the unmount fails, you can find out the proper mount-point with:

```
1 $ cat /proc/mounts | grep cgroup
```

How It Works

The main trick is to have the `--privileged` flag. Then, there are a few things to care about:

- cgroups pseudo-file systems have to be mounted, and they have to be mounted with the same hierarchies than the parent environment; this is done by a wrapper script, which is setup to run by default;
- `/var/lib/docker` cannot be on AUFS, so we make it a volume.

That's it.

Important Warning About Disk Usage

Since AUFS cannot use an AUFS mount as a branch, it means that we have to use a volume. Therefore, all inner Docker data (images, containers, etc.) will be in the volume. Remember: volumes are not cleaned up when you `docker rm`, so if you wonder where did your disk space go after nesting 10 Dockers within each other, look no further :-)

Which Version Of Docker Does It Run?

Outside: it will use your installed version.

Inside: the Dockerfile will retrieve the latest `docker` binary from <https://get.docker.io/>; so if you want to include *your* own `docker` build, you will have to edit it. If you want to always use your local version, you could change the `ADD` line to be e.g.:

```
1 ADD /usr/bin/docker /usr/local/bin/docker
```

Can I Run Docker-in-Docker-in-Docker?

Yes. Note, however, that there seems to be a weird FD leakage issue. To work around it, the `wrapdocker` script carefully closes all the file descriptors inherited from the parent Docker and `lxc-start` (except `stdio`). I'm mentioning this in case you were relying on those inherited file descriptors, or if you're trying to repeat the experiment at home.

`kojiromike/inception` is a wrapper script that uses `dind` to nest Docker to arbitrary depth.

Also, when you will be exiting a nested Docker, this will happen:

```
1 root@975423921ac5:/# exit
2 root@6b2ae8bf2f10:/# exit
3 root@419a67dfdf27:/# exit
4 root@bc9f450caf22:/# exit
5 jpetazzo@tarrasque:~/Work/DOTCLOUD/dind$
```

At that point, you should blast Hans Zimmer's *Dream Is Collapsing* on your loudspeakers while twirling a spinning top.