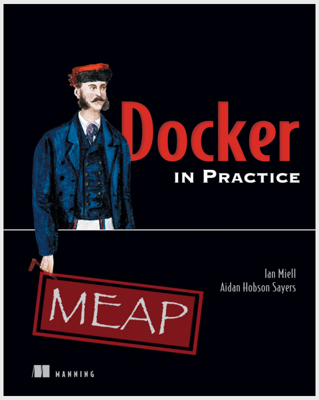

Docker in Practice: The Docker Daemon

By Ian Miell

If you want to gain an understanding of all the relevant pieces of Docker, the Docker daemon is the best place to start. In this article, I'll walk you through the daemon and what it does.

This article is excerpted from Docker in Practice. Save 39% on Docker in Practice with code 15dzamia at manning.com.

If you want to gain an understanding of all the relevant pieces of Docker, the Docker daemon is the best place to start. It is the hub of your interactions with Docker. It controls access to Docker on your machine, manages the state of your containers and images, and brokers interactions with the outside world.

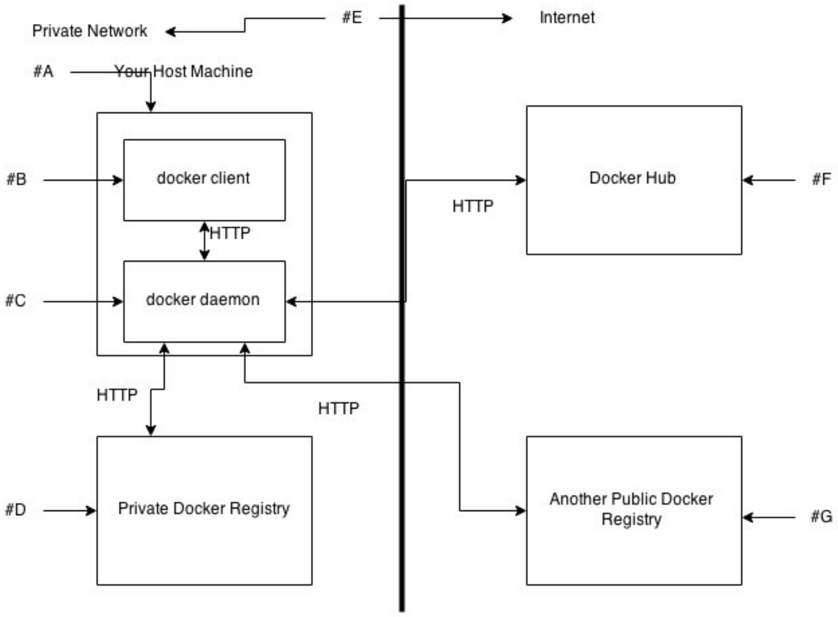

Figure 1 The Docker daemon

TIP Daemons and Servers

A daemon is a process that runs in the background rather than under the direct control of the user. Although you run docker commands on your host machine, these commands do none of the processing on your Docker containers and images. They are frequently also servers that accept requests from clients to perform actions for them.

TECHNIQUE Open your Docker daemon to the world

Although by default your Docker daemon is accessible only on your host, there can be good reason to allow others to access it. You might have a problem that you want someone to debug remotely, or you may want to open up a Docker daemon to allow another part of your DevOps workflow to kick off a process on a host machine.

WARNING Insecure!

Although this can be a powerful and useful technique, it is considered insecure. An open Docker daemon can be exploited by someone that stumbles on it to get escalated privileges.

PROBLEM

You want to open your Docker server up for others to access.

SOLUTION

Start the Docker daemon with an open tcp address.

DISCUSSION

The figure below gives an overview of this technique's workings.

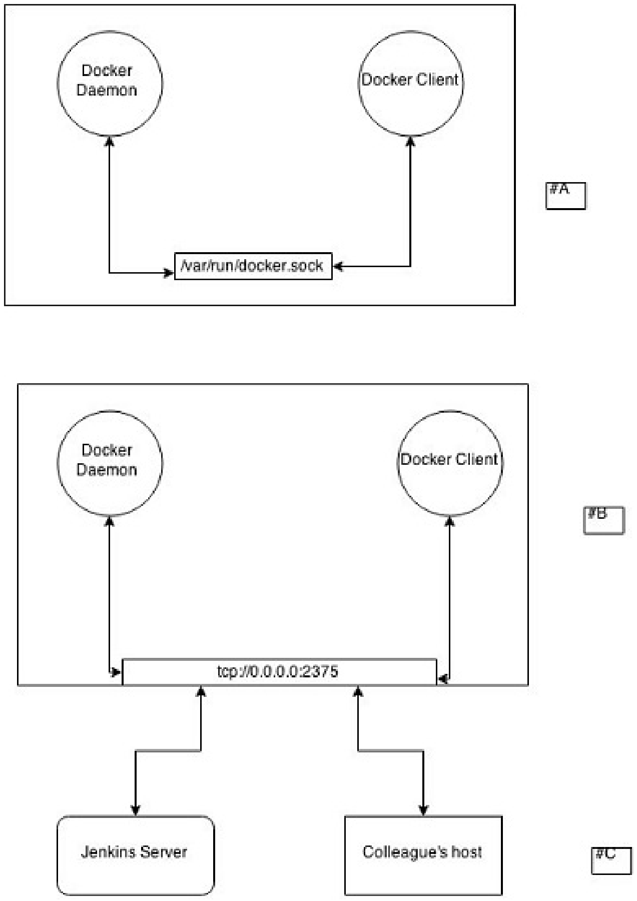

Figure 2 Docker accessibility: normal, and opened up

#A - The default Docker configuration, where access is restricted via the /var/run/docker.sock domain socket. Processes external to the host cannot gain access to Docker.
#B - This technique's open access to the Docker daemon on your host. Access to the Docker daemon is gained through the tcp socket 2375, available to all that can connect to the host.
#C - Third parties can access this Docker daemon. The Jenkins server and colleague's host connect to the host's IP address on port 2375 and can read and write requests and responses using that channel.

Before we open up the Docker daemon we must first shut the running one down. How to do this will vary depending on your operating system. If you're not sure how to do this, you can first try:

$ sudo service docker stop

If you get a message that looks like this:

```
The service command supports only basic LSB actions (start, stop, restart, try-restart, reload, force-reload, status). For other actions, please try to use systemctl.
```

then you have a systemctl-based startup system. Try:

```
$ systemctl stop docker
```

If this works, then you should not see any output from this command:

```
$ ps -ef | grep docker.-d | grep -v grep
```

Once the Docker daemon has been stopped, we can restart it manually and open it up to outside users with the following command:

```
$ docker [#A]-d [#B]-H [#C]tcp://[#D]0.0.0.0:[#E]2375
```

#A - start as a daemon
#B - define the host server with the -H flag
#C - use the tcp protocol
#D - open up to all ip addresses
#E - open on the standard Docker server port (2375) Connect Docker server port (2375) Connect from outside with:

```
$ docker -H http://your host's ip:2375
```

Note that you'll also need to do this from inside your local machine as Docker is no longer listening in the default location!

If you want to make this change permanent on your host, then you'll need to configure your startup system. Check the system documentation on your host OS for how to do this.

TECHNIQUE Running containers as daemons

As you get familiar with Docker (and if you're anything like us!), you will start to think of other use cases for it, and one of the first of these is to run Docker containers as running services.

Running Docker containers as services with predictable behaviour through software isolation is one of the principal use cases of Docker. This technique will allow you to manage services in this use case in a way that works for your operation.

PROBLEM

You want to run a Docker container in the background as a service.

SOLUTION

Use the -d flags to run the Docker run command, and related container management flags to define the service characteristics.

DISCUSSION

Docker containers - like all processes - will run by default in the foreground. The most obvious way to run a Docker container in the background is to use the standard "&" control operator. Although this works, you can run into problems if you log out of your terminal session, necessitating you to use the nohup flag, which creates a file in your local directory with the output that you have to manage… you get the idea. It's far neater to use the Docker daemon's functionality for this.

To achieve this, we use the -d flag.

```
$ docker run [#A]-d [#B]-i [#C]-p 1234:1234 [#D]--name daemon ubuntu [#E]nc -l 1234
```

#A - The -d flag, when used with docker run, runs the container as a daemon.
#B - Give this container the ability to interact with our telnet session
#C - Publish the 1234 port from the container to the host
#D - Give the container a name so we can refer to it later
#E - Run a simple listening echo server on port 1234 with netcat

If we now connect to it and send messages with telnet we can see that the container has received the message by using the docker log command.

Listing 1

```
$ telnet localhost 1234          #A
Trying ::1...
Connected to localhost.
Escape character is '^]'.
hello daemon                     #B
^]                               #C

telnet&gt; q                     #D
Connection closed.
$ docker logs daemon             #E
```

```
hello daemon
$ docker rm daemon              #F
daemon
$
```

#A - Connect to the container's netcat server with the telnet command
#B - Input a line of text to send to the netcat server
#C - Hold "CTRL" and then the "]" at the same time followed by the return key to quit the telnet session
#D - Type "q" and then the return key to quit the telnet program
#E - Run the docker logs command to see the container's output
#F - Clean up the container with the rm command

You can see that running a container as a daemon is simple enough, but, operationally, some questions remain to be answered.

- What happens to the service if it fails?
- What happens to the service when it terminates?
- What happens if the service keeps failing over and over?

Fortunately Docker provides flags to answer each of these questions!

NOTE Flags not required
Although used most often with the daemon flag (-d), technically it's not a requirement to run these flags with -d.

THE RESTART FLAG

Docker run's restart flag allows you to apply a set of rules to follow (a so-called "restart policy") when the container terminates.

Policy	Description
no	Do not restart when container exits
always	Always restart when container exits
on-failure[:max-retry]	Restart only on failure

The "no" policy is simple: when the container exits, it is not restarted. This is the default. The "always" policy is also simple, but worth discussing briefly.

```
$ docker run [#A] -d [#B] --restart=always ubuntu [#C] echo done
```

#A - Run the container as a daemon
#B - Always restart the container on termination
#C - A simple echo command that completes quickly, exiting the container

If you run the above command and then run a "docker ps" command you will see output similar to this:

```
$ docker ps   [#A]
CONTAINER ID        IMAGE               COMMAND                 CREATED
69828b118ec3        ubuntu:14.04        "echo done"     [#B]    4 seconds ago [#C]

STATUS                                  PORTS                   NAMES
Restarting [#D]             (0) Less than a second ago           sick_brattain
```

#A - This command lists all the running containers
#B - When the container was created
#C - The current status of the container - usually this will be in a restarting state, as it will only run for a very short time indeed!
#D - The exit code of the container's previous run. 0 means the run was successful.

It's important to note that Docker reuses the container id. It does not change on restart and there will only ever be one entry in the ps table for this docker invocation!

Finally, the "on-failure" policy is to restart only when the container returns a failing (i.e., non-zero) exit code from its main process.

```
$ docker run [#A] -d [#B] --restart=on-failure:10 ubuntu [#C] /bin/false
```

#A - Running the container as a daemon
#B - Restart only on failure 10 times, then exit
#C - A simple command that completes quickly and will definitely fail!

If you run the above command, wait a minute, and then run a "docker ps -a" command, you will see output similar to this:

```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                 CREATED
b0f40c410fe3        ubuntu:14.04        "/bin/false"            2 minutes ago

STATUS                                  PORTS                   NAMES
Exited (1) 25 seconds ago                                       loving_rosalind
```

TECHNIQUE Moving docker to a different partition

Docker stores all the data relating to your containers and images under a folder. Since as it can store a potentially large number of different images, this folder can get big fast!

If your host machine has different partitions on a machine, (as is common in enterprise Linux workstations) then you may hit this limit more quickly. In these cases, you may want to move the directory from which docker operates.

PROBLEM

You want to move where docker stores its data.

SOLUTION

Stop and start the docker daemon, specifying the new location with the -g flag.

DISCUSSION

First you will need to stop your docker daemon.

Let’s imagine we want to run docker from /home/dockeruser/mydocker. Then when you run:

```
$ docker -g /home/dockeruser/mydocker -d
```

a new set of folders and files will be created in this directory. These folders are internal to docker, so play with them at your peril (as we’ve discovered!).

Be aware that this will appear to wipe the containers and image from your previous docker daemon. But don’t despair. If you kill the docker process you ran above, and restart your docker service, your docker client will be pointed back at its original location and your docker containers and images will be returned to you.

If you want to make this move permanent, you will need to configure your host system’s startup process accordingly.