

# 2

## Docker CLI and Dockerfile

In the last chapter, we set up Docker in our development setup and ran our first container. In this chapter, we will explore the Docker command-line interface. Later in the chapter, we will see how to create our own Docker images using Dockerfiles and how to automate this process.

In this chapter, we will cover the following topics:

- Docker terminologies
- Docker commands
- Dockerfiles
- Docker workflow — pull-use-modify-commit-push workflow
- Automated builds

### Docker terminologies

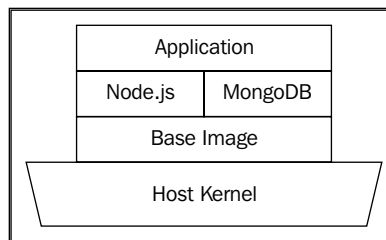
Before we begin our exciting journey into the Docker sphere, let's understand the Docker terminologies that will be used in this book a little better. Very similar in concept to VM images, a Docker image is a snapshot of a system. The difference between a VM image and a Docker image is that a VM image can have running services, whereas a Docker image is just a filesystem snapshot, which means that while you can configure the image to have your favorite packages, you can run only one command in the container. Don't fret though, since the limitation is one command, not one process, so there are ways to get a Docker container to do almost anything a VM instance can.

Docker has also implemented a Git-like distributed version management system for Docker images. Images can be stored in repositories (called a registry), both locally and remotely. The functionalities and terminologies borrow heavily from Git – snapshots are called commits, you pull an image repository, you push your local image to a repository, and so on.

## Docker container

A Docker container can be correlated to an instance of a VM. It runs sandboxed processes that share the same kernel as the host. The term **container** comes from the concept of shipping containers. The idea is that you can ship containers from your development environment to the deployment environment and the applications running in the containers will behave the same way no matter where you run them.

The following image shows the layers of AUFS:



This is similar in context to a shipping container, which stays sealed until delivery but can be loaded, unloaded, stacked, and transported in between.

The visible filesystem of the processes in the container is based on AUFS (although you can configure the container to run with a different filesystem too). AUFS is a layered filesystem. These layers are all read-only and the merger of these layers is what is visible to the processes. However, if a process makes a change in the filesystem, a new layer is created, which represents the difference between the original state and the new state. When you create an image out of this container, the layers are preserved. Thus, it is possible to build new images out of existing images, creating a very convenient hierarchical model of images.

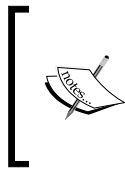
## The docker daemon

The `docker` daemon is the process that manages containers. It is easy to get this confused with the Docker client because the same binary is used to run both the processes. The `docker` daemon, though, needs the `root` privileges, whereas the client doesn't.

Unfortunately, since the `docker` daemon runs with root privileges, it also introduces an attack vector. Read <https://docs.Docker.com/articles/security/> for more details.

## Docker client

The Docker client is what interacts with the `docker` daemon to start or manage containers. Docker uses a RESTful API to communicate between the client and the daemon.



REST is an architectural style consisting of a coordinated set of architectural constraints applied to components, connectors, and data elements within a distributed hypermedia system. In plain words, a RESTful service works over standard HTTP methods such as the GET, POST, PUT, and DELETE methods.

## Dockerfile

A Dockerfile is a file written in a **Domain Specific Language (DSL)** that contains instructions on setting up a Docker image. Think of it as a Makefile equivalent of Docker.

## Docker registry

This is the public repository of all Docker images published by the Docker community. You can pull images from this registry freely, but to push images, you will have to register at <http://hub.docker.com>. Docker registry and Docker hub are services operated and maintained by Docker Inc., and they provide unlimited free repositories. You can also buy private repositories for a fee.

## Docker commands

Now let's get our hands dirty on the Docker CLI. We will look at the most common commands and their use cases. The Docker commands are modeled after Linux and Git, so if you have used either of these, you will find yourself at home with Docker.

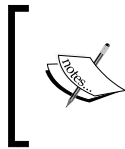
Only the most commonly used options are mentioned here. For the complete reference, you can check out the official documentation at <https://docs.docker.com/reference/commandline/cli/>.

## The daemon command

If you have installed the `docker` daemon through standard repositories, the command to start the `docker` daemon would have been added to the `init` script to automatically start as a service on startup. Otherwise, you will have to first run the `docker` daemon yourself for the client commands to work.

Now, while starting the daemon, you can run it with arguments that control the **Domain Name System (DNS)** configurations, storage drivers, and execution drivers for the containers:

```
$ export DOCKER_HOST="tcp://0.0.0.0:2375"
$ Docker -d -D -e lxc -s btrfs --dns 8.8.8.8 --dns-search example.com
```



You'll need these only if you want to start the daemon yourself. Otherwise, you can start the `docker` daemon with `$ sudo service Docker start`. For OSX and Windows, you need to run the commands mentioned in *Chapter 1, Installing Docker*.

The following table describes the various flags:

Flag	Explanation
<code>-d</code>	This runs Docker as a daemon.
<code>-D</code>	This runs Docker in debug mode.
<code>-e [option]</code>	This is the execution driver to be used. The default execution driver is <code>native</code> , which uses <code>libcontainer</code> .
<code>-s [option]</code>	This forces Docker to use a different storage driver. The default value is <code>""</code> , for which Docker uses <code>AUFS</code> .
<code>--dns [option(s)]</code>	This sets the DNS server (or servers) for all Docker containers.
<code>--dns-search [option(s)]</code>	This sets the DNS search domain (or domains) for all Docker containers.
<code>-H [option(s)]</code>	This is the socket (or sockets) to bind to. It can be one or more of <code>tcp://host:port</code> , <code>unix:///path/to/socket</code> , <code>fd://*</code> or <code>fd://socketfd</code> .

If multiple `docker` daemons are being simultaneously run, the client honors the value set by the `DOCKER_HOST` parameter. You can also make it connect to a specific daemon with the `-H` flag.

Consider this command:

```
$ docker -H tcp://0.0.0.0:2375 run -it ubuntu /bin/bash
```

The preceding command is the same as the following command:

```
$ DOCKER_HOST="tcp://0.0.0.0:2375" docker run -it ubuntu /bin/bash
```

## The version command

The `version` command prints out the version information:

```
$ docker -v
Docker version 1.1.1, build bd609d2
```

## The info command

The `info` command prints the details of the `docker` daemon configuration such as the execution driver, the storage driver being used, and so on:

```
$ docker info # The author is running it in boot2docker on OSX
Containers: 0
Images: 0
Storage Driver: aufs
  Root Dir: /mnt/sda1/var/lib/docker/aufs
  Dirs: 0
Execution Driver: native-0.2
Kernel Version: 3.15.3-tinycore64
Debug mode (server): true
Debug mode (client): false
Fds: 10
Goroutines: 10
EventsListeners: 0
Init Path: /usr/local/bin/docker
Sockets: [unix:///var/run/docker.sock tcp://0.0.0.0:2375]
```

## The run command

The run command is the command that we will be using most frequently. It is used to run Docker containers:

```
$ docker run [options] IMAGE [command] [args]
```

Flags	Explanation
<b>-a, --attach= []</b>	Attach to the <code>stdin</code> , <code>stdout</code> , or <code>stderr</code> files (standard input, output, and error files.).
<b>-d, --detach</b>	This runs the container in the background.
<b>-i, --interactive</b>	This runs the container in interactive mode (keeps the <code>stdin</code> file open).
<b>-t, --tty</b>	This allocates a pseudo <code>tty</code> flag (which is required if you want to attach to the container's terminal).
<b>-p, --publish= []</b>	This publishes a container's port to the host ( <code>ip:hostport:containerport</code> ).
<b>--rm</b>	This automatically removes the container when exited (it cannot be used with the <code>-d</code> flag).
<b>--privileged</b>	This gives additional privileges to this container.
<b>-v, --volume= []</b>	This bind mounts a volume (from host => <code>/host:/container</code> ).
<b>--volumes-from= []</b>	This mounts volumes from specified containers.
<b>-w, --workdir= ""</b>	This is the working directory inside the container.
<b>--name= ""</b>	This assigns a name to the container.
<b>-h, --hostname= ""</b>	This assigns a hostname to the container.
<b>-u, --user= ""</b>	This is the username or UID the container should run on.
<b>-e, --env= []</b>	This sets the environment variables.
<b>--env-file= []</b>	This reads environment variables from a new line-delimited file.
<b>--dns= []</b>	This sets custom DNS servers.
<b>--dns-search= []</b>	This sets custom DNS search domains.
<b>--link= []</b>	This adds link to another container ( <code>name:alias</code> ).
<b>-c, --cpu-shares=0</b>	This is the relative CPU share for this container.
<b>--cpuset= ""</b>	These are the CPUs in which to allow execution; starts with 0. (For example, 0 to 3).
<b>-m, --memory= ""</b>	This is the memory limit for this container ( <code>&lt;number&gt;&lt;b k m g&gt;</code> ).
<b>--restart= ""</b>	(v1.2+) This specifies a restart policy in case the container crashes.

Flags	Explanation
<code>--cap-add=""</code>	(v1.2+) This grants a capability to a container (refer to <i>Chapter 4, Security Best Practices</i> ).
<code>--cap-drop=""</code>	(v1.2+) This blacklists a capability to a container (refer to <i>Chapter 4, Security Best Practices</i> ).
<code>--device=""</code>	(v1.2+) This mounts a device on a container.

While running a container, it is important to keep in mind that the container's lifetime is associated with the lifetime of the command you run when you start the container. Now try to run this:

```
$ docker run -dt ubuntu ps
b1d037dfcfff6b076bde360070d3af0d019269e44929df61c93dfcdfaf29492c9
$ docker attach b1d037
2014/07/16 16:01:29 You cannot attach to a stopped container, start
it first
```

What happened here? When we ran the simple command, `ps`, the container ran the command and exited. Therefore, we got an error.



The `attach` command attaches the standard input and output to a running container.

Another important piece of information here is that you don't need to use the whole 64-character ID for all the commands that require the container ID. The first couple of characters are sufficient. With the same example as shown in the following code:

```
$ docker attach b1d03
2014/07/16 16:09:39 You cannot attach to a stopped container, start
it first
$ docker attach b1d0
2014/07/16 16:09:40 You cannot attach to a stopped container, start
it first
$ docker attach b1d
2014/07/16 16:09:42 You cannot attach to a stopped container, start
it first
$ docker attach b1
2014/07/16 16:09:44 You cannot attach to a stopped container, start
it first
$ docker attach b
2014/07/16 16:09:45 Error: No such container: b
```

A more convenient method though would be to name your containers yourself:

```
$ docker run -dit --name OD-name-example ubuntu /bin/bash
1b21af96c38836df8a809049fb3a040db571cc0cef000a54ebce978c1b5567ea
$ docker attach OD-name-example
root@1b21af96c388:/#
```

The `-i` flag is necessary to have any kind of interaction in the container, and the `-t` flag is necessary to create a pseudo-terminal.

The previous example also made us aware of the fact that even after we exit a container, it is still in a stopped state. That is, we will be able to start the container again, with its filesystem layer preserved. You can see this by running the following command:

```
$ docker ps -a
CONTAINER ID   IMAGE          COMMAND         CREATED        STATUS        NAMES
eb424f5a9d3f   ubuntu:latest  ps             1 hour ago    Exited         OD-name-example
```

While this can be convenient, you may pretty soon have your host's disk space drying up as more and more containers are saved. So, if you are going to run a disposable container, you can run it with the `--rm` flag, which will remove the container when the process exits:

```
$ docker run --rm -it --name OD-rm-example ubuntu /bin/bash
root@0fc99b2e35fb:/# exit
exit
$ docker ps -a
CONTAINER ID   IMAGE          COMMAND         CREATED        STATUS        PORTS        NAMES
```

## Running a server

Now, for our next example, we'll try running a web server. This example is chosen because the most common practical use case of Docker containers is the shipping of web applications:

```
$ docker run -it --name OD-pythonserver-1 --rm python:2.7 \
python -m SimpleHTTPServer 8000;
Serving HTTP on 0.0.0.0 port 8000
```



Now we know the problem; we have a server running in a container, but since the container's IP is assigned by Docker dynamically, it makes things difficult. However, we can bind the container's ports to the host's ports and Docker will take care of forwarding the networking traffic. Now let's try this command again with the `-p` flag:

```
$ docker run -p 0.0.0.0:8000:8000 -it --rm --name OD-pythonserver-2 \
python:2.7 python -m SimpleHTTPServer 8000;
Serving HTTP on 0.0.0.0 port 8000 ...
172.17.42.1 - - [18/Jul/2014 14:25:46] "GET / HTTP/1.1" 200 -
```

Now open your browser and go to `http://localhost:8000`. Voilà!

If you are an OS X user and you realize that you are not able to access `http://localhost:8000`, it is because VirtualBox hasn't been configured to respond to **Network Address Translation (NAT)** requests to the boot2Docker VM. Adding the following function to your aliases file (`bash_profile` or `.bashrc`) will save a lot of trouble:

```
natboot2docker () {
    VBoxManage controlvm boot2docker-vm natpf1 \
        "$1,tcp,127.0.0.1,$2,, $3";
}

removeDockerNat () {
    VBoxManage modifyvm boot2docker-vm \
        --natpf1 delete $1;
}
```

After this, you should be able to use the `$ natboot2docker mypythonserver 8000 8000` command to be able to access the Python server. But remember to run the `$ removeDockerDockerNat mypythonserver` command when you are done. Otherwise, when you run the boot2Docker VM next time, you will be faced with a bug that won't allow you to get the IP address or the ssh script into it:

```
$ boot2docker ssh
ssh_exchange_identification: Connection closed by remote host
2014/07/19 11:55:09 exit status 255
```

Your browser now shows the `/root` path of the container. What if you wanted to serve your host's directories? Let's try mounting a device:

```
root@eb53f7ec79fd:/# mount -t tmpfs /dev/random /mnt
mount: permission denied
```

As you can see, the `mount` command doesn't work. In fact, most kernel capabilities that are potentially dangerous are dropped, unless you include the `--privileged` flag.

However, you should never use this flag unless you know what you are doing. Docker provides a much easier way to bind mount host volumes and bind mount host volumes with the `-v` and `-volumes` options. Let's try this example again in the directory we are currently in:

```
$ docker run -v $(pwd):$(pwd) -p 0.0.0.0:8000:8000 -it -rm \
--name OD-pythonserver-3 python:2.7 python -m SimpleHTTPServer 8000;
Serving HTTP on 0.0.0.0 port 8000 ...
10.0.2.2 - - [18/Jul/2014 14:40:35] "GET / HTTP/1.1" 200 -
```

You have now bound the directory you are running the commands from to the container. However, when you access the container, you still get the directory listing of the root of the container. To serve the directory that has been bound to the container, let's set it as the working directory of the container (the directory the containerized process runs in) using the `-w` flag:

```
$ docker run -v $(pwd):$(pwd) -w $(pwd) -p 0.0.0.0:8000:8000 -it \ --name
OD-pythonserver-4 python:2.7 python -m SimpleHTTPServer 8000;
Serving HTTP on 0.0.0.0 port 8000 ...
10.0.2.2 - - [18/Jul/2014 14:51:35] "GET / HTTP/1.1" 200 -
```



Boot2Docker users will not be able to utilize this yet, unless you use guest additions and set up shared folders, the guide to which can be found at <https://medium.com/boot2docker-lightweight-linux-for-docker/boot2docker-together-with-virtualbox-guest-additions-dale3ab2465c>. Though this solution works, it is a hack and is not recommended. Meanwhile, the Docker community is actively trying to find a solution (check out issue #64 in the boot2Docker GitHub repository and #4023 in the Docker repository).

Now `http://localhost:8000` will serve the directory you are currently running in, but from a Docker container. Take care though, because any changes you make are written into the host's filesystem as well.

⚡ Since v1.1.1, you can bind mount the root of the host to a container using `$ docker run -v /:/my_host:ro ubuntu ls /my_host`, but mounting on the `/` path of the container is forbidden.

The volume can be optionally suffixed with the `:ro` or `:rw` commands to mount the volumes in read-only or read-write mode, respectively. By default, the volumes are mounted in the same mode (read-write or read-only) as they are in the host.

This option is mostly used to mount static assets and to write logs.

But what if I want to mount an external device?

Before v1.2, you had to mount the device in the host and bind mount using the `-v` flag in a privileged container, but v1.2 has added a `--device` flag that you can use to mount a device without needing to use the `--privileged` flag.

For example, to use the webcam in your container, run this command:

```
$ docker run --device=/dev/video0:/dev/video0
```

Docker v1.2 also added a `--restart` flag to specify a restart policy for containers. Currently, there are three restart policies:

- `no`: Do not restart the container if it dies (default).
- `on-failure`: Restart the container if it exits with a non-zero exit code. It can also accept an optional maximum restart count (for example, `on-failure:5`).
- `always`: Always restart the container no matter what exit code is returned.

The following is an example to restart endlessly:

```
$ docker run --restart=always code.it
```

The next line is used to try five times before giving up:

```
$ docker run --restart=on-failure:5 code.it
```

## The search command

The `search` command allows us to search for Docker images in the public registry. Let's search for all images related to Python:

```
$ docker search python | less
```

## The pull command

The `pull` command is used to pull images or repositories from a registry. By default, it pulls them from the public Docker registry, but if you are running your own registry, you can pull them from it too:

```
$ docker pull python # pulls repository from Docker Hub
$ docker pull python:2.7 # pulls the image tagged 2.7
$ docker pull <path_to_registry>/<image_or_repository>
```

## The start command

We saw when we discussed `docker run` that the container state is preserved on exit unless it is explicitly removed. The `docker start` command starts a stopped container:

```
$ docker start [-i] [-a] <container(s)>
```

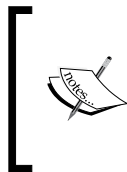
Consider the following example of the `start` command:

```
$ docker ps -a
CONTAINER ID IMAGE          COMMAND                  CREATED STATUS   NAMES
e3c4b6b39cff ubuntu:latest python -m 1h ago   Exited  OD-pythonserver-4
81bb2a92ab0c ubuntu:latest /bin/bash 1h ago   Exited  evil_rosalind
d52fef570d6e ubuntu:latest /bin/bash 1h ago   Exited  prickly_morse
eb424f5a9d3f ubuntu:latest /bin/bash 20h ago  Exited  OD-name-example
$ docker start -ai OD-pythonserver-4
Serving HTTP on 0.0.0.0 port 8000
```

The options have the same meaning as with the `docker run` command.

## The stop command

The `stop` command stops a running container by sending the `SIGTERM` signal and then the `SIGKILL` signal after a grace period:



`SIGTERM` and `SIGKILL` are Unix signals. A signal is a form of interprocess communication used in Unix, Unix-like, and other POSIX-compliant operating systems. `SIGTERM` signals the process to terminate. The `SIGKILL` signal is used to forcibly kill a process.

```

docker run -dit --name OD-stop-example ubuntu /bin/bash
$ docker ps
CONTAINER ID IMAGE          COMMAND          CREATED    STATUS    NAMES
679ece6f2a11 ubuntu:latest /bin/bash 5h ago    Up 3s    OD-stop-example
$ docker stop OD-stop-example
OD-stop-example
$ docker ps

```

```

CONTAINER ID IMAGE          COMMAND          CREATED    STATUS    NAMES

```

You can also specify the `-t` flag or `--time` flag, which allows you to set the wait time.

## The restart command

The `restart` command restarts a running container:

```

$ docker run -dit --name OD-restart-example ubuntu /bin/bash
$ sleep 15s # Suspends execution for 15 seconds
$ docker ps
CONTAINER ID IMAGE          COMMAND          STATUS    NAMES
cc5d0ae0b599 ubuntu:latest /bin/bash Up 20s    OD-restart-example

$ docker restart OD-restart-example
$ docker ps
CONTAINER ID IMAGE          COMMAND          STATUS    NAMES
cc5d0ae0b599 ubuntu:latest /bin/bash Up 2s    OD-restart-example

```

If you observe the status, you will notice that the container was rebooted.

## The rm command

The `rm` command removes Docker containers:

```

$ Docker ps -a # Lists containers including stopped ones
CONTAINER ID IMAGE COMMAND          CREATED    STATUS NAMES
cc5d0ae0b599 ubuntu /bin/bash 6h ago    Exited OD-restart-example
679ece6f2a11 ubuntu /bin/bash 7h ago    Exited OD-stop-example
e3c4b6b39cff ubuntu /bin/bash 9h ago    Exited OD-name-example

```

We seem to be having a lot of containers left over after our adventures. Let's remove one of them:

```
$ docker rm OD-restart-example
cc5d0ae0b599
```

We can also combine two Docker commands. Let's combine the `docker ps -a -q` command, which prints the ID parameters of the containers in the `docker ps -a`, and `docker rm` commands, to remove all containers in one go:

```
$ docker rm $(docker ps -a -q)
679ece6f2a11
e3c4b6b39cff
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              NAMES
```

This evaluates the `docker ps -a -q` command first, and the output is used by the `docker rm` command.

## The ps command

The `ps` command is used to list containers. It is used in the following way:

```
$ docker ps [option(s)]
```

Flag	Explanation
<code>-a, --all</code>	This shows all containers, including stopped ones.
<code>-q, --quiet</code>	This shows only container ID parameters.
<code>-s, --size</code>	This prints the sizes of the containers.
<code>-l, --latest</code>	This shows only the latest container (including stopped containers).
<code>-n=""</code>	This shows the last <i>n</i> containers (including stopped containers). Its default value is -1.
<code>--before=""</code>	This shows the containers created before the specified ID or name. It includes stopped containers.
<code>--after=""</code>	This shows the containers created after the specified ID or name. It includes stopped containers.

The `docker ps` command will show only running containers by default. To see all containers, run the `docker ps -a` command. To see only container ID parameters, run it with the `-q` flag.

## The logs command

The `logs` command shows the logs of the container:

Let us look at the logs of the python server we have been running

```
$ docker logs OD-pythonserver-4
Serving HTTP on 0.0.0.0 port 8000 ...
10.0.2.2 - - [18/Jul/2014 15:06:39] "GET / HTTP/1.1" 200 -
^CTraceback (most recent call last):
  File ...
...
```

KeyboardInterrupt

You can also provide a `--tail` argument to follow the output as the container is running.

## The inspect command

The `inspect` command allows you to get the details of a container or an image. It returns those details as a JSON array:

```
$ Docker inspect ubuntu # Running on an image
[{"Architecture": "amd64",
  "Author": "",
  "Comment": "",
  ".....",
  ".....",
  ".....",
  "DockerVersion": "0.10.0",
  "Id":
  "e54ca5efa2e962582a223ca9810f7f1b62ea9b5c3975d14a5da79d3bf6020f37",
  "Os": "linux",
  "Parent":
  "6c37f792ddacad573016e6aea7fc9fb377127b4767ce6104c9f869314a12041e",
  "Size": 178365
}]
```

Similarly, for a container we run the following command:

```
$ Docker inspect OD-pythonserver-4 # Running on a container
```

```
[{
  "Args": [
    "-m",
    "SimpleHTTPServer",
    "8000"
  ],
  .....,
  "Name": "/OD-pythonserver-4",
  "NetworkSettings": {
    "Bridge": "Docker0",
    "Gateway": "172.17.42.1",
    "IPAddress": "172.17.0.11",
    "IPPrefixLen": 16,
    "PortMapping": null,
    "Ports": {
      "8000/tcp": [
        {
          "HostIp": "0.0.0.0",
          "HostPort": "8000"
        }
      ]
    }
  },
  .....,
  "Volumes": {
    "/home/Docker": "/home/Docker"
  },
  "VolumesRW": {
    "/home/Docker": true
  }
}]
```



Docker inspect provides all of the low-level information about a container or image. In the preceding example, find out the IP address of the container and the exposed port and make a request to the `IP:port`. You will see that you are directly accessing the server running in the container.

However, manually looking through the entire JSON array is not optimal. So the `inspect` command provides a flag, `-f` (or the `--format` flag), which allows you to specify exactly what you want using Go templates. For example, if you just want to get the container's IP address, run the following command:

```
$ docker inspect -f '{{.NetworkSettings.IPAddress}}' \
OD-pythonserver-4;
172.17.0.11
```

The `{{.NetworkSettings.IPAddress}}` is a Go template that was executed over the JSON result. Go templates are very powerful, and some of the things that you can do with them have been listed at <http://golang.org/pkg/text/template/>.

## The top command

The `top` command shows the running processes in a container and their statistics, mimicking the Unix `top` command.

Let's download and run the ghost blogging platform and check out what processes are running in it:

```
$ docker run -d -p 4000:2368 --name OD-ghost dockerfile/ghost
ece88c79b0793b0a49e3d23e2b0b8e75d89c519e5987172951ea8d30d96a2936
```

```
$ docker top OD-ghost-1
```

PID	USER	COMMAND
1162	root	bash /ghost-start
1180	root	npm
1186	root	sh -c node index
1187	root	node index

Yes! We just set up our very own ghost blog, with just one command. This brings forth another subtle advantage and shows something that could be a future trend. Every tool that exposes its services through a TCP port can now be containerized and run in its own sandboxed world. All you need to do is expose its port and bind it to your host port. You don't need to worry about installations, dependencies, incompatibilities, and so on, and the uninstallation will be clean because all you need to do is stop all the containers and remove the image.



Ghost is an open source publishing platform that is beautifully designed, easy to use, and free for everyone. It is coded in Node.js, a server-side JavaScript execution engine.

## The attach command

The `attach` command attaches to a running container.

Let's start a container with Node.js, running the node interactive shell as a daemon, and later attach to it.



Node.js is an event-driven, asynchronous I/O web framework that runs applications written in JavaScript on Google's V8 runtime environment.

The container with Node.js is as follows:

```
$ docker run -dit --name OD-nodejs shykes/nodejs node  
8e0da647200efe33a9dd53d45ea38e3af3892b04aa8b7a6e167b3c093e522754
```

```
$ docker attach OD-nodejs  
console.log('Docker rocks!');Docker rocks!
```

## The kill command

The `kill` command kills a container and sends the `SIGTERM` signal to the process running in the container:

Let us kill the container running the ghost blog.

```
$ docker kill OD-ghost-1  
OD-ghost-1
```

```
$ docker attach OD-ghost-1 # Verification  
2014/07/19 18:12:51 You cannot attach to a stopped container, start  
it first
```

## The cp command

The `cp` command copies a file or folder from a container's filesystem to the host path. Paths are relative to the root of the filesystem.

It's time to have some fun. First, let's run an Ubuntu container with the `/bin/bash` command:

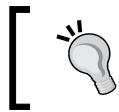
```
$ docker run -it --name OD-cp-bell ubuntu /bin/bash
```

Now, inside the container, let's create a file with a special name:

```
# touch $(echo -e '\007')
```

The `\007` character is an ASCII BEL character that rings the system bell when printed on a terminal. You might have already guessed what we're about to do. So let's open a new terminal and execute the following command to copy this newly created file to the host:

```
$ docker cp OD-cp-bell:/${echo -e '\007'} $(pwd)
```

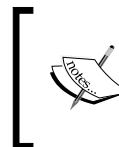


For the `docker cp` command to work, both the container path and the host path must be complete, so do not use shortcuts such as `.`, `..`, `*`, and so on.

So we created an empty file whose filename is the BEL character, in a container. Then we copied the file to the current directory in the host container. Just one last step is remaining. In the host tab where you executed the `docker cp` command, run the following command:

```
$ echo *
```

You will hear the system bell ring! We could have copied any file or directory from the container to the host. But it doesn't hurt to have some fun!



If you found this interesting, you might like to read <http://www.dwheeler.com/essays/fixing-unix-linux-filenames.html>. This is a great essay that discusses the edge cases in filenames, which can cause simple to complicated issues in a program.

## The port command

The `port` command looks up the public-facing port that is bound to an exposed port in the container:

```
$ docker port CONTAINER PRIVATE_PORT
```

```
$ docker port OD-ghost 2368
```

```
4000
```

Ghost runs a server at the 2368 port that allows you to write and publish a blog post. We bound a host port to the `OD-ghost` container's port 2368 in the example for the `top` command.

## Running your own project

By now, we are considerably familiar with the basic Docker commands. Let's up the ante. For the next couple of commands, I am going to use one of my side projects. Feel free to use a project of your own.

Let's start by listing out our requirements to determine the arguments we must pass to the `docker run` command.

Our application is going to run on Node.js, so we will choose the well-maintained `dockerfile/nodejs` image to start our base container:

- We know that our application is going to bind to port 8000, so we will expose the port to 8000 of the host.
- We need to give a descriptive name to the container so that we can reference it in future commands. In this case, let's choose the name of the application:

```
$ docker run -it --name code.it dockerfile/nodejs /bin/bash
[ root@3b0d5a04cdcd:/data ]$ cd /home
[ root@3b0d5a04cdcd:/home ]$
```

Once you have started your container, you need to check whether the dependencies for your application are already available. In our case, we only need Git (apart from Node.js), which is already installed in the `dockerfile/nodejs` image.

Now that our container is ready to run our application, all that is remaining is for us to fetch the source code and do the necessary setup to run the application:

```
$ git clone https://github.com/shrikrishnaholla/code.it.git
$ cd code.it && git submodule update --init --recursive
```

This downloads the source code for a plugin used in the application.

Then run the following command:

```
$ npm install
```

Now all the node modules required to run the application are installed.

Next, run this command:

```
$ node app.js
```

Now you can go to `localhost:8000` to use the application.

## The diff command

The `diff` command shows the difference between the container and the image it is based on. In this example, we are running a container with `code.it`. In a separate tab, run this command:

```
$ docker diff code.it
C /home
A /home/code.it
...
```

## The commit command

The `commit` command creates a new image with the filesystem of the container. Just as with Git's `commit` command, you can set a commit message that describes the image:

```
$ docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]
```

Flag	Explanation
<code>-p, --pause</code>	This pause the container during commit (available from v1.1.1+ onwards).
<code>-m, --message=" "</code>	This is a commit message. It can be a description of what the image does.
<code>-a, --author=" "</code>	This displays the author details.

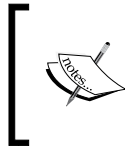
For example, let's use this command to commit the container we have set up:

```
$ docker commit -m "Code.it - A browser based text editor and interpreter" -a "Shrikrishna Holla <s**a@gmail.com>" code.it shrikrishna/code.it:v1
```



Replace the author details and the username portion of the image name in this example if you are copying these examples.

The output will be a lengthy image ID. If you look at the command closely, we have named the image `shrikrishna/code.it:v1`. This is a convention. The first part of an image/repository's name (before the forward slash) is the Docker Hub username of the author. The second part is the intended application or image name. The third part is a tag (usually a version description) separated from the second part by a colon.



Docker Hub is a public registry maintained by Docker, Inc. It hosts public Docker images and provides services to help you build and manage your Docker environment. More details about it can be found at <https://hub.docker.com>.

A collection of images tagged with different versions is a repository. The image you create by running the `docker commit` command will be a local one, which means that you will be able to run containers from it but it won't be available publicly. To make it public or to push to your private Docker registry, use the `docker push` command.

## The images command

The `images` command lists all the images in the system:

```
$ docker images [OPTIONS] [NAME]
```

Flag	Explanation
<code>-a, --all</code>	This shows all images, including intermediate layers.
<code>-f, --filter=[]</code>	This provides filter values.
<code>--no-trunc</code>	This doesn't truncate output (shows complete ID).
<code>-q, --quiet</code>	This shows only the image IDs.

Now let's look at a few examples of the usage of the `image` command:

```
$ docker images
```

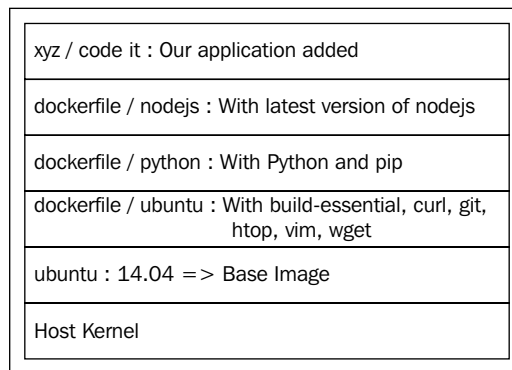
REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
shrikrishna/code.it	v1	a7cb6737a2f6	6m ago	704.4 MB

This lists all top-level images, their repository and tags, and their virtual size.

Docker images are nothing but a stack of read-only filesystem layers. A union filesystem, such as AUFS, then merges these layers and they appear to be one filesystem.

In Docker-speak, a read-only layer is an image. It never changes. When running a container, the processes think that the entire filesystem is read-write. But the changes go only at the topmost writeable layer, which is created when a container is started. The read-only layers of the image remain unchanged. When you commit a container, it freezes the top layer (the underlying layers are already frozen) and turns it into an image. Now, when a container is started this image, all the layers of the image (including the previously writeable layer) are read-only. All the changes are now made to a new writeable layer on top of all the underlying layers. However, because of how union filesystems (such as AUFS) work, the processes believe that the filesystem is read-write.

A rough schematic of the layers involved in our `code.it` example is as follows:



At this point, it might be wise to think just how much effort is to be made by the union filesystems to merge all of these layers and provide a consistent performance. After some point, things inevitably break. AUFS, for instance, has a 42-layer limit. When the number of layers goes beyond this, it just doesn't allow the creation of any more layers and the build fails. Read <https://github.com/docker/docker/issues/1171> for more information on this issue.

The following command lists the most recently created images:

```
$ docker images | head
```

The `-f` flag can be given arguments of the `key=value` type. It is frequently used to get the list of dangling images:

```
$ docker images -f "dangling=true"
```

This will display untagged images, that is, images that have been committed or built without a tag.

## The rmi command

The `rmi` command removes images. Removing an image also removes all the underlying images that it depends on and were downloaded when it was pulled:

```
$ docker rmi [OPTION] {IMAGE(s)}
```

Flag	Explanation
<code>-f, --force</code>	This forcibly removes the image (or images).
<code>--no-prune</code>	This command does not delete untagged parents.

This command removes one of the images from your machine:

```
$ docker rmi test
```

## The save command

The `save` command saves an image or repository in a tarball and this streams to the `stdout` file, preserving the parent layers and metadata about the image:

```
$ docker save -o codeit.tar code.it
```

The `-o` flag allows us to specify a file instead of streaming to the `stdout` file. It is used to create a backup that can then be used with the `docker load` command.

## The load command

The `load` command loads an image from a tarball, restoring the filesystem layers and the metadata associated with the image:

```
$ docker load -i codeit.tar
```

The `-i` flag allows us to specify a file instead of trying to get a stream from the `stdin` file.

## The export command

The `export` command saves the filesystem of a container as a tarball and streams to the `stdout` file. It flattens filesystem layers. In other words, it merges all the filesystem layers. All of the metadata of the image history is lost in this process:

```
$ sudo Docker export red_panda > latest.tar
```

Here, `red_panda` is the name of one of my containers.



## The import command

The `import` command creates an empty filesystem image and imports the contents of the tarball to it. You have the option of tagging it the image:

```
$ docker import URL|- [REPOSITORY[:TAG]]
```

URLs must start with `http`.

```
$ docker import http://example.com/test.tar.gz # Sample url
```

If you would like to import from a local directory or archive, you can use the `-` parameter to take the data from the `stdin` file:

```
$ cat sample.tgz | docker import - testimage:imported
```

## The tag command

You can add a `tag` command to an image. It helps identify a specific version of an image.

For example, the `python` image name represents `python:latest`, the latest version of Python available, which can change from time to time. But whenever it is updated, the older versions are tagged with the respective Python versions. So the `python:2.7` command will have Python 2.7 installed. Thus, the `tag` command can be used to represent versions of the images, or for any other purposes that need identification of the different versions of the image:

```
$ docker tag IMAGE [REGISTRYHOST/] [USERNAME/] NAME[:TAG]
```

The `REGISTRYHOST` command is only needed if you are using a private registry of your own. The same image can have multiple tags:

```
$ docker tag shrikrishna/code.it:v1 shrikrishna/code.it:latest
```



Whenever you are tagging an image, follow the `username/repository:tag` convention.

Now, running the `docker images` command again will show that the same image has been tagged with both the `v1` and `latest` commands:

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
shrikrishna/code.it	v1	a7cb6737a2f6	8 days ago	704.4 MB
shrikrishna/code.it	latest	a7cb6737a2f6	8 days ago	704.4 MB

## The login command

The `login` command is used to register or log in to a Docker registry server. If no server is specified, `https://index.docker.io/v1/` is the default:

```
$ docker login [OPTIONS] [SERVER]
```

Flag	Explanation
<code>-e, --email=""</code>	Email
<code>-p, --password=""</code>	Password
<code>-u, --username=""</code>	Username

If the flags haven't been provided, the server will prompt you to provide the details. After the first login, the details will be stored in the `$HOME/.dockercfg` path.

## The push command

The `push` command is used to push an image to the public image registry or a private Docker registry:

```
$ docker push NAME[:TAG]
```

## The history command

The `history` command shows the history of the image:

```
$ docker history shykes/nodejs
```

IMAGE	CREATED	CREATED BY	SIZE
6592508b0790	15 months ago	/bin/sh -c wget http://nodejs.	15.07 MB
0a2ff988ae20	15 months ago	/bin/sh -c apt-get install ...	25.49 MB
43c5d81f45de	15 months ago	/bin/sh -c apt-get update	96.48 MB
b750fe79269d	16 months ago	/bin/bash	77 B
27cf78414709	16 months ago		175.3 MB

## The events command

Once started, the `events` command prints all the events that are handled by the docker daemon, in real time:

```
$ docker events [OPTIONS]
```

---

Flag	Explanation
<code>--since=""</code>	This shows all events created since timestamp (in Unix).
<code>--until=""</code>	This stream events until timestamp.

For example the `events` command is used as follows:

```
$ docker events
```

Now, in a different tab, run this command:

```
$ docker start code.it
```

Then run the following command:

```
$ docker stop code.it
```

Now go back to the tab running Docker events and see the output. It will be along these lines:

```
[2014-07-21 21:31:50 +0530 IST]
c7f2485863b2c7d0071477e6cb8c8301021ef9036afd4620702a0de08a4b3f7b: (from
dockerfile/nodejs:latest) start
```

```
[2014-07-21 21:31:57 +0530 IST]
c7f2485863b2c7d0071477e6cb8c8301021ef9036afd4620702a0de08a4b3f7b: (from
dockerfile/nodejs:latest) stop
```

```
[2014-07-21 21:31:57 +0530 IST]
c7f2485863b2c7d0071477e6cb8c8301021ef9036afd4620702a0de08a4b3f7b: (from
dockerfile/nodejs:latest) die
```

You can use flags such as `--since` and `--until` to get the event logs of specific timeframes.

## The wait command

The `wait` command blocks until a container stops, then prints its exit code:

```
$ docker wait CONTAINER(s)
```

## The build command

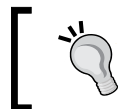
The build command builds an image from the source files at a specified path:

```
$ Docker build [OPTIONS] PATH | URL | -
```

Flag	Explanation
<code>-t, --tag=""</code>	This is the repository name (and an optional tag) to be applied to the resulting image in case of success.
<code>-q, --quiet</code>	This suppresses the output, which by default is verbose.
<code>--rm=true</code>	This removes intermediate containers after a successful build.
<code>--force-rm</code>	This always removes intermediate containers, even after unsuccessful builds.
<code>--no-cache</code>	This command does not use the cache while building the image.

This command uses a Dockerfile and a context to build a Docker image.

A Dockerfile is like a Makefile. It contains instructions on the various configurations and commands that need to be run in order to create an image. We will look at writing Dockerfiles in the next section.



It would be a good idea to read the section about Dockerfiles first and then come back here to get a better understanding of this command and how it works.

The files at the `PATH` or `URL` paths are called **context** of the build. The context is used to refer to the files or folders in the Dockerfile, for instance in the `ADD` instruction (and that is the reason an instruction such as `ADD ../file.txt` won't work. It's not in the context!).

When a GitHub URL or a URL with the `git://` protocol is given, the repository is used as the context. The repository and its submodules are recursively cloned in your local machine, and then uploaded to the `docker` daemon as the context. This allows you to have Dockerfiles in your private Git repositories, which you can access from your local user credentials or from the **Virtual Private Network (VPN)**.

## Uploading to Docker daemon

Remember that Docker engine has both the `docker` daemon and the Docker client. The commands that you give as a user are through the Docker client, which then talks to the `docker` daemon (either through a TCP or a Unix socket), which does the necessary work. The `docker` daemon and Docker host can be in different hosts (which is the premise with which `boot2Docker` works), with the `DOCKER_HOST` environment variable set to the location of the remote `docker` daemon.

When you give a context to the `docker build` command, all the files in the local directory get tared and are sent to the `docker` daemon. The `PATH` variable specifies where to find the files for the context of the build in the `docker` daemon. So when you run `docker build .`, all the files in the current folder get uploaded, not just the ones listed to be added in the Dockerfile.

Since this can be a bit of a problem (as some systems such as Git and some IDEs such as Eclipse create hidden folders to store metadata), Docker provides a mechanism to ignore certain files or folders by creating a file called `.dockerignore` in the `PATH` variable with the necessary exclusion patterns. For an example, look up <https://github.com/docker/docker/blob/master/.dockerignore>.

If a plain URL is given or if the Dockerfile is streamed through the `stdin` file, then no context is set. In these cases, the `ADD` instruction works only if it refers to a remote URL.

Now let's build the `code.it` example image through a Dockerfile. The instructions on how to create this Dockerfile are provided in the *Dockerfile* section.

At this point, you would have created a directory and placed the Dockerfile inside it. Now, on your terminal, go to that directory and execute the `docker build` command:

```
$ docker build -t shrikrishna/code.it:docker Dockerfile .
Sending build context to Docker daemon 2.56 kB
Sending build context to Docker daemon
Step 0 : FROM Dockerfile/nodejs
--> 1535da87b710
```

```
Step 1 : MAINTAINER Shrikrishna Holla <s**a@gmail.com>
---> Running in e4be61c08592
---> 4c0eabc44a95
Removing intermediate container e4be61c08592
Step 2 : WORKDIR /home
---> Running in 067e8951cb22
---> 81ead6b62246
Removing intermediate container 067e8951cb22
. . . . .
. . . . .
Step 7 : EXPOSE 8000
---> Running in 201e07ec35d3
---> 1db6830431cd
Removing intermediate container 201e07ec35d3
Step 8 : WORKDIR /home
---> Running in cd128a6f090c
---> ba05b89b9cc1
Removing intermediate container cd128a6f090c
Step 9 : CMD ["usr/bin/node", "/home/code.it/app.js"]
---> Running in 6da5d364e3e1
---> 031e9ed9352c
Removing intermediate container 6da5d364e3e1
Successfully built 031e9ed9352c
```

Now, you will be able to look at your newly built image in the output of Docker images

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
shrikrishna/code.it	Dockerfile	031e9ed9352c	21 hours ago	1.02 GB


To see the caching in action, run the same command again

```
$ docker build -t shrikrishna/code.it:dockerfile .
Sending build context to Docker daemon 2.56 kB
Sending build context to Docker daemon
Step 0 : FROM dockerfile/nodejs
---> 1535da87b710
Step 1 : MAINTAINER Shrikrishna Holla <s**a@gmail.com>
---> Using cache
---> 4c0eabc44a95
```

```

Step 2 : WORKDIR /home
---> Using cache
---> 81ead6b62246
Step 3 : RUN      git clone https://github.com/shrikrishnaholla/code.
it.git
---> Using cache
---> adb4843236d4
Step 4 : WORKDIR code.it
---> Using cache
---> 755d248840bb
Step 5 : RUN      git submodule update --init --recursive
---> Using cache
---> 2204a519efd3
Step 6 : RUN      npm install
---> Using cache
---> 501e028d7945
Step 7 : EXPOSE   8000
---> Using cache
---> 1db6830431cd
Step 8 : WORKDIR /home
---> Using cache
---> ba05b89b9cc1
Step 9 : CMD      ["/usr/bin/node", "/home/code.it/app.js"]
---> Using cache
---> 031e9ed9352c
Successfully built 031e9ed9352c

```

[  Now experiment with this caching. Change one of the lines in the middle (the port number for example), or add a `RUN echo "testing cache"` line somewhere in the middle and see what happens. ]

An example of building an image using a repository URL is as follows:

```

$ docker build -t shrikrishna/optimus:git_url \ git://github.com/
shrikrishnaholla/optimus
Sending build context to Docker daemon 1.305 MB
Sending build context to Docker daemon
Step 0 : FROM      dockerfile/nodejs

```

```
---> 1535da87b710
Step 1 : MAINTAINER Shrikrishna Holla
---> Running in d2aae3dba68c
---> 0e8636eac25b
Removing intermediate container d2aae3dba68c
Step 2 : RUN          git clone https://github.com/pesos/optimus.git
/home/optimus
---> Running in 0b46e254e90a
. . . . .
. . . . .
. . . . .
Step 5 : CMD          ["/usr/local/bin/npm", "start"]
---> Running in 0e01c71faa0b
---> 0f0dd3deae65
Removing intermediate container 0e01c71faa0b
Successfully built 0f0dd3deae65
```

## Dockerfile

We have seen how to create images by committing containers. What if you want to update the image with new versions of dependencies or new versions of your own application? It soon becomes impractical to do the steps of starting, setting up, and committing over and over again. We need a repeatable method to build images. In comes Dockerfile, which is nothing more than a text file that contains instructions to automate the steps you would otherwise have taken to build an image. `docker build` will read these instructions sequentially, committing them along the way, and build an image.

The `docker build` command takes this Dockerfile and a context to execute the instructions, and builds a Docker image. Context refers to the path or source code repository URL given to the `docker build` command.

A Dockerfile contains instructions in this format:

```
# Comment
INSTRUCTION arguments
```



Any line beginning with # will be considered as a comment. If a # sign is present anywhere else, it will be considered a part of arguments. The instruction is not case-sensitive, although it is an accepted convention for instructions to be uppercase so as to distinguish them from the arguments.

Let's look at the instructions that we can use in a Dockerfile.

## The FROM instruction

The FROM instruction sets the base image for the subsequent instructions. A valid Dockerfile's first non-comment line will be a FROM instruction:

```
FROM <image>:<tag>
```

The image can be any valid local or public image. If it is not found locally, the Docker build command will try to pull it from the public registry. The tag command is optional here. If it is not given, the latest command is assumed. If the incorrect tag command is given, it returns an error.

## The MAINTAINER instruction

The MAINTAINER instruction allows you to set the author for the generated images:

```
MAINTAINER <name>
```

## The RUN instruction

The RUN instruction will execute any command in a new layer on top of the current image, and commit this image. The image thus committed will be used for the next instruction in the Dockerfile.

The RUN instruction has two forms:

- The RUN <command> form
- The RUN ["executable", "arg1", "arg2"...] form

In the first form, the command is run in a shell, specifically the /bin/sh -c <command> shell. The second form is useful in instances where the base image doesn't have a /bin/sh shell. Docker uses a cache for these image builds. So in case your image build fails somewhere in the middle, the next run will reuse the previously successful partial builds and continue from the point where it failed.

The cache will be invalidated in these situations:

- When the `docker build` command is run with the `--no-cache` flag.
- When a non-cacheable command such as `apt-get update` is given. All the following `RUN` instructions will be run again.
- When the first encountered `ADD` instruction will invalidate the cache for all the following instructions from the Dockerfile if the contents of the context have changed. This will also invalidate the cache for the `RUN` instructions.

## The CMD instruction

The `CMD` instruction provides the default command for a container to execute. It has the following forms:

- The `CMD ["executable", "arg1", "arg2"...]` form
- The `CMD ["arg1", "arg2"...]` form
- The `CMD command arg1 arg2 ...` form

The first form is like an `exec` and it is the preferred form, where the first value is the path to the executable and is followed by the arguments to it.

The second form omits the executable but requires the `ENTRYPOINT` instruction to specify the executable.

If you use the shell form of the `CMD` instruction, then the `<command>` command will execute in the `/bin/sh -c shell`.



If the user provides a command in `docker run`, it overrides the `CMD` command.

The difference between the `RUN` and `CMD` instructions is that a `RUN` instruction actually runs the command and commits it, whereas the `CMD` instruction is not executed during build time. It is a default command to be run when the user starts a container, unless the user provides a command to start it with.

For example, let's write a Dockerfile that brings a Star Wars output to your terminal:

```
FROM ubuntu:14.04
MAINTAINER shrikrishna
RUN apt-get -y install telnet
CMD ["/usr/bin/telnet", "towel.blinkenlights.nl"]
```

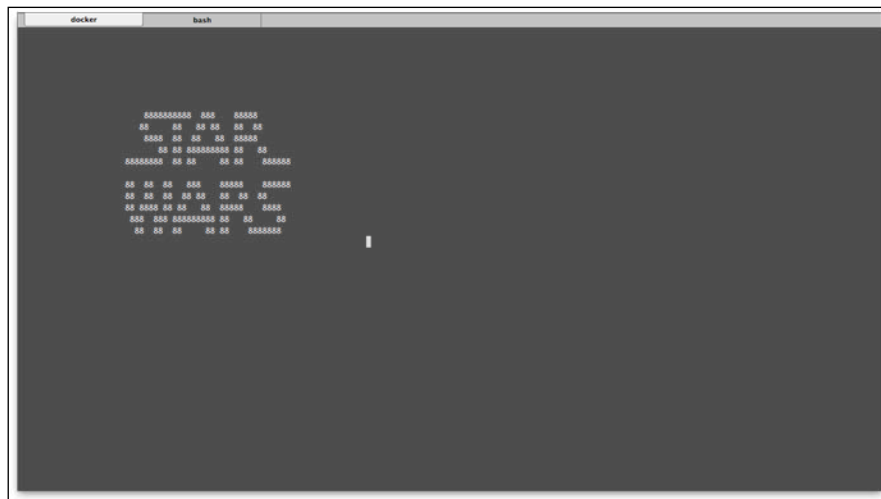
Save this in a folder named `star_wars` and open your terminal at this location. Then run this command:

```
$ docker build -t starwars .
```

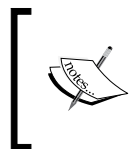
Now you can run it using the following command:

```
$ docker run -it starwars
```

The following screenshot shows the `starwars` output:



Thus, you can watch **Star Wars** in your terminal!



This *Star Wars* tribute was created by Simon Jansen, Sten Spans, and Mike Edwards. When you've had enough, hold *Ctrl + J*. You will be given a prompt where you can type *close* to exit.

## The ENTRYPOINT instruction

The `ENTRYPOINT` instruction allows you to turn your Docker image into an executable. In other words, when you specify an executable in an `ENTRYPOINT`, containers will run as if it was just that executable.

The `ENTRYPOINT` instruction has two forms:

1. The `ENTRYPOINT ["executable", "arg1", "arg2"...]` form.
2. The `ENTRYPOINT command arg1 arg2 ...` form.

This instruction adds an entry command that will not be overridden when arguments are passed to the `docker run` command, unlike the behavior of the `CMD` instruction. This allows arguments to be passed to the `ENTRYPOINT` instruction. The `docker run <image> -arg` command will pass the `-arg` argument to the command specified in the `ENTRYPOINT` instruction.

Parameters, if specified in the `ENTRYPOINT` instruction, will not be overridden by the `docker run` arguments, but parameters specified via the `CMD` instruction will be overridden.

As an example, let's write a Dockerfile with `cowsay` as the `ENTRYPOINT` instruction:



The `cowsay` is a program that generates ASCII pictures of a cow with a message. It can also generate pictures using premade images of other animals, such as Tux the Penguin, the Linux mascot.

```
FROM ubuntu:14.04
RUN apt-get -y install cowsay
ENTRYPOINT ["/usr/games/cowsay"]
CMD ["Docker is so awesomooooooooo!"]
```

Save this with the name `Dockerfile` in a folder named `cowsay`. Then through terminal, go to that directory, and run this command:

```
$ docker build -t cowsay .
```

Once the image is built, run the following command:

```
$ docker run cowsay
```

The following screenshot shows the output of the preceding command:

```
FDLMC219-MacBook-Pro:cowsay shrikrishna$ docker run shrikrishna/cowsay
< Docker is so awesomooooooooo! >
      /\
     /  \
    (oo)\_____.
    (__)\       )\/\
       ||----w |
       ||     ||

FDLMC219-MacBook-Pro:cowsay shrikrishna$ docker run shrikrishna/cowsay -f tux "This book is great tooooooo"
< This book is great tooooooo >
      /\
     /  \
    (oo)\_____.
    (__)\       )\/\
       ||----w |
       ||     ||
```

If you look at the screenshot closely, the first run has no arguments and it used the argument we configured in the Dockerfile. However, when we gave our own arguments in the second run, it overrode the default and passed all the arguments (The `-f` flag and the sentence) to the `cowsay` folder.



If you are the kind who likes to troll others, here's a tip: apply the instructions given at <http://superuser.com/a/175802> to set up a pre-exec script (a function that is called whenever a command is executed) that passes every command to this Docker container, and place it in the `.bashrc` file. Now `cowsay` will print every command that it execute in a text balloon, being said by an ASCII cow!

## The WORKDIR instruction

The `WORKDIR` instruction sets the working directory for the `RUN`, `CMD`, and `ENTRYPOINT` Dockerfile commands that follow it:

```
WORKDIR /path/to/working/directory
```

This instruction can be used multiple times in the same Dockerfile. If a relative path is provided, the `WORKDIR` instruction will be relative to the path of the previous `WORKDIR` instruction.

## The EXPOSE instruction

The `EXPOSE` instruction informs Docker that a certain port is to be exposed when a container is started:

```
EXPOSE port1 port2 ...
```

Even after exposing ports, while starting a container, you still need to provide port mapping using the `-p` flag to `Docker run`. This instruction is useful when linking containers, which we will see in *Chapter 3, Linking Containers*.

## The ENV instruction

The `ENV` command is used to set environment variables:

```
ENV <key> <value>
```

This sets the `<key>` environment variable to `<value>`. This value will be passed to all future `RUN` instructions. This is equivalent to prefixing the command with `<key>=<value>`.

The environment variables set using the `ENV` command will persist. This means that when a container is run from the resulting image, the environment variable will be available to the running process as well. The `docker inspect` command shows the values that have been assigned during the creation of the image. However, these can be overridden using the `$ docker run -env <key>=<value>` command.

## The `USER` instruction

The `USER` instruction sets the username or UID to use when running the image and any following the `RUN` directives:

```
USER xyz
```

## The `VOLUME` instruction

The `VOLUME` instruction will create a mount point with the given name and mark it as holding externally mounted volumes from the host or from other containers:

```
VOLUME [path]
```

Here is an example of the `VOLUME` instruction:

```
VOLUME ["/data"]
```

Here is another example of this instruction:

```
VOLUME /var/log
```

Both formats are acceptable.

## The `ADD` instruction

The `ADD` instruction is used to copy files into the image:

```
ADD <src> <dest>
```

The `ADD` instruction will copy files from `<src>` into the path at `<dest>`.

The `<src>` path must be the path to a file or directory relative to the source directory being built (also called the context of the build) or a remote file URL.

The `<dest>` path is the absolute path to which the source will be copied inside the destination container.



If you build by passing a Dockerfile through the `stdin` file (`docker build - <somefile>`), there is no build context, so the Dockerfile can only contain a URL-based `ADD` statement. You can also pass a compressed archive through the `stdin` file (`docker build - <archive.tar.gz>`). Docker will look for a Dockerfile at the root of the archive and the rest of the archive will get used as the context of the build.

The `ADD` instruction obeys the following rules:

- The `<src>` path must be inside the context of the build. You cannot use `ADD ../file` as `..` syntax, as it is beyond the context.
- If `<src>` is a URL and the `<dest>` path doesn't end with a trailing slash (it's a file), then the file at the URL is copied to the `<dest>` path.
- If `<src>` is a URL and the `<dest>` path ends with a trailing slash (it's a directory), then the content at the URL is fetched and a filename is inferred from the URL and saved into the `<dest>/filename` path. So, the URL cannot have a simple path such as `example.com` in this case.
- If `<src>` is a directory, the entire directory is copied, along with the filesystem metadata.
- If `<src>` is a local tar archive, then it is extracted into the `<dest>` path. The result at `<dest>` is union of:
  - Whatever existed at the path `<dest>`.
  - Contents of the extracted tar archive, with conflicts in favor of the path `<src>`, on a file-by-file basis.
- If `<dest>` path doesn't exist, it is created along with all the missing directories along its path.

## The COPY instruction

The `COPY` instruction copies a file into the image:

```
COPY <src> <dest>
```

The `Copy` instruction is similar to the `ADD` instruction. The difference is that the `COPY` instruction does not allow any file out of the context. So, if you are streaming Dockerfile via the `stdin` file or a URL (which doesn't point to a source code repository), the `COPY` instruction cannot be used.

## The ONBUILD instruction

The ONBUILD instruction adds to the image a trigger that will be executed when the image is used as a base image for another build:

**ONBUILD [INSTRUCTION]**

This is useful when the source application involves generators that need to compile before they can be used. Any build instruction apart from the FROM, MAINTAINER, and ONBUILD instructions can be registered.

Here's how this instruction works:

1. During a build, if the ONBUILD instruction is encountered, it registers a trigger and adds it to the metadata of the image. The current build is not otherwise affected in any way.
2. A list of all such triggers is added to the image manifest as a key named OnBuild at the end of the build (which can be seen through the Docker inspect command).
3. When this image is later used as a base image for a new build, as part of processing the FROM instruction, the OnBuild key triggers are read and executed in the order they were registered. If any of them fails, the FROM instruction aborts, causing the build to fail. Otherwise, the FROM instruction completes and the build continues as usual.
4. Triggers are cleared from the final image after being executed. In other words they are not inherited by *grand-child builds*.

Let's bring cowsay back! Here's a Dockerfile with the ONBUILD instruction:

```
FROM ubuntu:14.04
RUN apt-get -y install cowsay
RUN apt-get -y install fortune
ENTRYPOINT ["/usr/games/cowsay"]
CMD ["Docker is so awesomooooooooo!"]
ONBUILD RUN /usr/games/fortune | /usr/games/cowsay
```

Now save this file in a folder named OnBuild, open a terminal in that folder, and run this command:

```
$ Docker build -t shrikrishna/onbuild .
```

We need to write another Dockerfile that builds on this image. Let's write one:

```
FROM shrikrishna/onbuild
```



```
RUN apt-get moo
CMD ['/usr/bin/apt-get', 'moo']
```



The `apt-get moo` command is an example of Easter eggs typically found in many open source tools, added just for the sake of fun!

Building this image will now execute the `ONBUILD` instruction we gave earlier:

```
$ docker build -t shrikrishna/apt-moo apt-moo/
Sending build context to Docker daemon 2.56 kB
Sending build context to Docker daemon
Step 0 : FROM shrikrishna/onbuild
# Executing 1 build triggers
Step onbuild-0 : RUN /usr/games/fortune | /usr/games/cowsay
---> Running in 887592730f3d
```

```
/ It was all so different before \
\ everything changed.              /
```

```
-----
      ^__^
      (oo)\_______
          (__)\       )\/\
              ||----w |
              ||     ||
```

```
---> df01e4caldc7
```

```
---> df01e4caldc7
```

```
Removing intermediate container 887592730f3d
```

```
Step 1 : RUN apt-get moo
---> Running in fc596cb91c2a
```

```
      (__ )
      (oo)
    /-----\
  /  |      |
 * / \---\
    ~ ~    ~ ~
```

```
..."Have you mooed today?"...
---> 623cd16a51a7
Removing intermediate container fc596cb91c2a
Step 2 : CMD ['/usr/bin/apt-get', 'moo']
---> Running in 22aa0b415af4
---> 7e03264fbb76
Removing intermediate container 22aa0b415af4
Successfully built 7e03264fbb76
```

Now let's use our newly gained knowledge to write a Dockerfile for the `code.it` application that we previously built by manually satisfying dependencies in a container and committing. The Dockerfile would look something like this:

```
# Version 1.0
FROM dockerfile/nodejs
MAINTAINER Shrikrishna Holla <s**a@gmail.com>

WORKDIR /home
RUN    git clone \ https://github.com/shrikrishnaholla/code.it.git

WORKDIR code.it
RUN    git submodule update --init --recursive
RUN    npm install

EXPOSE 8000

WORKDIR /home
CMD    ["/usr/bin/node", "/home/code.it/app.js"]
```

Create a folder named `code.it` and save this content as a file named `Dockerfile`.



It is good practice to create a separate folder for every Dockerfile even if there is no context needed. This allows you to separate concerns between different projects. You might notice as you go that many Dockerfile authors club `RUN` instructions (for example, check out the Dockerfiles in `dockerfile.github.io`). The reason is that AUFS limits the number of possible layers to 42. For more information, check out this issue at <https://github.com/docker/docker/issues/1171>.

You can go back to the section on *Docker build* to see how to build an image out of this Dockerfile.

## Docker workflow - pull-use-modify-commit-push

Now, as we are nearing the end of this chapter, we can guess what a typical Docker workflow is like:

1. Prepare a list of requirements to run your application.
2. Determine which public image (or one of your own) can satisfy most of these requirements, while also being well-maintained (this is important as you would need the image to be updated with newer versions whenever they are available).
3. Next, fulfill the remaining requirements either by running a container and executing the commands that fulfill the requirements (which can be installing dependencies, bind mounting volumes, or fetching your source code), or by writing a Dockerfile (which is preferable since you will be able to make the build repeatable).
4. Push your new image to the public Docker registry so that the community can use it too (or to a private registry or repository if needs be).