LEARNING Docker

#docker

Chapter 2: Building images

Parameters

Parameter	Details
--pull	Ensures that the base image (`FROM`) is up-to-date before building the rest of the Dockerfile.

Examples

Building an image from a Dockerfile

Once you have a Dockerfile, you can build an image from it using `docker build`. The basic form of this command is:

```
docker build -t image-name path
```

If your Dockerfile isn't named `Dockerfile`, you can use the `-f` flag to give the name of the Dockerfile to build.

```
docker build -t image-name -f Dockerfile2 .
```

For example, to build an image named `dockerbuild-example:1.0.0` from a `Dockerfile` in the current working directory:

```
$ ls
Dockerfile Dockerfile2

$ docker build -t dockerbuild-example:1.0.0 .

$ docker build -t dockerbuild-example-2:1.0.0 -f Dockerfile2 .
```

See the `docker build` usage documentation for more options and settings.

A common mistake is creating a Dockerfile in the user home directory (`~`). This is a bad idea because during `docker build -t mytag .` this message will appear for a long time:

> Uploading context

The cause is the docker daemon trying to copy all the user's files (both the home directory and it's subdirectories). Avoid this by always specifying a directory for the Dockerfile.

Adding a `.dockerignore` file to the build directory is a good practice. Its syntax is similar to `.gitignore` files and will make sure only wanted files and directories are uploaded as the context of the build.

A simple Dockerfile

```
FROM node:5
```

The `FROM` directive specifies an image to start from. Any valid image reference may be used.

```
WORKDIR /usr/src/app
```

The `WORKDIR` directive sets the current working directory inside the container, equivalent to running `cd` inside the container. (Note: `RUN cd` will *not* change the current working directory.)

```
RUN npm install cowsay knock-knock-jokes
```

`RUN` executes the given command inside the container.

```
COPY cowsay-knockknock.js ./
```

`COPY` copies the file or directory specified in the first argument from the build context (the *`path`* passed to `docker build` *`path`*) to the location in the container specified by the second argument.

```
CMD node cowsay-knockknock.js
```

`CMD` specifies a command to execute when the image is run and no command is given. It can be overridden by passing a command to `docker run`.

There are many other instructions and options; see the Dockerfile reference for a complete list.

Difference between ENTRYPOINT and CMD

There are two `Dockerfile` directives to specify what command to run by default in built images. If you only specify `CMD` then docker will run that command using the default `ENTRYPOINT`, which is `/bin/sh -c`. You can override either or both the entrypoint and/or the command when you start up the built image. If you specify both, then the `ENTRYPOINT` specifies the executable of your container process, and `CMD` will be supplied as the parameters of that executable.

For example if your `Dockerfile` contains

```
FROM ubuntu:16.04
CMD ["/bin/date"]
```

Then you are using the default `ENTRYPOINT` directive of `/bin/sh -c`, and running `/bin/date` with that default entrypoint. The command of your container process will be `/bin/sh -c /bin/date`. Once you run this image then it will by default print out the current date

```
$ docker build -t test .
$ docker run test
Tue Jul 19 10:37:43 UTC 2016
```

You can override `CMD` on the command line, in which case it will run the command you have specified.

```
$ docker run test /bin/hostname
bf0274ec8820
```

If you specify an `ENTRYPOINT` directive, Docker will use that executable, and the `CMD` directive specifies the default parameter(s) of the command. So if your `Dockerfile` contains:

```
FROM ubuntu:16.04
ENTRYPOINT ["/bin/echo"]
CMD ["Hello"]
```

Then running it will produce

```
$ docker build -t test .
$ docker run test
Hello
```

You can provide different parameters if you want to, but they will all run `/bin/echo`

```
$ docker run test Hi
Hi
```

If you want to override the entrypoint listed in your Dockerfile (i.e. if you wish to run a different command than `echo` in this container), then you need to specify the `--entrypoint` parameter on the command line:

```
$ docker run --entrypoint=/bin/hostname test
b2c70e74df18
```

Generally you use the `ENTRYPOINT` directive to point to your main application you want to run, and `CMD` to the default parameters.

Exposing a Port in the Dockerfile

```
EXPOSE <port> [<port>...]
```

From Docker's documentation:

> The `EXPOSE` instruction informs Docker that the container listens on the specified network ports at runtime. `EXPOSE` does not make the ports of the container accessible to the host. To do that, you must use either the `-p` flag to publish a range of ports or the `-P` flag to publish all of the exposed ports. You can expose one port number and publish it externally under another number.

Example:

Inside your Dockerfile:

```
EXPOSE 8765
```

To access this port from the host machine, include this argument in your `docker run` command:

```
-p 8765:8765
```

ENTRYPOINT and CMD seen as verb and parameter

Suppose you have a Dockerfile ending with

`ENTRYPOINT [ "nethogs"] CMD ["wlan0"]`

if you build this image with a

`docker built -t inspector .`

launch the image built with such a Dockerfile with a command such as

`docker run -it --net=host --rm inspector`

,nethogs will monitor the interface named wlan0

Now if you want to monitor the interface eth0 (or wlan1, or ra1...), you will do something like

`docker run -it --net=host --rm inspector eth0`

or

`docker run -it --net=host --rm inspector wlan1`

Pushing and Pulling an Image to Docker Hub or another Registry

Locally created images can be pushed to Docker Hub or any other docker repo host, known as a registry. Use `docker login` to sign in to an existing docker hub account.

```
docker login

Login with your Docker ID to push and pull images from Docker Hub.
If you don't have a Docker ID, head over to https://hub.docker.com to create one.

Username: cjsimon
Password:
Login Succeeded
```

A different docker registry can be used by specifying a server name. This also works for private or self-hosted registries. Further, using an external credentials store for safety is possible.

```
docker login quay.io
```

You can then tag and push images to the registry that you are logged in to. Your repository must be specified as `server/username/reponame:tag`. Omitting the server currently defaults to Docker Hub. (The default registry cannot be changed to another provider, and there are no plans to implement this feature.)

```
docker tag mynginx quay.io/cjsimon/mynginx:latest
```

Different tags can be used to represent different versions, or branches, of the same image. An image with multiple different tags will display each tag in the same repo.

Use `docker images` to see a list of installed images installed on your local machine, including your newly tagged image. Then use push to upload it to the registry and pull to download the image.

```
docker push quay.io/cjsimon/mynginx:latest
```

All tags of an images can be pulled by specifying the `-a` option

```
docker pull quay.io/cjsimon/mynginx:latest
```

Building using a proxy

Often when building a Docker image, the Dockerfile contains instructions that runs programs to fetch resources from the Internet (`wget` for example to pull a program binary build on GitHub for example).

It is possible to instruct Docker to pass set set environment variables so that such programs perform those fetches through a proxy:

```
$ docker build --build-arg http_proxy=http://myproxy.example.com:3128 \
               --build-arg https_proxy=http://myproxy.example.com:3128 \
               --build-arg no_proxy=internal.example.com \
               -t test .
```

`build-arg` are environment variables which are available at build time only.

Read Building images online: https://riptutorial.com/docker/topic/713/building-images

Chapter 7: Data Volumes and Data Containers

Examples

Data-Only Containers

Data-only containers are obsolete and are now considered an anti-pattern!

In the days of yore, before Docker's `volume` subcommand, and before it was possible to create named volumes, Docker deleted volumes when there were no more references to them in any containers. Data-only containers are obsolete because Docker now provides the ability to create named volumes, as well as much more utility via the various `docker volume` subcommand. Data-only containers are now considered an anti-pattern for this reason.

Many resources on the web from the last couple of years mention using a pattern called a "data-only container", which is simply a Docker container that exists only to keep a reference to a data volume around.

Remember that in this context, a "data volume" is a Docker volume which is not mounted from the host. To clarify, a "data volume" is a volume which is created either with the `VOLUME` Dockerfile directive, or using the `-v` switch on the command line in a `docker run` command, specifically with the format `-v /path/on/container`. Therefore a "data-only container" is a container whose only purpose is to have a data volume attached, which is used by the `--volumes-from` flag in a `docker run` command. For example:

```
docker run -d --name "mysql-data" -v "/var/lib/mysql" alpine /bin/true
```

When the above command is run, a "data-only container" is created. It is simply an empty container which has a data volume attached. It was then possible to use this volume in another container like so:

```
docker run -d --name="mysql" --volumes-from="mysql-data" mysql
```

The `mysql` container now has the same volume in it that is also in `mysql-data`.

Because Docker now provides the `volume` subcommand and named volumes, this pattern is now obsolete and not recommended.

To get started with the `volume` subcommand and named volumes see Creating a named volume

Creating a data volume

```
docker run -d --name "mysql-1" -v "/var/lib/mysql" mysql
```

This command creates a new container from the `mysql` image. It also creates a new data volume, which it then mounts in the container at `/var/lib/mysql`. This volume helps any data inside of it persist beyond the lifetime of the container. That is to say, when a container is removed, its filesystem changes are also removed. If a database was storing data in the container, and the container is removed, all of that data is also removed. Volumes will persist a particular location even beyond when its container is removed.

It is possible to use the same volume in multiple containers with the `--volumes-from` command line option:

```
docker run -d --name="mysql-2" --volumes-from="mysql-1" mysql
```

The `mysql-2` container now has the data volume from `mysql-1` attached to it, also using the path `/var/lib/mysql`.

Read Data Volumes and Data Containers online: https://riptutorial.com/docker/topic/3224/data-volumes-and-data-containers

Chapter 8: Debugging a container

Syntax

- docker stats [OPTIONS] [CONTAINER...]
- docker logs [OPTIONS] CONTAINER
- docker top [OPTIONS] CONTAINER [ps OPTIONS]

Examples

Entering in a running container

To execute operations in a container, use the `docker exec` command. Sometimes this is called "entering the container" as all commands are executed inside the container.

```
docker exec -it container_id bash
```

or

```
docker exec -it container_id /bin/sh
```

And now you have a shell in your running container. For example, list files in a directory and then leave the container:

```
docker exec container_id ls -la
```

You can use the `-u flag` to enter the container with a specific user, e.g. `uid=1013`, `gid=1023`.

```
docker exec -it -u 1013:1023 container_id ls -la
```

The uid and gid does not have to exist in the container but the command can result in errors.If you want to launch a container and immediately enter inside in order to check something, you can do

`docker run...; docker exec -it $(docker ps -lq) bash`

the command `docker ps -lq` outputs only the id of the last (the l in `-lq`) container started. (this supposes you have bash as interpreter available in your container, you may have sh or zsh or any other)

Monitoring resource usage

Inspecting system resource usage is an efficient way to find misbehaving applications. This example is an equivalent of the traditional `top` command for containers:

```
docker stats
```

To follow the stats of specific containers, list them on the command line:

```
docker stats 7786807d8084 7786807d8085
```

Docker stats displays the following information:

```
CONTAINER     CPU %   MEM USAGE / LIMIT   MEM %    NET I/O               BLOCK I/O
7786807d8084  0.65%   1.33 GB / 3.95 GB   33.67%   142.2 MB / 57.79 MB   46.32 MB / 0 B
```

By default `docker stats` displays the id of the containers, and this is not very helpful, if your prefer to display the names of the container, just do

```
docker stats $(docker ps --format '{{.Names}}')
```

Monitoring processes in a container

Inspecting system resource usage is an efficient way to narrow down a problem on a live running application. This example is an equivalent of the traditional `ps` command for containers.

```
docker top 7786807d8084
```

To filter of format the output, add `ps` options on the command line:

```
docker top 7786807d8084 faux
```

Or, to get the list of processes running as root, which is a potentially harmful practice:

```
docker top 7786807d8084 -u root
```

The `docker top` command proves especially useful when troubleshooting minimalistic containers without a shell or the `ps` command.

Attach to a running container

'Attaching to a container' is the act of starting a terminal session within the context that the container (and any programs therein) is running. This is primarily used for debugging purposes, but may also be needed if specific data needs to be passed to programs running within the container.

The `attach` command is utilized to do this. It has this syntax:

```
docker attach <container>
```

`<container>` can be either the container id or the container name. For instance:

```
docker attach c8a9cf1a1fa8
```

Or:

```
docker attach graceful_hopper
```

You may need to `sudo` the above commands, depending on your user and how docker is set up.

> Note: Attach only allows a single shell session to be attached to a container at a time.

> Warning: *all* keyboard input will be forwarded to the container. Hitting `Ctrl-c` will *kill* your container.

To detach from an attached container, successively hit `Ctrl-p` then `Ctrl-q`

To attach multiple shell sessions to a container, or simply as an alternative, you can use `exec`. Using the container id:

```
docker exec -i -t c8a9cf1a1fa8 /bin/bash
```

Using the container's name:

```
docker exec -i -t graceful_hopper /bin/bash
```

`exec` will run a program within a container, in this case `/bin/bash` (a shell, presumably one the container has). `-i` indicates an interactive session, while `-t` allocates a pseudo-TTY.

> Note: Unlike *attach*, hitting `Ctrl-c` will only terminate the *exec*'d command when running interactively.

Printing the logs

Following the logs is the less intrusive way to debug a live running application. This example reproduces the behavior of the traditional `tail -f some-application.log` on container `7786807d8084`.

```
docker logs --follow --tail 10 7786807d8084
```

This command basically shows the standard output of the container process (the process with pid 1).

If your logs do not natively include timestamping, you may add the `--timestamps` flag.

It is possible to look at the logs of a stopped container, either

- start the failing container with `docker run ... ; docker logs $(docker ps -lq)`

- find the container id or name with

```
docker ps -a
```

and then

`docker logs container-id` or

`docker logs containername`

as it is possible to look at the logs of a stopped container

Docker container process debugging

Docker is just a fancy way to run a process, not a virtual machine. Therefore, debugging a process "in a container" is also possible "on the host" by simply examining the running container process as a user with the appropriate permissions to inspect those processes on the host (e.g. root). For example, it's possible to list every "container process" on the host by running a simple `ps` as root:

```
sudo ps aux
```

Any currently running Docker containers will be listed in the output.

This can be useful during application development for debugging a process running in a container. As a user with appropriate permissions, typical debugging utilities can be used on the container process, such as strace, ltrace, gdb, etc.

Read Debugging a container online: https://riptutorial.com/docker/topic/1333/debugging-a-container

Chapter 9: Docker Data Volumes

Introduction

Docker data volumes provide a way to persist data independent of a container's life cycle. Volumes present a number of helpful features such as:

Mounting a host directory within the container, sharing data in-between containers using the filesystem and preserving data if a container gets deleted

Syntax

- docker volume [OPTIONS] [COMMAND]

Examples

Mounting a directory from the local host into a container

It is possible to mount a host directory to a specific path in your container using the `-v` or `--volume` command line option. The following example will mount `/etc` on the host to `/mnt/etc` in the container:

```
(on linux) docker run -v "/etc:/mnt/etc" alpine cat /mnt/etc/passwd
(on windows)  docker run -v "/c/etc:/mnt/etc" alpine cat /mnt/etc/passwd
```

The default access to the volume inside the container is read-write. To mount a volume read-only inside of a container, use the suffix `:ro`:

```
docker run -v "/etc:/mnt/etc:ro" alpine touch /mnt/etc/passwd
```

Creating a named volume

```
docker volume create --name="myAwesomeApp"
```

Using a named volume makes managing volumes much more human-readable. It is possible to create a named volume using the command specified above, but it's also possible to create a named volume inside of a `docker run` command using the `-v` or `--volume` command line option:

```
docker run -d --name="myApp-1" -v="myAwesomeApp:/data/app" myApp:1.5.3
```

Note that creating a named volume in this form is similar to mounting a host file/directory as a volume, except that instead of a valid path, the volume name is specified. Once created, named volumes can be shared with other containers:

```
docker run -d --name="myApp-2" --volumes-from "myApp-1" myApp:1.5.3
```

After running the above command, a new container has been created with the name `myApp-2` from the `myApp:1.5.3` image, which is sharing the `myAwesomeApp` named volume with `myApp-1`. The `myAwesomeApp` named volume is mounted at `/data/app` in the `myApp-2` container, just as it is mounted at `/data/app` in the `myApp-1` container.

Read Docker Data Volumes online: https://riptutorial.com/docker/topic/1318/docker-data-volumes

Chapter 16: Docker network

Examples

How to find the Container's host ip

You need to find out the IP address of the container running in the host so you can, for example, connect to the web server running in it.

`docker-machine` is what is used on MacOSX and Windows.

Firstly, list your machines:

```
$ docker-machine ls

NAME      ACTIVE   DRIVER       STATE     URL                         SWARM
default   *        virtualbox   Running   tcp://192.168.99.100:2376
```

Then select one of the machines (the default one is called default) and:

```
$ docker-machine ip default

192.168.99.100
```

Creating a Docker network

```
docker network create app-backend
```

This command will create a simple bridged network called `appBackend`. No containers are attached to this network by default.

Listing Networks

```
docker network ls
```

This command lists all networks that have been created on the local Docker host. It includes the default bridge `bridge` network, the host `host` network, and the null `null` network. All containers by default are attached to the default bridge `bridge` network.

Add container to network

```
docker network connect app-backend myAwesomeApp-1
```

This command attaches the `myAwesomeApp-1` container to the `app-backend` network. When you add a container to a user-defined network, the embedded DNS resolver (which is not a full-featured DNS server, and is not exportable) allows each container on the network to resolve each other container on the same network. This simple DNS resolver is not available on the default bridge `bridge` network.

Detach container from network

```
docker network disconnect app-backend myAwesomeApp-1
```

This command detaches the `myAwesomeApp-1` container from the `app-backend` network. The container will no longer be able to communicate with other containers on the network it has been disconnected from, nor use the embedded DNS resolver to look up other containers on the network it has been detached from.

Remove a Docker network

```
docker network rm app-backend
```

This command removes the user-defined `app-backend` network from the Docker host. All containers on the network not otherwise connected via another network will lose communication with other containers. It is not possible to remove the default bridge `bridge` network, the `host` host network, or the `null` null network.

Inspect a Docker network

```
docker network inspect app-backend
```

This command will output details about the `app-backend` network.

The of the output of this command should look similar to:

```
[
    {
        "Name": "foo",
        "Id": "a0349d78c8fd7c16f5940bdbaf1adec8d8399b8309b2e8a969bd4e3226a6fc58",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1/16"
                }
            ]
        },
        "Internal": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
        }
]
```

Read Docker network online: https://riptutorial.com/docker/topic/3221/docker-network

Chapter 22: Dockerfiles

Introduction

Dockerfiles are files used to programatically build Docker images. They allow you to quickly and reproducibly create a Docker image, and so are useful for collaborating. Dockerfiles contain instructions for building a Docker image. Each instruction is written on one row, and is given in the form `<INSTRUCTION><argument(s)>`. Dockerfiles are used to build Docker images using the `docker build` command.

Remarks

Dockerfiles are of the form:

```
# This is a comment
INSTRUCTION arguments
```

- Comments starts with a `#`
- Instructions are upper case only
- The first instruction of a Dockerfile must be `FROM` to specify the base image

---

When building a Dockerfile, the Docker client will send a "build context" to the Docker daemon. The build context includes all files and folder in the same directory as the Dockerfile. `COPY` and `ADD` operations can only use files from this context.

Some Docker file may start with:

```
# escape=`
```

This is used to instruct the Docker parser to use `` ` `` as an escape character instead of `\`. This is mostly useful for Windows Docker files.

Examples

HelloWorld Dockerfile

A minimal Dockerfile looks like this:

```
FROM alpine
CMD ["echo", "Hello StackOverflow!"]
```

This will instruct Docker to build an image based on Alpine (`FROM`), a minimal distribution for containers, and to run a specific command (`CMD`) when executing the resulting image.

Build and run it:

```
docker build -t hello .
docker run --rm hello
```

This will output:

```
Hello StackOverflow!
```

Copying files

To copy files from the build context in a Docker image, use the `COPY` instruction:

```
COPY localfile.txt containerfile.txt
```

If the filename contains spaces, use the alternate syntax:

```
COPY ["local file", "container file"]
```

The `COPY` command supports wildcards. It can be used for example to copy all images to the `images/` directory:

```
COPY *.jpg images/
```

Note: in this example, `images/` may not exist. In this case, Docker will create it automatically.

Exposing a port

To declare exposed ports from a Dockerfile use the `EXPOSE` instruction:

```
EXPOSE 8080 8082
```

Exposed ports setting can be overridden from the Docker commandline but it is a good practice to explicitly set them in the Dockerfile as it helps understand what an application does.

Dockerfiles best pratices

Group common operations

Docker builds images as a collection of layers. Each layer can only add data, even if this data says that a file has been deleted. Every instruction creates a new layer. For example:

```
RUN apt-get -qq update
RUN apt-get -qq install some-package
```

Has a couple of downsides:

- It will create two layers, producing a larger image.
- Using `apt-get update` alone in a `RUN` statement causes caching issues and subsequently `apt-get install` instructions may **fail**. Suppose you later modify `apt-get install` by adding extra packages, then docker interprets the initial and modified instructions as identical and reuses the cache from previous steps. As a result the `apt-get update` command is **not** executed because its cached version is used during the build.

Instead, use:

```
RUN apt-get -qq update && \
    apt-get -qq install some-package
```

as this only produce one layer.

Mention the maintainer

This is usually the second line of the Dockerfile. It tells who is in charge and will be able to help.

```
LABEL maintainer John Doe <john.doe@example.com>
```

If you skip it, it will not break your image. But it will not help your users either.

Be concise

Keep your Dockerfile short. If a complex setup is necessary, consider using a dedicated script or setting up base images.

USER Instruction

```
USER daemon
```

The `USER` instruction sets the user name or UID to use when running the image and for any `RUN`, `CMD` and `ENTRYPOINT` instructions that follow it in the `Dockerfile`.

WORKDIR Instruction

```
WORKDIR /path/to/workdir
```

The `WORKDIR` instruction sets the working directory for any `RUN`, `CMD`, `ENTRYPOINT`, `COPY` and `ADD` instructions that follow it in the Dockerfile. If the `WORKDIR` doesn't exist, it will be created even if it's not used in any subsequent `Dockerfile` instruction.

It can be used multiple times in the one `Dockerfile`. If a relative path is provided, it will be relative to the path of the previous `WORKDIR` instruction. For example:

```
WORKDIR /a
WORKDIR b
WORKDIR c
```

```
RUN pwd
```

The output of the final `pwd` command in this `Dockerfile` would be `/a/b/c`.

The `WORKDIR` instruction can resolve environment variables previously set using `ENV`. You can only use environment variables explicitly set in the `Dockerfile`. For example:

```
ENV DIRPATH /path
WORKDIR $DIRPATH/$DIRNAME
RUN pwd
```

The output of the final `pwd` command in this Dockerfile would be `/path/$DIRNAME`

VOLUME Instruction

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

The `VOLUME` instruction creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers. The value can be a JSON array, `VOLUME ["/var/log/"]`, or a plain string with multiple arguments, such as `VOLUME /var/log` or `VOLUME /var/log /var/db`. For more information/examples and mounting instructions via the Docker client, refer to Share Directories via Volumes documentation.

The `docker run` command initializes the newly created volume with any data that exists at the specified location within the base image. For example, consider the following Dockerfile snippet:

```
FROM ubuntu
RUN mkdir /myvol
RUN echo "hello world" > /myvol/greeting
VOLUME /myvol
```

This Dockerfile results in an image that causes docker run, to create a new mount point at /myvol and copy the greeting file into the newly created volume.

Note: If any build steps change the data within the volume after it has been declared, those changes will be discarded.

Note: The list is parsed as a JSON array, which means that you must use double-quotes (“) around words not single-quotes (‘).

COPY Instruction

`COPY` has two forms:

```
COPY <src>... <dest>
COPY ["<src>",... "<dest>"] (this form is required for paths containing whitespace)
```

The `COPY` instruction copies new files or directories from `<src>` and adds them to the filesystem of the container at the path `<dest>`.

Multiple `<src>` resource may be specified but they must be relative to the source directory that is being built (the context of the build).

Each `<src>` may contain wildcards and matching will be done using Go's `filepath.Match` rules. For example:

```
COPY hom* /mydir/        # adds all files starting with "hom"
COPY hom?.txt /mydir/    # ? is replaced with any single character, e.g., "home.txt"
```

The `<dest>` is an absolute path, or a path relative to `WORKDIR`, into which the source will be copied inside the destination container.

```
COPY test relativeDir/   # adds "test" to `WORKDIR`/relativeDir/
COPY test /absoluteDir/  # adds "test" to /absoluteDir/
```

All new files and directories are created with a UID and GID of 0.

Note: If you build using stdin (`docker build - < somefile`), there is no build context, so `COPY` can't be used.

`COPY` obeys the following rules:

- The `<src>` path must be inside the context of the build; you cannot `COPY` ../something /something, because the first step of a docker build is to send the context directory (and subdirectories) to the docker daemon.
- If `<src>` is a directory, the entire contents of the directory are copied, including filesystem metadata. Note: The directory itself is not copied, just its contents.
- If `<src>` is any other kind of file, it is copied individually along with its metadata. In this case, if `<dest>` ends with a trailing slash /, it will be considered a directory and the contents of `<src>` will be written at `<dest>/base(<src>)`.
- If multiple `<src>` resources are specified, either directly or due to the use of a wildcard, then `<dest>` must be a directory, and it must end with a slash `/`.
- If `<dest>` does not end with a trailing slash, it will be considered a regular file and the contents of `<src>` will be written at `<dest>`.
- If `<dest>` doesn't exist, it is created along with all missing directories in its path.

The ENV and ARG Instruction

ENV

```
ENV <key> <value>
ENV <key>=<value> ...
```

The `ENV` instruction sets the environment variable `<key>` to the value . This value will be in the environment of all “descendant” Dockerfile commands and can be replaced inline in many as well.

The `ENV` instruction has two forms. The first form, `ENV <key> <value>`, will set a single variable to a value. The entire string after the first space will be treated as the `<value>` - including characters such as spaces and quotes.

The second form, `ENV <key>=<value> ...`, allows for multiple variables to be set at one time. Notice that the second form uses the equals sign (=) in the syntax, while the first form does not. Like command line parsing, quotes and backslashes can be used to include spaces within values.

For example:

```
ENV myName="John Doe" myDog=Rex\ The\ Dog \
    myCat=fluffy
```

and

```
ENV myName John Doe
ENV myDog Rex The Dog
ENV myCat fluffy
```

will yield the same net results in the final container, but the first form is preferred because it produces a single cache layer.

The environment variables set using `ENV` will persist when a container is run from the resulting image. You can view the values using docker inspect, and change them using `docker run --env <key>=<value>`.

ARG

If you don't wish to persist the setting, use `ARG` instead. `ARG` will set environments only during the build. For example, setting

```
ENV DEBIAN_FRONTEND noninteractive
```

may confuse `apt-get` users on a Debian-based image when they enter the container in an interactive context via `docker exec -it the-container bash`.

Instead, use:

```
ARG DEBIAN_FRONTEND noninteractive
```

You might alternativly also set a value for a single command only by using:

```
RUN <key>=<value> <command>
```

EXPOSE Instruction

```
EXPOSE <port> [<port>...]
```

The `EXPOSE` instruction informs Docker that the container listens on the specified network ports at runtime. `EXPOSE` does NOT make the ports of the container accessible to the host. To do that, you must use either the `-p` flag to publish a range of ports or the `-P` flag to publish all of the exposed ports. These flags are used in the `docker run [OPTIONS] IMAGE [COMMAND][ARG...]` to expose the port to the host. You can expose one port number and publish it externally under another number.

```
docker run -p 2500:80 <image name>
```

This command will create a container with the name <image> and bind the container's port 80 to the host machine's port 2500.

To set up port redirection on the host system, see using the `-P` flag. The Docker network feature supports creating networks without the need to expose ports within the network, for detailed information see the overview of this feature).

LABEL Instruction

```
LABEL <key>=<value> <key>=<value> <key>=<value> ...
```

The `LABEL` instruction adds metadata to an image. A `LABEL` is a key-value pair. To include spaces within a `LABEL` value, use quotes and backslashes as you would in command-line parsing. A few usage examples:

```
LABEL "com.example.vendor"="ACME Incorporated"
LABEL com.example.label-with-value="foo"
LABEL version="1.0"
LABEL description="This text illustrates \
that label-values can span multiple lines."
```

An image can have more than one label. To specify multiple labels, Docker recommends combining labels into a single `LABEL` instruction where possible. Each `LABEL` instruction produces a new layer which can result in an inefficient image if you use many labels. This example results in a single image layer.

```
LABEL multi.label1="value1" multi.label2="value2" other="value3"
```

The above can also be written as:

```
LABEL multi.label1="value1" \
      multi.label2="value2" \
      other="value3"
```

Labels are additive including `LABEL`s in `FROM` images. If Docker encounters a label/key that already exists, the new value overrides any previous labels with identical keys.

To view an image's labels, use the docker inspect command.

```
"Labels": {
    "com.example.vendor": "ACME Incorporated"
    "com.example.label-with-value": "foo",
    "version": "1.0",
    "description": "This text illustrates that label-values can span multiple lines.",
    "multi.label1": "value1",
    "multi.label2": "value2",
    "other": "value3"
},
```

CMD Instruction

The `CMD` instruction has three forms:

```
CMD ["executable","param1","param2"] (exec form, this is the preferred form)
CMD ["param1","param2"] (as default parameters to ENTRYPOINT)
CMD command param1 param2 (shell form)
```

There can only be one `CMD` instruction in a `Dockerfile`. If you list more than one `CMD` then only the last `CMD` will take effect.

The main purpose of a `CMD` is to provide defaults for an executing container. These defaults can include an executable, or they can omit the executable, in which case you must specify an `ENTRYPOINT` instruction as well.

Note: If `CMD` is used to provide default arguments for the `ENTRYPOINT` instruction, both the `CMD` and `ENTRYPOINT` instructions should be specified with the JSON array format.

Note: The exec form is parsed as a JSON array, which means that you must use double-quotes (“) around words not single-quotes (‘).

Note: Unlike the shell form, the exec form does not invoke a command shell. This means that normal shell processing does not happen. For example, `CMD [ "echo", "$HOME" ]` will not do variable substitution on `$HOME`. If you want shell processing then either use the shell form or execute a shell directly, for example: `CMD [ "sh", "-c", "echo $HOME" ]`.

When used in the shell or exec formats, the `CMD` instruction sets the command to be executed when running the image.

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

```
FROM ubuntu
CMD echo "This is a test." | wc -
```

If you want to run your command without a shell then you must express the command as a JSON array and give the full path to the executable. This array form is the preferred format of `CMD`. Any additional parameters must be individually expressed as strings in the array:

```
FROM ubuntu
CMD ["/usr/bin/wc","--help"]
```

If you would like your container to run the same executable every time, then you should consider using `ENTRYPOINT` in combination with `CMD`. See `ENTRYPOINT`.

If the user specifies arguments to docker run then they will override the default specified in `CMD`.

Note: don't confuse `RUN` with `CMD`. `RUN` actually runs a command at image building time and commits the result; `CMD` does not execute anything at build time, but specifies the intended command for the image.

MAINTAINER Instruction

```
MAINTAINER <name>
```

The `MAINTAINER` instruction allows you to set the Author field of the generated images.

DO NOT USE THE MAINTAINER DIRECTIVE

According to Official Docker Documentation the `MAINTAINER` instruction is deprecated. Instead, one should use the `LABEL` instruction to define the author of the generated images. The `LABEL` instruction is more flexible, enables setting metadata, and can be easily viewed with the command `docker inspect`.

```
LABEL maintainer="someone@something.com"
```

FROM Instruction

```
FROM <image>
```

Or

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

Or

```
FROM <image>@<digest>
```

The `FROM` instruction sets the Base Image for subsequent instructions. As such, a valid Dockerfile must have `FROM` as its first instruction. The image can be any valid image – it is especially easy to start by pulling an image from the Public Repositories.

`FROM` must be the first non-comment instruction in the Dockerfile.

`FROM` can appear multiple times within a single Dockerfile in order to create multiple images. Simply make a note of the last image ID output by the commit before each new `FROM` command.

The tag or digest values are optional. If you omit either of them, the builder assumes a latest by default. The builder returns an error if it cannot match the tag value.

RUN Instruction

`RUN` has 2 forms:

```
RUN <command> (shell form, the command is run in a shell, which by default is /bin/sh -c on
Linux or cmd /S /C on Windows)
RUN ["executable", "param1", "param2"] (exec form)
```

The `RUN` instruction will execute any commands in a new layer on top of the current image and commit the results. The resulting committed image will be used for the next step in the `Dockerfile`.

Layering `RUN` instructions and generating commits conforms to the core concepts of Docker where commits are cheap and containers can be created from any point in an image's history, much like source control.

The exec form makes it possible to avoid shell string munging, and to `RUN` commands using a base image that does not contain the specified shell executable.

The default shell for the shell form can be changed using the `SHELL` command.

In the shell form you can use a `\` (backslash) to continue a single `RUN` instruction onto the next line. For example, consider these two lines:

```
RUN /bin/bash -c 'source $HOME/.bashrc ;\
echo $HOME'
```

Together they are equivalent to this single line:

```
RUN /bin/bash -c 'source $HOME/.bashrc ; echo $HOME'
```

Note: To use a different shell, other than '/bin/sh', use the exec form passing in the desired shell. For example, `RUN ["/bin/bash", "-c", "echo hello"]`

Note: The exec form is parsed as a JSON array, which means that you must use double-quotes (`"`) around words not single-quotes (`'`).

Note: Unlike the shell form, the exec form does not invoke a command shell. This means that normal shell processing does not happen. For example, `RUN [ "echo", "$HOME" ]` will not do variable substitution on `$HOME`. If you want shell processing then either use the shell form or execute a shell directly, for example: `RUN [ "sh", "-c", "echo $HOME" ]`.

Note: In the JSON form, it is necessary to escape backslashes. This is particularly relevant on Windows where the backslash is the path separator. The following line would otherwise be treated as shell form due to not being valid JSON, and fail in an unexpected way: `RUN ["c:\windows\system32\tasklist.exe"]`

The correct syntax for this example is: `RUN ["c:\\windows\\system32\\tasklist.exe"]`

The cache for `RUN` instructions isn't invalidated automatically during the next build. The cache for an instruction like `RUN apt-get dist-upgrade -y` will be reused during the next build. The cache for `RUN` instructions can be invalidated by using the --no-cache flag, for example docker build --no-cache.

See the Dockerfile Best Practices guide for more information.

The cache for `RUN` instructions can be invalidated by `ADD` instructions. See below for details.

ONBUILD Instruction

```
ONBUILD [INSTRUCTION]
```

The `ONBUILD` instruction adds to the image a trigger instruction to be executed at a later time, when the image is used as the base for another build. The trigger will be executed in the context of the downstream build, as if it had been inserted immediately after the `FROM` instruction in the downstream Dockerfile.

Any build instruction can be registered as a trigger.

This is useful if you are building an image which will be used as a base to build other images, for example an application build environment or a daemon which may be customized with user-specific configuration.

For example, if your image is a reusable Python application builder, it will require application source code to be added in a particular directory, and it might require a build script to be called after that. You can't just call `ADD` and `RUN` now, because you don't yet have access to the application source code, and it will be different for each application build. You could simply provide application developers with a boilerplate Dockerfile to copy-paste into their application, but that is inefficient, error-prone and difficult to update because it mixes with application-specific code.

The solution is to use `ONBUILD` to register advance instructions to run later, during the next build stage.

Here's how it works:

When it encounters an `ONBUILD` instruction, the builder adds a trigger to the metadata of the image being built. The instruction does not otherwise affect the current build.

At the end of the build, a list of all triggers is stored in the image manifest, under the key OnBuild. They can be inspected with the `docker inspect` command. Later the image may be used as a base for a new build, using the `FROM` instruction. As part of processing the `FROM` instruction, the downstream builder looks for `ONBUILD` triggers, and executes them in the same order they were registered. If any of the triggers fail, the `FROM` instruction is aborted which in turn causes the build to fail. If all triggers succeed, the `FROM` instruction completes and the build continues as usual.

Triggers are cleared from the final image after being executed. In other words they are not inherited by "grand-children" builds.

For example you might add something like this:

```
[...]
ONBUILD ADD . /app/src
ONBUILD RUN /usr/local/bin/python-build --dir /app/src
[...]
```

Warning: Chaining `ONBUILD` instructions using `ONBUILD ONBUILD` isn't allowed.

Warning: The `ONBUILD` instruction may not trigger `FROM` or `MAINTAINER` instructions.

STOPSIGNAL Instruction

```
STOPSIGNAL signal
```

The `STOPSIGNAL` instruction sets the system call signal that will be sent to the container to exit. This signal can be a valid unsigned number that matches a position in the kernel's syscall table, for instance 9, or a signal name in the format SIGNAME, for instance SIGKILL.

HEALTHCHECK Instruction

The `HEALTHCHECK` instruction has two forms:

```
HEALTHCHECK [OPTIONS] CMD command (check container health by running a command inside the
container)
HEALTHCHECK NONE (disable any healthcheck inherited from the base image)
```

The `HEALTHCHECK` instruction tells Docker how to test a container to check that it is still working. This can detect cases such as a web server that is stuck in an infinite loop and unable to handle new connections, even though the server process is still running.

When a container has a healthcheck specified, it has a health status in addition to its normal status. This status is initially starting. Whenever a health check passes, it becomes healthy (whatever state it was previously in). After a certain number of consecutive failures, it becomes unhealthy.

The options that can appear before `CMD` are:

```
--interval=DURATION (default: 30s)
--timeout=DURATION (default: 30s)
--retries=N (default: 3)
```

The health check will first run interval seconds after the container is started, and then again interval seconds after each previous check completes.

If a single run of the check takes longer than timeout seconds then the check is considered to have failed.

It takes retries consecutive failures of the health check for the container to be considered unhealthy.

There can only be one `HEALTHCHECK` instruction in a `Dockerfile`. If you list more than one then only the last `HEALTHCHECK` will take effect.

The command after the `CMD` keyword can be either a shell command (e.g. `HEALTHCHECK CMD /bin/check-running`) or an exec array (as with other Dockerfile commands; see e.g. `ENTRYPOINT` for details).

The command's exit status indicates the health status of the container. The possible values are:

- `0: success` - the container is healthy and ready for use
- `1: unhealthy` - the container is not working correctly
- `2: starting` - the container is not ready for use yet, but is working correctly

If the probe returns 2 ("starting") when the container has already moved out of the "starting" state then it is treated as "unhealthy" instead.

For example, to check every five minutes or so that a web-server is able to serve the site's main page within three seconds:

```
HEALTHCHECK --interval=5m --timeout=3s \
  CMD curl -f http://localhost/ || exit 1
```

To help debug failing probes, any output text (UTF-8 encoded) that the command writes on stdout or stderr will be stored in the health status and can be queried with `docker inspect`. Such output should be kept short (only the first 4096 bytes are stored currently).

When the health status of a container changes, a `health_status` event is generated with the new status.

The `HEALTHCHECK` feature was added in Docker 1.12.

SHELL Instruction

```
SHELL ["executable", "parameters"]
```

The `SHELL` instruction allows the default shell used for the shell form of commands to be overridden. The default shell on Linux is `["/bin/sh", "-c"]`, and on Windows is `["cmd", "/S", "/C"]`. The `SHELL` instruction must be written in JSON form in a Dockerfile.

The `SHELL` instruction is particularly useful on Windows where there are two commonly used and quite different native shells: cmd and powershell, as well as alternate shells available including sh.

The `SHELL` instruction can appear multiple times. Each `SHELL` instruction overrides all previous `SHELL` instructions, and affects all subsequent instructions. For example:

```
FROM windowsservercore

# Executed as cmd /S /C echo default
RUN echo default
```

```
# Executed as cmd /S /C powershell -command Write-Host default
RUN powershell -command Write-Host default

# Executed as powershell -command Write-Host hello
SHELL ["powershell", "-command"]
RUN Write-Host hello

# Executed as cmd /S /C echo hello
SHELL ["cmd", "/S"", "/C"]
RUN echo hello
```

The following instructions can be affected by the `SHELL` instruction when the shell form of them is used in a Dockerfile: `RUN`, `CMD` and `ENTRYPOINT`.

The following example is a common pattern found on Windows which can be streamlined by using the `SHELL` instruction:

```
...
RUN powershell -command Execute-MyCmdlet -param1 "c:\foo.txt"
...
```

The command invoked by docker will be:

```
cmd /S /C powershell -command Execute-MyCmdlet -param1 "c:\foo.txt"
```

This is inefficient for two reasons. First, there is an un-necessary cmd.exe command processor (aka shell) being invoked. Second, each `RUN` instruction in the shell form requires an extra powershell -command prefixing the command.

To make this more efficient, one of two mechanisms can be employed. One is to use the JSON form of the `RUN` command such as:

```
...
RUN ["powershell", "-command", "Execute-MyCmdlet", "-param1 \"c:\\foo.txt\""]
...
```

While the JSON form is unambiguous and does not use the un-necessary cmd.exe, it does require more verbosity through double-quoting and escaping. The alternate mechanism is to use the `SHELL` instruction and the shell form, making a more natural syntax for Windows users, especially when combined with the escape parser directive:

```
# escape=`

FROM windowsservercore
SHELL ["powershell","-command"]
RUN New-Item -ItemType Directory C:\Example
ADD Execute-MyCmdlet.ps1 c:\example\
RUN c:\example\Execute-MyCmdlet -sample 'hello world'
```

Resulting in:

```
PS E:\docker\build\shell> docker build -t shell .
Sending build context to Docker daemon 3.584 kB
Step 1 : FROM windowsservercore
 ---> 5bc36a335344
Step 2 : SHELL powershell -command
 ---> Running in 87d7a64c9751
 ---> 4327358436c1
Removing intermediate container 87d7a64c9751
Step 3 : RUN New-Item -ItemType Directory C:\Example
 ---> Running in 3e6ba16b8df9


Directory: C:\


Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----         6/2/2016   2:59 PM                Example


 ---> 1f1dfdcec085
Removing intermediate container 3e6ba16b8df9
Step 4 : ADD Execute-MyCmdlet.ps1 c:\example\
 ---> 6770b4c17f29
Removing intermediate container b139e34291dc
Step 5 : RUN c:\example\Execute-MyCmdlet -sample 'hello world'
 ---> Running in abdcf50dfd1f
Hello from Execute-MyCmdlet.ps1 - passed hello world
 ---> ba0e25255fda
Removing intermediate container abdcf50dfd1f
Successfully built ba0e25255fda
PS E:\docker\build\shell>
```

The `SHELL` instruction could also be used to modify the way in which a shell operates. For example, using `SHELL cmd /S /C /V:ON|OFF` on Windows, delayed environment variable expansion semantics could be modified.

The `SHELL` instruction can also be used on Linux should an alternate shell be required such zsh, csh, tcsh and others.

The `SHELL` feature was added in Docker 1.12.

Installing Debian/Ubuntu packages

Run the install on a single run command to merge the update and install. If you add more packages later, this will run the update again and install all the packages needed. If the update is run separately, it will be cached and package installs may fail. Setting the frontend to noninteractive and passing the -y to install is needed for scripted installs. Cleaning and purging at the end of the install minimizes the size of the layer.

```
FROM debian

RUN apt-get update \
 && DEBIAN_FRONTEND=noninteractive apt-get install -y \
    git \
    openssh-client \
    sudo \
    vim \
    wget \
&& apt-get clean \
&& rm -rf /var/lib/apt/lists/*
```

Read Dockerfiles online: https://riptutorial.com/docker/topic/3161/dockerfiles

Chapter 25: Inspecting a running container

Syntax

- docker inspect [OPTIONS] CONTAINERIIMAGE [CONTAINERIIMAGE...]

Examples

Get container information

To get all the information for a container you can run:

```
docker inspect <container>
```

Get specific information from a container

You can get an specific information from a container by running:

```
docker inspect -f '<format>' <container>
```

For instance, you can get the Network Settings by running:

```
docker inspect -f '{{ .NetworkSettings }}' <container>
```

You can also get just the IP address:

```
docker inspect -f '{{ .NetworkSettings.IPAddress }}' <container>
```

The parameter -f means format and will receive a Go Template as input to format what is expected, but this won't bring a beautiful return, so try:

```
docker inspect -f '{{ json .NetworkSettings }}' {{containerIdOrName}}
```

the json keyword will bring the return as a JSON.

So to finish, a little tip is to use python in there to format the output JSON:

```
docker inspect -f '{{ json .NetworkSettings }}' <container> | python -mjson.tool
```

And voila, you can query anything on the docker inspect and make it look pretty in your terminal.

It's also possible to use a utility called "jq" in order to help process `docker inspect` command output.

```
docker inspect -f '{{ json .NetworkSettings }}' aa1 | jq [.Gateway]
```

The above command will return the following output:

```
[
  "172.17.0.1"
]
```

This output is actually a list containing one element. Sometimes, `docker inspect` displays a list of several elements, and you may want to refer to a specific element. For example, if `Config.Env` contains several elements, you can refer to the first element of this list using `index`:

```
docker inspect --format '{{ index (index .Config.Env) 0 }}' <container>
```

The first element is indexed at zero, which means that the second element of this list is at index `1`:

```
docker inspect --format '{{ index (index .Config.Env) 1 }}' <container>
```

Using `len` it is possible to get the number of elements of the list:

```
docker inspect --format ‘{{ len .Config.Env }}’ <container>
```

And using negative numbers, it's possible to refer to the last element of the list:

```
docker inspect –format “{{ index .Config.Cmd $[$(docker inspect –format ‘{{ len .Config.Cmd
}}’ <container>)-1]}}” <container>
```

Some `docker inspect` information comes as a dictionary of key:value, here is an extract of a `docker inspect`of a jess/spotify running container

```
"Config": { "Hostname": "8255f4804dde", "Domainname": "", "User": "spotify", "AttachStdin":
false, "AttachStdout": false, "AttachStderr": false, "Tty": false, "OpenStdin": false,
"StdinOnce": false, "Env": [ "DISPLAY=unix:0",
"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin", "HOME=/home/spotify" ],
"Cmd": [ "-stylesheet=/home/spotify/spotify-override.css" ], "Image": "jess/spotify", "Volumes":
null, "WorkingDir": "/home/spotify", "Entrypoint": [ "spotify" ], "OnBuild": null, "Labels": {}
},
```

so I an get the values of the whole Config section

```
docker inspect -f '{{.Config}}' 825

{8255f4804dde spotify false false false map[] false false false [DISPLAY=unix:0
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin HOME=/home/spotify] [-
stylesheet=/home/spotify/spotify-override.css] false jess/spotify map[] /home/spotify [spotify]
false [] map[] }
```

but also a single field, like the value of Config.Image

```
docker inspect -f '{{index (.Config) "Image" }}' 825

jess/spotify
```

or Config.Cmd

```
docker inspect -f '{{.Config.Cmd}}' 825

[-stylesheet=/home/spotify/spotify-override.css]
```

Inspect an image

In order to inspect an image, you can use the image ID or the image name, consisting of repository and tag. Say, you have the CentOS 6 base image:

```
➜  ~ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
centos              centos6             cf2c3ece5e41        2 weeks ago         194.6 MB
```

In this case you can run either of the following:

- ➜ ~ docker inspect cf2c3ece5e41
- ➜ ~ docker inspect centos:centos6

Both of these command will give you all information available in a JSON array:

```
[
    {
        "Id": "sha256:cf2c3ece5e418fd063bfad5e7e8d083182195152f90aac3a5ca4dbfbf6a1fc2a",
        "RepoTags": [
            "centos:centos6"
        ],
        "RepoDigests": [],
        "Parent": "",
        "Comment": "",
        "Created": "2016-07-01T22:34:39.970264448Z",
        "Container": "b355fe9a01a8f95072e4406763138c5ad9ca0a50dbb0ce07387ba905817d6702",
        "ContainerConfig": {
            "Hostname": "68a1f3cfce80",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
            ],
            "Cmd": [
                "/bin/sh",
                "-c",
                "#(nop) CMD [\"/bin/bash\"]"
            ],
            "Image":
"sha256:cdbcc7980b002dc19b4d5b6ac450993c478927f673339b4e6893647fe2158fa7",
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": null,
            "OnBuild": null,
            "Labels": {
                "build-date": "20160701",
                "license": "GPLv2",
                "name": "CentOS Base Image",
                "vendor": "CentOS"
            }
        },
        "DockerVersion": "1.10.3",
        "Author": "https://github.com/CentOS/sig-cloud-instance-images",
        "Config": {
            "Hostname": "68a1f3cfce80",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
            ],
            "Cmd": [
                "/bin/bash"
            ],
            "Image":
"sha256:cdbcc7980b002dc19b4d5b6ac450993c478927f673339b4e6893647fe2158fa7",
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": null,
            "OnBuild": null,
            "Labels": {
                "build-date": "20160701",
                "license": "GPLv2",
                "name": "CentOS Base Image",
                "vendor": "CentOS"
            }
        },
        "Architecture": "amd64",
        "Os": "linux",
        "Size": 194606575,
        "VirtualSize": 194606575,
        "GraphDriver": {
            "Name": "aufs",
            "Data": null
        },
        "RootFS": {
            "Type": "layers",
            "Layers": [
                "sha256:2714f4a6cdee9d4c987fef019608a4f61f1cda7ccf423aeb8d7d89f745c58b18"
            ]
        }
    }
]
```

Printing specific informations

docker `inspect` supports Go Templates via the `--format` option. This allows for better integration in scripts, without resorting to pipes/sed/grep traditional tools.

Print a container internal IP:

```
docker inspect --format '{{ .NetworkSettings.IPAddress }}' 7786807d8084
```

This is useful for direct network access of load-balancers auto-configuration.

Print a container *init* PID:

```
docker inspect --format '{{ .State.Pid }}' 7786807d8084
```

This is useful for deeper inspection via `/proc` or tools like `strace`.

Advanced formating:

```
docker inspect --format 'Container {{ .Name }} listens on {{ .NetworkSettings.IPAddress }}:{{
range $index, $elem := .Config.ExposedPorts }}{{ $index }}{{ end }}' 5765847de886 7786807d8084
```

Will output:

```
Container /redis listens on 172.17.0.3:6379/tcp
Container /api listens on 172.17.0.2:4000/tcp
```

Debugging the container logs using docker inspect

`docker inspect` command can be used to debug the container logs.

The stdout and stderr of container can be checked to debug the container, whose location can be obtained using `docker inspect`.

Command : `docker inspect <container-id> | grep Source`

It gives the location of containers stdout and stderr.

Examining stdout/stderr of a running container

```
docker logs --follow <containerid>
```

This tails the output of the running container. This is useful if you did not set up a logging driver on the docker daemon.

Read Inspecting a running container online: https://riptutorial.com/docker/topic/1336/inspecting-a-running-container

Chapter 28: Managing containers

Syntax

- docker rm [OPTIONS] CONTAINER [CONTAINER...]
- docker attach [OPTIONS] CONTAINER
- docker exec [OPTIONS] CONTAINER COMMAND [ARG...]
- docker ps [OPTIONS]
- docker logs [OPTIONS] CONTAINER
- docker inspect [OPTIONS] CONTAINERIIMAGE [CONTAINERIIMAGE...]

Remarks

- In the examples above, whenever container is a parameter of the docker command, it is mentioned as `<container>` or `container id` or `<CONTAINER_NAME>`. In all these places you can either pass a container name or container id to specify a container.

Examples

Listing containers

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
PORTS                     NAMES
2bc9b1988080        redis               "docker-entrypoint.sh"   2 weeks ago         Up 2
hours          0.0.0.0:6379->6379/tcp    elephant-redis
817879be2230        postgres            "/docker-entrypoint.s"   2 weeks ago         Up 2
hours          0.0.0.0:65432->5432/tcp   pt-postgres
```

`docker ps` on its own only prints currently running containers. To view all containers (including stopped ones), use the `-a` flag:

```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
PORTS                     NAMES
9cc69f11a0f7        docker/whalesay     "ls /"                   26 hours ago        Exited
(0) 26 hours ago                             berserk_wozniak
2bc9b1988080        redis               "docker-entrypoint.sh"   2 weeks ago         Up 2
hours                0.0.0.0:6379->6379/tcp    elephant-redis
817879be2230        postgres            "/docker-entrypoint.s"   2 weeks ago         Up 2
hours                0.0.0.0:65432->5432/tcp   pt-postgres
```

To list containers with a specific status, use the `-f` command line option to filter the results. Here is an example of listing all containers which have exited:

```
$ docker ps -a -f status=exited
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
```

```
PORTS                      NAMES
9cc69f11a0f7        docker/whalesay     "ls /"                   26 hours ago        Exited
(0) 26 hours ago
```

It is also possible to list only the Container IDs with the `-q` switch. This makes it very easy to operate on the result with other Unix utilities (such as `grep` and `awk`):

```
$ docker ps -aq
9cc69f11a0f7
2bc9b1988080
817879be2230
```

When launching a container with `docker run --name mycontainer1` you give a specific name and not a random name (in the form mood_famous, such as nostalgic_stallman), and it can be easy to find them with such a command

`docker ps -f name=mycontainer1`

Referencing containers

Docker commands which take the name of a container accept three different forms:

Type	Example
Full UUID	`9cc69f11a0f76073e87f25cb6eaf0e079fbfbd1bc47c063bcd25ed3722a8cc4a`
Short UUID	`9cc69f11a0f7`
Name	`berserk_wozniak`

Use `docker ps` to view these values for the containers on your system.

The UUID is generated by Docker and cannot be modified. You can provide a name to the container when you start it `docker run --name <given name> <image>`. Docker will generate a random name to the container if you don't specify one at the time of starting the container.

***NOTE**: The value of the UUID (or a 'short' UUID) can be any length as long as the given value is unique to one container*

Starting and stopping containers

To stop a running container:

```
docker stop <container> [<container>...]
```

This will send the main process in the container a SIGTERM, followed by a SIGKILL if it doesn't stop within the grace period. The name of each container is printed as it stops.

To start a container which is stopped:

```
docker start <container> [<container>...]
```

This will start each container passed in the background; the name of each container is printed as it starts. To start the container in the foreground, pass the `-a` (`--attach`) flag.

List containers with custom format

```
docker ps --format 'table {{.ID}}\t{{.Names}}\t{{.Status}}'
```

Finding a specific container

```
docker ps --filter name=myapp_1
```

Find container IP

To find out the IP address of your container, use:

```
docker inspect <container id> | grep IPAddress
```

or use docker inspect

`docker inspect --format '{{ .NetworkSettings.IPAddress }}' ${CID}`

Restarting docker container

```
docker restart <container> [<container>...]
```

Option **--time** : Seconds to wait for stop before killing the container (default 10)

```
docker restart <container> --time 10
```

Remove, delete and cleanup containers

`docker rm` can be used to remove a specific containers like this:

```
docker rm <container name or id>
```

To remove all containers you can use this expression:

```
docker rm $(docker ps -qa)
```

By default docker will not delete a container that is running. Any container that is running will produce a warning message and not be deleted. All other containers will be deleted.

Alternatively you can use `xargs`:

```
docker ps -aq -f status=exited | xargs -r docker rm
```

Where `docker ps -aq -f status=exited` will return a list of container IDs of containers that have a status of "Exited".

> Warning: All the above examples will only remove 'stopped' containers.

To remove a container, regardless of whether or not it is stopped, you can use the force flag `-f`:

```
docker rm -f <container name or id>
```

To remove all containers, regardless of state:

```
docker rm -f $(docker ps -qa)
```

If you want to remove only containers with a `dead` status:

```
docker rm $(docker ps --all -q -f status=dead)
```

If you want to remove only containers with an `exited` status:

```
docker rm $(docker ps --all -q -f status=exited)
```

These are all permutations of filters used when listing containers.

To remove both unwanted containers and dangling images that use space after version 1.3, use the following (similar to the Unix tool `df`):

```
$ docker system df
```

To remove all unused data:

```
$ docker system prune
```

Run command on an already existing docker container

`docker exec -it <container id> /bin/bash`

It is common to log in an already running container to make some quick tests or see what the application is doing. Often it denotes bad container use practices due to logs and changed files should be placed in volumes. This example allows us log in the container. This supposes that /bin/bash is available in the container, it can be /bin/sh or something else.

```
docker exec <container id> tar -czvf /tmp/backup.tgz /data
docker cp <container id>:/tmp/backup.tgz .
```

This example archives the content of data directory in a tar. Then with `docker cp` you can retrieve it.

Container logs

```
Usage:  docker logs [OPTIONS] CONTAINER

Fetch the logs of a container

  -f, --follow=false        Follow log output
  --help=false              Print usage
  --since=                  Show logs since timestamp
  -t, --timestamps=false    Show timestamps
  --tail=all                Number of lines to show from the end of the logs
```

For example:

```
$ docker ps
CONTAINER ID    IMAGE    COMMAND                  CREATED     STATUS       PORTS
ff9716dda6cb    nginx    "nginx -g 'daemon off"  8 days ago  Up 22 hours  443/tcp,
0.0.0.0:8080->80/tcp

$ docker logs ff9716dda6cb
xx.xx.xx.xx - - [15/Jul/2016:14:03:44 +0000] "GET /index.html HTTP/1.1" 200 511
"https://google.com" "Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/50.0.2661.75 Safari/537.36"
xx.xx.xx.xx - - [15/Jul/2016:14:03:44 +0000] "GET /index.html HTTP/1.1" 200 511
"https://google.com" "Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/50.0.2661.75 Safari/537.36"
```

Connect to an instance running as daemon

There are two ways to achieve that, the first and most known is the following:

```
docker attach --sig-proxy=false <container>
```

This one literally attaches your bash to the container bash, meaning that if you have a running script, you will see the result.

To detach, just type: `Ctl-P Ctl-Q`

But if you need a more friendly way and to be able to create new bash instances, just run the following command:

```
docker exec -it <container> bash
```

Copying file from/to containers

from container to host

```
docker cp CONTAINER_NAME:PATH_IN_CONTAINER PATH_IN_HOST
```

from host to container

```
docker cp PATH_IN_HOST CONTAINER_NAME:PATH_IN_CONTAINER
```

If I use jess/transmission from

https://hub.docker.com/r/jess/transmission/builds/bsn7eqxrkzrhxazcuytbmzp/

, the files in the container are in /transmission/download

and my current directory on the host is /home/$USER/abc, after

`docker cp transmission_id_or_name:/transmission/download .`

I will have the files copied to

`/home/$USER/abc/transmission/download`

you can not, using `docker cp` copy only one file, you copy the directory tree and the files

Remove, delete and cleanup docker volumes

Docker volumes are not automatically removed when a container is stopped. To remove associated volumes when you stop a container:

```
docker rm -v <container id or name>
```

If the `-v` flag is not specified, the volume remains on-disk as a 'dangling volume'. To delete all dangling volumes:

```
docker volume rm $(docker volume ls -qf dangling=true)
```

The `docker volume ls -qf dangling=true` filter will return a list of docker volumes names, including untagged ones, that are not attached to a container.

Alternatively, you can use `xargs`:

```
docker volume ls -f dangling=true -q | xargs --no-run-if-empty docker volume rm
```

Export and import Docker container filesystems

It is possible to save a Docker container's filesystem contents to a tarball archive file. This is useful in a pinch for moving container filesystems to different hosts, for example if a database container has important changes and it isn't otherwise possible to replicate those changes elsewhere. **Please note** that it is preferable to create an entirely new container from an updated image using a `docker run` command or `docker-compose.yml` file, instead of exporting and moving a container's filesystem. Part of Docker's power is the auditability and accountability of its declarative style of creating images and containers. By using `docker export` and `docker import`, this power is subdued because of the obfuscation of changes made inside of a container's filesystem from its original state.

```
docker export -o redis.tar redis
```

The above command will create an empty image and then export the filesystem of the `redis` container into this empty image. To import from a tarball archive, use:

```
docker import ./redis.tar redis-imported:3.0.7
```

This command will create the `redis-imported:3.0.7` image, from which containers can be created. It is also possible to create changes on import, as well as set a commit message:

```
docker import -c="ENV DEBUG true" -m="enable debug mode" ./redis.tar redis-changed
```

The Dockerfile directives available for use with the `-c` command line option are `CMD`, `ENTRYPOINT`, `ENV`, `EXPOSE`, `ONBUILD`, `USER`, `VOLUME`, `WORKDIR`.

Read Managing containers online: https://riptutorial.com/docker/topic/689/managing-containers

Chapter 30: Multiple processes in one container instance

Remarks

Usually each container should hosts one process. In case you need multiple processes in one container (e.g. an SSH server to login to your running container instance) you could get the idea to write you own shell script that starts those processes. In that case you had to take care about the `SIGNAL` handling yourself (e.g. redirecting a caught `SIGINT` to the child processes of your script). That's not really what you want. A simple solution is to use `supervisord` as the containers root process which takes care about `SIGNAL` handling and its child processes lifetime.

But keep in mind, that this ist not the "docker way". To achive this example in the docker way you would log into the `docker host` (the machine the container runs on) and run `docker exec -it container_name /bin/bahs`. This command opens you a shell inside the container as ssh would do.

Examples

Dockerfile + supervisord.conf

To run multiple processes e.g. an Apache web server together with an SSH daemon inside the same container you can use `supervisord`.

Create your `supervisord.conf` configuration file like:

```
[supervisord]
nodaemon=true

[program:sshd]
command=/usr/sbin/sshd -D

[program:apache2]
command=/bin/bash -c "source /etc/apache2/envvars && exec /usr/sbin/apache2 -DFOREGROUND"
```

Then create a `Dockerfile` like:

```
FROM ubuntu:16.04
RUN apt-get install -y openssh-server apache2 supervisor
RUN mkdir -p /var/lock/apache2 /var/run/apache2 /var/run/sshd /var/log/supervisor
COPY supervisord.conf /etc/supervisor/conf.d/supervisord.conf
CMD ["/usr/bin/supervisord"]
```

Then you can build your image:

```
docker build -t supervisord-test .
```

Afterwards you can run it:

```
$ docker run -p 22 -p 80 -t -i supervisord-test
2016-07-26 13:15:21,101 CRIT Supervisor running as root (no user in config file)
2016-07-26 13:15:21,101 WARN Included extra file     "/etc/supervisor/conf.d/supervisord.conf"
during parsing
2016-07-26 13:15:21,112 INFO supervisord started with pid 1
2016-07-26 13:15:21,113 INFO spawned: 'sshd' with pid 6
2016-07-26 13:15:21,115 INFO spawned: 'apache2' with pid 7
...
```

Read Multiple processes in one container instance online:
https://riptutorial.com/docker/topic/4053/multiple-processes-in-one-container-instance

Chapter 31: passing secret data to a running container

Examples

ways to pass secrets in a container

The not very secure way (because `docker inspect` will show it) is to pass an environment variable to

```
docker run
```

such as

`docker run -e password=abc`

or in a file

`docker run --env-file myfile`

where myfile can contain

`password1=abc password2=def`

it is also possible to put them in a volume

`docker run -v $(pwd)/my-secret-file:/secret-file`

some better ways, use

keywhiz https://square.github.io/keywhiz/

vault https://www.hashicorp.com/blog/vault.html

etcd with crypt https://xordataexchange.github.io/crypt/

Read passing secret data to a running container online:
https://riptutorial.com/docker/topic/6481/passing-secret-data-to-a-running-container

Chapter 34: Running containers

Syntax

- docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

Examples

Running a container

```
docker run hello-world
```

This will fetch the latest hello-world image from the Docker Hub (if you don't already have it), create a new container, and run it. You should see a message stating that your installation appears to be working correctly.

Running a different command in the container

```
docker run docker/whalesay cowsay 'Hello, StackExchange!'
```

This command tells Docker to create a container from the `docker/whalesay` image and run the command `cowsay 'Hello, StackExchange!'` in it. It should print a picture of a whale saying `Hello, StackExchange!` to your terminal.

If the entrypoint in the image is the default you can run any command that's available in the image:

```
docker run docker/whalesay ls /
```

If it has been changed during image build you need to reverse it back to the default

```
docker run --entrypoint=/bin/bash docker/whalesay -c ls /
```

Automatically delete a container after running it

Normally, a Docker container persists after it has exited. This allows you to run the container again, inspect its filesystem, and so on. However, sometimes you want to run a container and delete it immediately after it exits. For example to execute a command or show a file from the filesystem. Docker provides the `--rm` command line option for this purpose:

```
docker run --rm ubuntu cat /etc/hosts
```

This will create a container from the "ubuntu" image, show the content of **/etc/hosts** file and then delete the container immediately after it exits. This helps to prevent having to clean up containers after you're done experimenting.

> **Note:** The `--rm` flag doesn't work in conjunction with the `-d` (`--detach`) flag in docker < 1.13.0.

When `--rm` flag is set, Docker also removes the volumes associated with the container when the container is removed. This is similar to running `docker rm -v my-container`. ***Only volumes that are specified without a name are removed***.

For example, with `docker run -it --rm -v /etc -v logs:/var/log centos /bin/produce_some_logs`, the volume of `/etc` will be removed, but the volume of `/var/log` will not. Volumes inherited via --volumes-from will be removed with the same logic -- if the original volume was specified with a name it will not be removed.

Specifying a name

By default, containers created with `docker run` are given a random name like `small_roentgen` or `modest_dubinsky`. These names aren't particularly helpful in identifying the purpose of a container. It is possible to supply a name for the container by passing the `--name` command line option:

```
docker run --name my-ubuntu ubuntu:14.04
```

Names must be unique; if you pass a name that another container is already using, Docker will print an error and no new container will be created.

Specifying a name will be useful when referencing the container within a Docker network. This works for both background and foreground Docker containers.

Containers on the default bridge network **must** be linked to communicate by name.

Binding a container port to the host

```
docker run -p "8080:8080" myApp
docker run -p "192.168.1.12:80:80" nginx
docker run -P myApp
```

In order to use ports on the host have been exposed in an image (via the `EXPOSE` Dockerfile directive, or `--expose` command line option for `docker run`), those ports need to be bound to the host using the `-p` or `-P` command line options. Using `-p` requires that the particular port (and optional host interface) to be specified. Using the uppercase `-P` command line option will force Docker to bind *all* exposed ports in a container's image to the host.

Container restart policy (starting a container at boot)

```
docker run --restart=always -d <container>
```

By default, Docker will not restart containers when the Docker daemon restarts, for example after a host system reboot. Docker provides a restart policy for your containers by supplying the `--restart` command line option. Supplying `--restart=always` will always cause a container to be restarted after the Docker daemon is restarted. **However** when that container is manually stopped (e.g. with `docker stop <container>`), the restart policy will not be applied to the container.

Multiple options can be specified for `--restart` option, based on the requirement (`--restart=[policy]`). These options effect how the container starts at boot as well.

Policy	Result
no	The **default** value. Will not restart container automatically, when container is stopped.
on-failure[:max-retries]	Restart only if the container exits with a failure (`non-zero exit status`). To avoid restarting it indefinitely (in case of some problem), one can limit the number of restart retries the Docker daemon attempts.
always	Always restart the container regardless of the exit status. When you specify `always`, the Docker daemon will try to restart the container indefinitely. The container will also always start on daemon startup, regardless of the current state of the container.
unless-stopped	Always restart the container regardless of its exit status, but do not start it on daemon startup if the container has been put to a stopped state before.

Run a container in background

To keep a container running in the background, supply the `-d` command line option during container startup:

```
docker run -d busybox top
```

The option `-d` runs the container in detached mode. It is also equivalent to `-d=true`.

A container in detached mode cannot be removed automatically when it stops, this means one cannot use the --rm option in combination with -d option.

Assign a volume to a container

A Docker volume is a file or directory which persists beyond the lifetime of the container. It is possible to mount a host file or directory into a container as a volume (bypassing the UnionFS).

Add a volume with the `-v` command line option:

```
docker run -d -v "/data" awesome/app bootstrap.sh
```

This will create a volume and mount it to the path `/data` inside the container.

- Note: You can use the flag `--rm` to automatically remove the volume when the container is removed.

Mounting host directories

To mount a host file or directory into a container:

```
docker run -d -v "/home/foo/data:/data" awesome/app bootstrap.sh
```

- **When specifying a host directory, an absolute path must be supplied.**

This will mount the host directory `/home/foo/data` onto `/data` inside the container. This "bind-mounted host directory" volume is the same thing as a Linux `mount --bind` and therefore temporarily mounts the host directory over the specified container path for the duration of the container's lifetime. Changes in the volume from either the host or the container are reflected immediately in the other, because they are the same destination on disk.

UNIX example mounting a relative folder

```
docker run -d -v $(pwd)/data:/data awesome/app bootstrap.sh
```

Naming volumes

A volume can be named by supplying a string instead of a host directory path, docker will create a volume using that name.

```
docker run -d -v "my-volume:/data" awesome/app bootstrap.sh
```

After creating a named volume, the volume can then be shared with other containers using that name.

Setting environment variables

```
$ docker run -e "ENV_VAR=foo" ubuntu /bin/bash
```

Both `-e` and `--env` can be used to define environment variables inside of a container. It is possible to supply many environment variables using a text file:

```
$ docker run --env-file ./env.list ubuntu /bin/bash
```

Example environment variable file:

```
# This is a comment
TEST_HOST=10.10.0.127
```

The `--env-file` flag takes a filename as an argument and expects each line to be in the `VARIABLE=VALUE` format, mimicking the argument passed to `--env`. Comment lines need only be prefixed with `#`.

Regardless of the order of these three flags, the `--env-file` are processed first, and then `-e`/`--env` flags. This way, any environment variables supplied individually with `-e` or `--env` will override variables supplied in the `--env-var` text file.

Specifying a hostname

By default, containers created with docker run are given a random hostname. You can give the container a different hostname by passing the --hostname flag:

```
docker run --hostname redbox -d ubuntu:14.04
```

Run a container interactively

To run a container interactively, pass in the `-it` options:

```
$ docker run -it ubuntu:14.04 bash
root@8ef2356d919a:/# echo hi
hi
root@8ef2356d919a:/#
```

`-i` keeps STDIN open, while `-t` allocates a pseudo-TTY.

Running container with memory/swap limits

Set memory limit and disable swap limit

```
docker run -it -m 300M --memory-swap -1 ubuntu:14.04 /bin/bash
```

Set both memory and swap limit. In this case, container can use 300M memory and 700M swap.

```
docker run -it -m 300M --memory-swap 1G ubuntu:14.04 /bin/bash
```

Getting a shell into a running (detached) container

Log into a running container

A user can enter a running container in a new interactive bash shell with `exec` command.

Say a container is called `jovial_morse` then you can get an interactive, pseudo-TTY bash shell by running:

```
docker exec -it jovial_morse bash
```

Log into a running container with a specific user

If you want to enter a container as a specific user, you can set it with `-u` or `--user` parameter. The username must exists in the container.

> `-u, --user` Username or UID (format: `<name|uid>[:<group|gid>]`)

This command will log into `jovial_morse` with the `dockeruser` user

```
docker exec -it -u dockeruser jovial_morse bash
```

Log into a running container as root

If you want to log in as root, just simply use the **`-u root`** parameter. Root user always exists.

```
docker exec -it -u root jovial_morse bash
```

Log into a image

You can also log into a image with the `run` command, but this requires an image name instead of a container name.

```
docker run -it dockerimage bash
```

Log into a intermediate image (debug)

You can log into an intermediate image as well, which is created during a Dockerfile build.

Output of `docker build .`

```
$ docker build .
Uploading context 10240 bytes
Step 1 : FROM busybox
Pulling repository busybox
 ---> e9aa60c60128MB/2.284 MB (100%) endpoint: https://cdn-registry-1.docker.io/v1/
Step 2 : RUN ls -lh /
 ---> Running in 9c9e81692ae9
total 24
drwxr-xr-x    2 root     root         4.0K Mar 12  2013 bin
drwxr-xr-x    5 root     root         4.0K Oct 19 00:19 dev
drwxr-xr-x    2 root     root         4.0K Oct 19 00:19 etc
drwxr-xr-x    2 root     root         4.0K Nov 15 23:34 lib
lrwxrwxrwx    1 root     root            3 Mar 12  2013 lib64 -> lib
dr-xr-xr-x  116 root     root            0 Nov 15 23:34 proc
lrwxrwxrwx    1 root     root            3 Mar 12  2013 sbin -> bin
dr-xr-xr-x   13 root     root            0 Nov 15 23:34 sys
drwxr-xr-x    2 root     root         4.0K Mar 12  2013 tmp
drwxr-xr-x    2 root     root         4.0K Nov 15 23:34 usr
 ---> b35f4035db3f
Step 3 : CMD echo Hello world
```

```
 ---> Running in 02071fceb21b
 ---> f52f38b7823e
```

Notice the `---> Running in 02071fceb21b` output, you can log into these images:

```
docker run -it 02071fceb21b bash
```

Passing stdin to the container

In cases such as restoring a database dump, or otherwise wishing to push some information through a pipe from the host, you can use the `-i` flag as an argument to `docker run` or `docker exec`.

E.g., assuming you want to put to a containerized mariadb client a database dump that you have on the host, in a local `dump.sql` file, you can perform the following command:

```
docker exec -i mariadb bash -c 'mariadb "-p$MARIADB_PASSWORD" ' < dump.sql
```

In general,

```
docker exec -i container command < file.stdin
```

Or

```
docker exec -i container command <<EOF
inline-document-from-host-shell-HEREDOC-syntax
EOF
```

Detaching from a container

While attached to a running container with a pty assigned (`docker run -it ...`), you can press `ControlP` - `ControlQ` to detach.

Overriding image entrypoint directive

```
docker run --name="test-app" --entrypoint="/bin/bash" example-app
```

This command will override the `ENTRYPOINT` directive of the `example-app` image when the container `test-app` is created. The `CMD` directive of the image will remain unchanged unless otherwise specified:

```
docker run --name="test-app" --entrypoint="/bin/bash" example-app /app/test.sh
```

In the above example, both the `ENTRYPOINT` and the `CMD` of the image have been overridden. This container process becomes `/bin/bash /app/test.sh`.

Add host entry to container

```
docker run --add-host="app-backend:10.15.1.24" awesome-app
```

This command adds an entry to the container's `/etc/hosts` file, which follows the format `--add-host <name>:<address>`. In this example, the name `app-backend` will resolve to `10.15.1.24`. This is particularly useful for tying disparate app components together programmatically.

Prevent container from stopping when no commands are running

A container will stop if no command is running on the foreground. Using the `-t` option will keep the container from stopping, even when detached with the `-d` option.

```
docker run -t -d debian bash
```

Stopping a container

```
docker stop mynginx
```

Additionally, the container id can also be used to stop the container instead of its name.

This will stop a running container by sending the SIGTERM signal and then the SIGKILL signal if necessary.

Further, the kill command can be used to immediately send a SIGKILL or any other specified signal using the `-s` option.

```
docker kill mynginx
```

Specified signal:

```
docker kill -s SIGINT mynginx
```

Stopping a container doesn't delete it. Use `docker ps -a` to see your stopped container.

Execute another command on a running container

When required you can tell Docker to execute additional commands on an already running container using the `exec` command. You need the container's ID which you can get with `docker ps`.

```
docker exec 294fbc4c24b3 echo "Hello World"
```

You can attach an interactive shell if you use the `-it` option.

```
docker exec -it 294fbc4c24b3 bash
```

Running GUI apps in a Linux container

By default, a Docker container won't be able to *run* a GUI application.