CMD vs ENTRYPOINT

Docker has a default entrypoint which is `/bin/sh -c` but does not have a default command.

When you run docker like this: `docker run -i -t ubuntu bash` the entrypoint is the default `/bin/sh -c`, the image is `ubuntu` and the command is `bash`.

The command is run via the entrypoint. i.e., the actual thing that gets executed is `/bin/sh -c bash`. This allowed Docker to implement `RUN` quickly by relying on the shell's parser.

Later on, people asked to be able to customize this, so `ENTRYPOINT` and `--entrypoint` were introduced.

Everything after `ubuntu` in the example above is the command and is passed to the entrypoint. When using the `CMD` instruction, it is exactly as if you were doing `docker run -i -t ubuntu <cmd>`. <cmd> will be the parameter of the entrypoint.

You will also get the same result if you instead type this command `docker run -i -t ubuntu`. You will still start a bash shell in the container because of the ubuntu Dockerfile specified a default CMD: `CMD ["bash"]`

As everything is passed to the entrypoint, you can have a very nice behavior from your images. @Jiri example is good, it shows how to use an image as a "binary". When using `["/bin/cat"]` as entrypoint and then doing `docker run img /etc/passwd`, you get it, `/etc/passwd` is the command and is passed to the entrypoint so the end result execution is simply `/bin/cat /etc/passwd`.

Another example would be to have any cli as entrypoint. For instance, if you have a redis image, instead of running `docker run redisimg redis -H something -u toto get key`, you can simply have `ENTRYPOINT ["redis", "-H", "something", "-u", "toto"]` and then run like this for the same result: `docker run redisimg get key`.

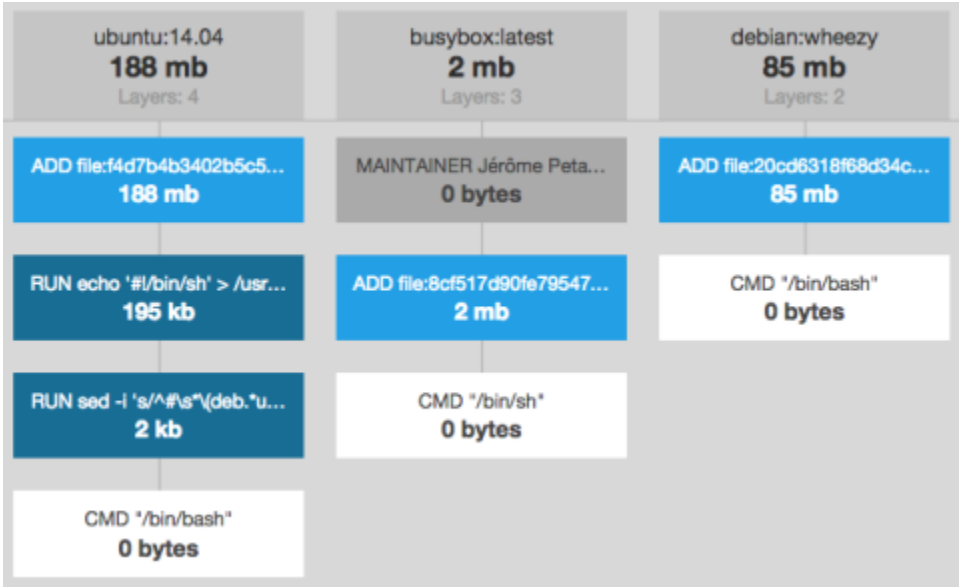
ENTRYPOINT or CMD

Ultimately, both ENTRYPOINT and CMD give you a way to identify which executable should be run when a container is started from your image. In fact, if you want your image to be runnable (without additional `docker run` command line arguments) you **must** specify an ENTRYPOINT or CMD.

Trying to run an image which doesn't have an ENTRYPOINT or CMD declared will result in an error

```
$ docker run alpine
FATA[0000] Error response from daemon: No command specified
```

Many of the Linux distro base images that you find on the Docker Hub will use a shell like */bin/sh* or */bin/bash* as the the CMD executable. This means that anyone who runs those images will get dropped into an interactive shell by default (assuming, of course, that they used the `-i` and `-t` flags with the `docker run` command).



This makes sense for a general-purpose base image, but you will probably want to pick a more specific CMD or ENTRYPOINT for your own images.

Overrides

The ENTRYPOINT or CMD that you specify in your Dockerfile identify the default executable for your image. However, the user has the option to override either of these values at run time.

For example, let's say that we have the following Dockerfile

```
FROM ubuntu:trusty
CMD ping localhost
```

If we build this image (with tag "demo") and run it we would see the following output:

```
$ docker run -t demo
PING localhost (127.0.0.1) 56(84) bytes of data.
64 bytes from localhost (127.0.0.1): icmp_seq=1 ttl=64 time=0.051 ms
64 bytes from localhost (127.0.0.1): icmp_seq=2 ttl=64 time=0.038 ms
^C
--- localhost ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.026/0.032/0.039/0.008 ms
```

You can see that the *ping* executable was run automatically when the container was started. However, we can override the default CMD by specifying an argument **after** the image name when starting the container:

```
$ docker run demo hostname
6c1573c0d4c0
```

In this case, *hostname* was run in place of *ping*

The default ENTRYPOINT can be similarly overridden but it requires the use of the `--entrypoint` flag:

```
$ docker run --entrypoint hostname demo
075a2fa95ab7
```

Given how much easier it is to override the CMD, the recommendation is use CMD in your Dockerfile when you want the user of your image to have the flexibility to run whichever executable they choose when starting the container. For example, maybe you have a general Ruby image that will start-up an interactive *irb* session by default (`CMD irb`) but you also want to give the user the option to run an arbitrary Ruby script (`docker run ruby ruby -e 'puts "Hello"'`)

In contrast, ENTRYPOINT should be used in scenarios where you want the container to behave exclusively as if it were the executable it's wrapping. That is, when you don't want or expect the user to override the executable you've specified.

There are many situations where it may be convenient to use Docker as portable packaging for a specific executable. Imagine you have a utility implemented as a Python script you need to distribute but don't want to burden the end-user with installation of the correct interpreter version and

dependencies. You could package everything in a Docker image with an ENTRYPOINT referencing your script. Now the user can simply `docker run` your image and it will behave as if they are running your script directly.

Of course you can achieve this same thing with CMD, but the use of ENTRYPOINT sends a strong message that this container is only intended to run this one command.

The utility of ENTRYPOINT will become clearer when we show how you can combine ENTRYPOINT and CMD together, but we'll get to that later.

Shell vs. Exec

Both the ENTRYPOINT and CMD instructions support two different forms: the *shell form* and the *exec form*. In the example above, we used the *shell form* which looks like this:

```
CMD executable param1 param2
```

When using the shell form, the specified binary is executed with an invocation of the shell using `/bin/sh -c`. You can see this clearly if you run a container and then look at the `docker ps` output:

```
$ docker run -d demo
15bfcddb11b5cde0e230246f45ba6eeb1e6f56edb38a91626ab9c478408cb615

$ docker ps -l
CONTAINER ID IMAGE COMMAND CREATED
15bfcddb4312 demo:latest "/bin/sh -c 'ping localhost'" 2 seconds ago
```

Here we've run the "demo" image again and you can see that the command which was executed was `/bin/sh -c 'ping localhost'`.

This appears to work just fine, but there are some subtle issues that can occur when using the *shell form* of either the ENTRYPOINT or CMD instruction. If we peek inside our running container and look at the running processes we will see something like this:

```
$ docker exec 15bfcddb ps -f
UID PID PPID C STIME TTY TIME CMD
root 1 0 0 20:14 ? 00:00:00 /bin/sh -c ping localhost
root 9 1 0 20:14 ? 00:00:00 ping localhost
root 49 0 0 20:15 ? 00:00:00 ps -f
```

Note how the process running as PID 1 is **not** our ping command, but is the */bin/sh* executable. This can be problematic if we need to send any sort of POSIX signals to the container since */bin/sh* won't forward signals to child processes (for a detailed write-up, see Gracefully Stopping Docker Containers).

Beyond the PID 1 issue, you may also run into problems with the *shell form* if you're building a minimal image which doesn't even include a shell binary. When Docker is constructing the command to be run it doesn't check to see if the shell is available inside the container -- if you don't have */bin/sh* in your image, the container will simply fail to start.

A better option is to use the *exec form* of the ENTRYPOINT/CMD instructions which looks like this:

```
CMD ["executable","param1","param2"]
```

Note that the content appearing after the CMD instruction in this case is formatted as a JSON array.

When the *exec form* of the CMD instruction is used the command will be executed without a shell.

Let's change our Dockerfile from the example above to see this in action:

```
FROM ubuntu:trusty
CMD ["/bin/ping","localhost"]
```

Rebuild the image and look at the command that is generated for the running container:

```
$ docker build -t demo .
[truncated]

$ docker run -d demo
90cd472887807467d699b55efaf2ee5c4c79eb74ed7849fc4d2dbfea31dce441

$ docker ps -l
CONTAINER ID IMAGE COMMAND CREATED
90cd47288780 demo:latest "/bin/ping localhost" 4 seconds ago
```

Now */bin/ping* is being run directly without the intervening shell process (and, as a result, will end up as PID 1 inside the container).

Whether you're using ENTRYPOINT or CMD (or both) the recommendation is to always use the *exec form* so that's it's obvious which command is running as PID 1 inside your container.

ENTRYPOINT and CMD

Up to this point, we've discussed how to use ENTRYPOINT **or** CMD to specify your image's default executable. However, there are some cases where it makes sense to use ENTRYPOINT **and** CMD together.

Combining ENTRYPOINT and CMD allows you to specify the default executable for your image while also providing default arguments to that executable which may be overridden by the user. Let's look at an example:

```
FROM ubuntu:trusty
ENTRYPOINT ["/bin/ping","-c","3"]
CMD ["localhost"]
```

Let's build and run this image without any additional `docker run` arguments:

```
$ docker build -t ping .
[truncated]

$ docker run ping
PING localhost (127.0.0.1) 56(84) bytes of data.
64 bytes from localhost (127.0.0.1): icmp_seq=1 ttl=64 time=0.025 ms
64 bytes from localhost (127.0.0.1): icmp_seq=2 ttl=64 time=0.038 ms
64 bytes from localhost (127.0.0.1): icmp_seq=3 ttl=64 time=0.051 ms
```

```
--- localhost ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1999ms
rtt min/avg/max/mdev = 0.025/0.038/0.051/0.010 ms

$ docker ps -l
CONTAINER ID IMAGE COMMAND CREATED
82df66a2a9f1 ping:latest "/bin/ping -c 3 localhost" 6 seconds ago
```

Note that the command which was executed is a combination of the ENTRYPOINT and CMD values that were specified in the Dockerfile. When both an ENTRYPOINT and CMD are specified, the CMD string(s) will be appended to the ENTRYPOINT in order to generate the container's command string. Remember that the CMD value can be easily overridden by supplying one or more arguments to `docker run` after the name of the image. In this case we could direct our ping to a different host by doing something like this:

```
$ docker run ping docker.io
PING docker.io (162.242.195.84) 56(84) bytes of data.
64 bytes from 162.242.195.84: icmp_seq=1 ttl=61 time=76.7 ms
64 bytes from 162.242.195.84: icmp_seq=2 ttl=61 time=81.5 ms
64 bytes from 162.242.195.84: icmp_seq=3 ttl=61 time=77.8 ms

--- docker.io ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 76.722/78.695/81.533/2.057 ms

$ docker ps -l --no-trunc
CONTAINER ID IMAGE COMMAND CREATED
0d739d5ea4e5 ping:latest "/bin/ping -c 3 docker.io" 51 seconds ago
```

Running the image starts to feel like running any other executable -- you specify the name of the command you want to run followed by the arguments you want to pass to that command.

Note how the `-c 3` argument that was included as part of the ENTRYPOINT essentially becomes a "hard-coded" argument for the *ping* command (the `-c` flag is used to limit the ping count to the specified number). It's included in each invocation of the image and can't be overridden in the same way as the CMD parameter.

Always Exec

When using ENTRYPOINT and CMD together it's important that you **always** use the *exec form* of both instructions. Trying to use the *shell form*, or mixing-and-matching the *shell* and *exec forms* will almost never give you the result you want.

The table below shows the command string that results from combining the various forms of the ENTRYPOINT and CMD instructions.

```
Dockerfile    Command
ENTRYPOINT /bin/ping -c 3
CMD localhost    /bin/sh -c '/bin/ping -c 3' /bin/sh -c localhost
ENTRYPOINT ["/bin/ping","-c","3"]
CMD localhost    /bin/ping -c 3 /bin/sh -c localhost
ENTRYPOINT /bin/ping -c 3
CMD ["localhost"]"    /bin/sh -c '/bin/ping -c 3' localhost
```

```
ENTRYPOINT ["/bin/ping","-c","3"]
CMD ["localhost"]    /bin/ping -c 3 localhost
```

The only one of these that results in a valid command string is when the ENTRYPOINT and CMD are both specified using the *exec form*.

Conclusion

If you want your image to actually do anything when it is run, you should definitely configure some sort of ENTRYPOINT or CMD in you Dockerfile. However, remember that they aren't mutually exclusive. In many cases you can improve the user experience of your image by using them in combination.