Overriding Dockerfile image defaults

When a developer builds an image from a *Dockerfile* or when she commits it, the developer can set a number of default parameters that take effect when the image starts up as a container.

Four of the Dockerfile commands cannot be overridden at runtime: `FROM`, `MAINTAINER`, `RUN`, and `ADD`. Everything else has a corresponding override in `docker run`. We'll go through what the developer might have set in each Dockerfile instruction and how the operator can override that setting.

- CMD (Default Command or Options)
- ENTRYPOINT (Default Command to Execute at Runtime)
- EXPOSE (Incoming Ports)
- ENV (Environment Variables)
- HEALTHCHECK
- VOLUME (Shared Filesystems)
- USER
- WORKDIR

CMD (default command or options)

Recall the optional `COMMAND` in the Docker commandline:

```
$ docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]
```

This command is optional because the person who created the `IMAGE` may have already provided a default `COMMAND` using the Dockerfile `CMD` instruction. As the operator (the person running a container from the image), you can override that `CMD` instruction just by specifying a new `COMMAND`.

If the image also specifies an `ENTRYPOINT` then the `CMD` or `COMMAND` get appended as arguments to the `ENTRYPOINT`.

ENTRYPOINT (default command to execute at runtime)

```
--entrypoint="": Overwrite the default entrypoint set by the image
```

The `ENTRYPOINT` of an image is similar to a `COMMAND` because it specifies what executable to run when the container starts, but it is (purposely) more difficult to override. The `ENTRYPOINT` gives a container its default nature or behavior, so that when you set an `ENTRYPOINT` you can run the container *as if it were that binary*, complete with default options, and you can pass in more options via the `COMMAND`. But, sometimes an operator may want to run something else inside the container, so you can override the default `ENTRYPOINT` at runtime by using a string to specify the new `ENTRYPOINT`. Here is an example of how to run a shell in a container that has been set up to automatically run something else (like `/usr/bin/redis-server`):

```
$ docker run -it --entrypoint /bin/bash example/redis
```

or two examples of how to pass more parameters to that ENTRYPOINT:

```
$ docker run -it --entrypoint /bin/bash example/redis -c ls -l
$ docker run -it --entrypoint /usr/bin/redis-cli example/redis --help
```

> **Note**: Passing `--entrypoint` will clear out any default command set on the image (i.e. any `CMD` instruction in the Dockerfile used to build it).

EXPOSE (incoming ports)

The following `run` command options work with container networking:

```
--expose=[]: Expose a port or a range of ports inside the container.
             These are additional to those exposed by the `EXPOSE` instruction
-P         : Publish all exposed ports to the host interfaces
-p=[]      : Publish a container's port or a range of ports to the host
               format: ip:hostPort:containerPort | ip::containerPort |
hostPort:containerPort | containerPort
               Both hostPort and containerPort can be specified as a
               range of ports. When specifying ranges for both, the
               number of container ports in the range must match the
               number of host ports in the range, for example:
                   -p 1234-1236:1234-1236/tcp

               When specifying a range for hostPort only, the
               containerPort must not be a range.  In this case the
               container port is published somewhere within the
               specified hostPort range. (e.g., `-p 1234-1236:1234/tcp`)

               (use 'docker port' to see the actual mapping)

--link=""  : Add link to another container (<name or id>:alias or <name or id>)
```

With the exception of the `EXPOSE` directive, an image developer hasn't got much control over networking. The `EXPOSE` instruction defines the initial incoming ports that provide services. These ports are available to processes inside the container. An operator can use the `--expose` option to add to the exposed ports.

To expose a container's internal port, an operator can start the container with the `-P` or `-p` flag. The exposed port is accessible on the host and the ports are available to any client that can reach the host.

The `-P` option publishes all the ports to the host interfaces. Docker binds each exposed port to a random port on the host. The range of ports are within an *ephemeral port range* defined by `/proc/sys/net/ipv4/ip_local_port_range`. Use the `-p` flag to explicitly map a single port or range of ports.

The port number inside the container (where the service listens) does not need to match the port number exposed on the outside of the container (where clients connect). For example, inside the container an HTTP service is listening on port 80 (and so the image developer specifies `EXPOSE 80` in the Dockerfile). At runtime, the port might be bound to 42800 on the host. To find the

mapping between the host ports and the exposed ports, use `docker port`.

If the operator uses `--link` when starting a new client container in the default bridge network, then the client container can access the exposed port via a private networking interface. If `--link` is used when starting a container in a user-defined network as described in *Docker network overview*), it will provide a named alias for the container being linked to.

ENV (environment variables)

When a new container is created, Docker will set the following environment variables automatically:

Variable	Value
`HOME`	Set based on the value of `USER`
`HOSTNAME`	The hostname associated with the container
`PATH`	Includes popular directories, such as : `/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin`
`TERM`	`xterm` if the container is allocated a pseudo-TTY

Additionally, the operator can **set any environment variable** in the container by using one or more `-e` flags, even overriding those mentioned above, or already defined by the developer with a Dockerfile `ENV`:

```
$ docker run -e "deep=purple" --rm ubuntu /bin/bash -c export
declare -x HOME="/"
declare -x HOSTNAME="85bc26a0e200"
declare -x OLDPWD
declare -x PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
declare -x PWD="/"
declare -x SHLVL="1"
declare -x deep="purple"
```

Similarly the operator can set the **hostname** with `-h`.

HEALTHCHECK

```
  --health-cmd            Command to run to check health
  --health-interval       Time between running the check
  --health-retries        Consecutive failures needed to report unhealthy
  --health-timeout        Maximum time to allow one check to run
  --no-healthcheck        Disable any container-specified HEALTHCHECK
```

Example:

```
$ docker run --name=test -d \
    --health-cmd='stat /etc/passwd || exit 1' \
    --health-interval=2s \
    busybox sleep 1d
$ sleep 2; docker inspect --format='{{.State.Health.Status}}' test
healthy
$ docker exec test rm /etc/passwd
$ sleep 2; docker inspect --format='{{json .State.Health}}' test
{
  "Status": "unhealthy",
  "FailingStreak": 3,
  "Log": [
    {
      "Start": "2016-05-25T17:22:04.635478668Z",
      "End": "2016-05-25T17:22:04.7272552Z",
      "ExitCode": 0,
      "Output": "  File: /etc/passwd\n  Size: 334       \tBlocks: 8          IO
Block: 4096   regular file\nDevice: 32h/50d\tInode: 12          Links:
1\nAccess: (0664/-rw-rw-r--)  Uid: (    0/    root)   Gid: (    0/
root)\nAccess: 2015-12-05 22:05:32.000000000\nModify: 2015..."
    },
    {
      "Start": "2016-05-25T17:22:06.732900633Z",
      "End": "2016-05-25T17:22:06.822168935Z",
      "ExitCode": 0,
      "Output": "  File: /etc/passwd\n  Size: 334       \tBlocks: 8          IO
Block: 4096   regular file\nDevice: 32h/50d\tInode: 12          Links:
1\nAccess: (0664/-rw-rw-r--)  Uid: (    0/    root)   Gid: (    0/
root)\nAccess: 2015-12-05 22:05:32.000000000\nModify: 2015..."
    },
    {
      "Start": "2016-05-25T17:22:08.823956535Z",
      "End": "2016-05-25T17:22:08.897359124Z",
      "ExitCode": 1,
      "Output": "stat: can't stat '/etc/passwd': No such file or directory\n"
    },
    {
      "Start": "2016-05-25T17:22:10.898802931Z",
      "End": "2016-05-25T17:22:10.969631866Z",
      "ExitCode": 1,
      "Output": "stat: can't stat '/etc/passwd': No such file or directory\n"
    },
    {
      "Start": "2016-05-25T17:22:12.971033523Z",
      "End": "2016-05-25T17:22:13.082015516Z",
      "ExitCode": 1,
      "Output": "stat: can't stat '/etc/passwd': No such file or directory\n"
    }
  ]
}
```

The health status is also displayed in the `docker ps` output.

TMPFS (mount tmpfs filesystems)

```
--tmpfs=[]: Create a tmpfs mount with: container-dir[:<options>],
            where the options are identical to the Linux
            'mount -t tmpfs -o' command.
```

The example below mounts an empty tmpfs into the container with the `rw`, `noexec`, `nosuid`, and `size=65536k` options.

```
$ docker run -d --tmpfs /run:rw,noexec,nosuid,size=65536k my_image
```

VOLUME (shared filesystems)

```
-v, --volume=[host-src:]container-dest[:<options>]: Bind mount a volume.
The comma-delimited `options` are [rw|ro], [z|Z],
[[r]shared|[r]slave|[r]private], and [nocopy].
The 'host-src' is an absolute path or a name value.

If neither 'rw' or 'ro' is specified then the volume is mounted in
read-write mode.

The `nocopy` modes is used to disable automatic copying requested volume
path in the container to the volume storage location.
For named volumes, `copy` is the default mode. Copy modes are not supported
for bind-mounted volumes.

--volumes-from="": Mount all volumes from the given container(s)
```

> **Note**: When using systemd to manage the Docker daemon's start and stop, in the systemd unit file there is an option to control mount propagation for the Docker daemon itself, called `MountFlags`. The value of this setting may cause Docker to not see mount propagation changes made on the mount point. For example, if this value is `slave`, you may not be able to use the `shared` or `rshared` propagation on a volume.

The volumes commands are complex enough to have their own documentation in section *Manage data in containers*. A developer can define one or more `VOLUME`'s associated with an image, but only the operator can give access from one container to another (or from a container to a volume mounted on the host).

The `container-dest` must always be an absolute path such as `/src/docs`. The `host-src` can either be an absolute path or a `name` value. If you supply an absolute path for the `host-dir`, Docker bind-mounts to the path you specify. If you supply a `name`, Docker creates a named volume by that `name`.

A `name` value must start with an alphanumeric character, followed by `a-z0-9`, _ (underscore), `.` (period) or `-` (hyphen). An absolute path starts with a `/` (forward slash).

For example, you can specify either `/foo` or `foo` for a `host-src` value. If you supply the `/foo` value, Docker creates a bind-mount. If you supply the `foo` specification, Docker creates a named volume.

USER

`root` (id = 0) is the default user within a container. The image developer can create additional users. Those users are accessible by name. When passing a numeric ID, the user does not have to exist in the container.

The developer can set a default user to run the first process with the Dockerfile `USER` instruction. When starting a container, the operator can override the `USER` instruction by passing the `-u` option.

```
-u="", --user="": Sets the username or UID used and optionally the groupname or
GID for the specified command.

The followings examples are all valid:
--user=[ user | user:group | uid | uid:gid | user:gid | uid:group ]
```

> **Note:** if you pass a numeric uid, it must be in the range of 0-2147483647.

WORKDIR

The default working directory for running binaries within a container is the root directory (`/`), but the developer can set a different default with the Dockerfile `WORKDIR` command. The operator can override this with:

```
-w="": Working directory inside the container
```