DOCKER CMD VS. ENTRYPOINT: WHAT'S THE DIFFERENCE AND HOW TO CHOOSE



CMD and ENTRYPOINT are two Dockerfile instructions that together define the command that runs when your container starts. You must use these instructions in your Dockerfiles so that users can easily interact with your images. Because CMD and ENTRYPOINT work in tandem, they can often be confusing to understand. This article helps eliminate any potential disparity in this realm.

In a cloud-native setup, Docker containers are essential elements that ensure an application runs effectively across different computing environments. These containers are meant to carry specific tasks and processes of an application workflow and are supported by Docker images.

The images, on the other hand, are run by executing Docker instructions through a Dockerfile. There are three types of instructions (commands) that you use to build and run Dockerfiles:

- **RUN.** Mainly used to build images and install applications and packages, RUN builds a new layer over an existing image by committing the results.
- **CMD.** Sets default parameters that can be overridden from the Docker Command Line Interface (CLI) when a container is running.
- **ENTRYPOINT.** Default parameters that cannot be overridden when Docker Containers run with CLI parameters.

Any Docker image must have an ENTRYPOINT or CMD declaration for a container to start. Though the ENTRYPOINT and CMD instructions may seem similar at first glance, there are fundamental differences in how they build container images.

(*This is part of our Docker Guide. Use the right-hand menu to navigate.)*

Shell form vs. executable form

First, we need to understand how a Docker Daemon processes instructions once they are passed.

All Docker instruction types (commands) can be specified in either shell or exec forms. Let's build a sample Dockerfile to understand these two commands.

(Explore more Docker commands.)

Shell command form

As the name suggests, a shell form of instructions initiate processes that run within the shell. To execute this, invoke /bin/sh -c <command>.

Typically, every execution through a shell command requires environment variables to go through validation before returning the results.

Syntaxes of shell commands are specified in the form:
<instruction> <command>
Examples of shell form commands include:

```
RUN         yum -y update
RUN         yum -y install httpd
COPY        ./index.html/var/www/index.html
CMD         echo “Hello World”
```

A Dockerfile named Darwin that uses the shell command will have the following specifications:

```
ENV name Darwin
ENTRYPOINT /bin/echo "Welcome, $name"
```

(The command specifications used above are for reference. You can include any other shell command based on your own requirements.)

Based on the specification above, the output of the docker run-it Darwin command will be:

Welcome, Darwin

This command form directs the shell to go through validation before returning results, which often leads to performance bottlenecks. As a result, shell forms are usually not a preferred method unless there are specific command/environment validation requirements.

Executable command form

Unlike the shell command type, an instruction written in executable form directly runs the executable binaries, without going through shell validation and processing.

Executable command syntaxes are specified in the form:

<instruction>

Examples of executable commands include:

```
RUN
```

```
CMD
COPY
```

To build a Dockerfile named Darwin in exec form:

```
ENV name Darwin
ENTRYPOINT
```

Because this avoids a shell processing, the output of the docker run -it Darwin command will be returned as: Welcome, $name.

This is because the environment variable is not substituted in the Dockerfile. To run bash in exec form, specify /bin/bash as executable, i.e.:

```
ENV name Darwin
ENTRYPOINT
```

This prompts shell processing, so the output of the Dockerfile will be: Welcome, Darwin.

Commands in a containerized setup are essential instructions that are passed to the operating environment for a desired output. It is of utmost importance to use the right command form for passing instructions in order to:

- Return the desired result
- Ensure that you don't push the environment into unnecessary processing, thereby impacting operational efficiency

---

Interested in Enterprise DevOps? Learn more about DevOps Solutions and Tools with BMC. ›

CMD vs. ENTRYPOINT: Fundamental differences

CMD and ENTRYPOINT instructions have fundamental differences in how they function, making each one suitable for different applications, environments, and scenarios.

They both specify programs that execute when the container starts running, but:

- CMD commands are ignored by Daemon when there are parameters stated within the docker run command.
- ENTRYPOINT instructions are not ignored, but instead, are appended as command-line parameters by treating those as arguments of the command.

Next, let's take a closer look. We'll use both command forms to go through the different stages of running a Docker container.

Docker CMD

Docker CMD commands are passed through a Dockerfile that consists of:

- Instructions on building a Docker image
- Default binaries for running a container over the image

With a CMD instruction type, a default command/program executes even if no command is specified in the CLI.

Ideally, there should be a single CMD command within a Dockerfile. For example, where there are multiple CMD commands in a Dockerfile, all except the last one are ignored for execution.

An essential feature of a CMD command is its ability to be overridden. This allows users to execute commands through the CLI to override CMD instructions within a Dockerfile.

A Docker CMD instruction can be written in both Shell and Exec forms as:

- Exec form: `CMD`
- Shell form: `CMD command parameter1 parameter2`

Stage 1. Creating a Dockerfile

When building a Dockerfile, the CMD instruction specifies the default program that will execute once the container runs. A quick point to note: CMD commands will only be utilized when command-line arguments are missing.

We'll look at a Dockerfile named `Darwin` with CMD instructions and analyze its behavior.

The Dockerfile specifications for `Darwin` are:

```
FROM centos:7
RUN    apt-get update
RUN     apt-get -y install python
COPY ./opt/source code
CMD
```

The `CMD` instruction in the file above echoes the message `Hello, Darwin` when the container is started without a CLI argument.

Stage 2. Building an image

Docker images are built from Dockerfiles using the command:

```
$ docker build -t Darwin .
```

The above command does two things:

- Tells the Docker Daemon to build an image
- Sets the tag name to `Darwin` located within the current directory

Stage 3. Running a Docker container

To run a Docker container, use the `docker run` command:

```
$ docker run Darwin
```

Since this excludes a Command-line argument, the container runs the default CMD instruction and

displays Hello, Darwin as output.

If we add an argument with the run command, it overrides the default instruction, i.e.:

```
$ docker run Darwin hostname
```

As a CMD default command gets overridden, the above command will run the container and display the hostname, thereby ignoring the echo instruction in the Dockerfile with the following output:

6e14beead430

which is the hostname of the Darwin container.

When to use CMD

The best way to use a CMD instruction is by specifying default programs that should run when users do not input arguments in the command line.

This instruction ensures the container is in a running state by starting an application as soon as the container image is run. By doing so, the CMD argument loads the base image as soon as the container starts.

Additionally, in specific use cases, a docker run command can be executed through a CLI to override instructions specified within the Dockerfile.

Docker ENTRYPOINT

In Dockerfiles, an ENTRYPOINT instruction is used to set executables that will always run when the container is initiated.

Unlike CMD commands, ENTRYPOINT commands cannot be ignored or overridden—even when the container runs with command line arguments stated.

A Docker ENTRYPOINT instruction can be written in both shell and exec forms:

- Exec form: ENTRYPOINT
- Shell form: ENTRYPOINT command parameter1 parameter2

Stage 1. Creating a Dockerfile

ENTRYPOINT instructions are used to build Dockerfiles meant to run specific commands.

These are reference Dockerfile specifications with an Entrypoint command:

```
FROM centos:7
RUN    apt-get update
RUN     apt-get -y install python
COPY ./opt/source code
ENTRYPOINT
```

The above Dockerfile uses an ENTRYPOINT instruction that echoes Hello, Darwin when the container is running.

Stage 2. Building an Image

The next step is to build a Docker image. Use the command:

```
$ docker build -t Darwin .
```

When building this image, the daemon looks for the ENTRYPOINT instruction and specifies it as a default program that will run with or without a command-line input.

Stage 3. Running a Docker container

When running a Docker container using the Darwin image without command-line arguments, the default ENTRYPOINT instructions are executed, echoing Hello, Darwin.

In case additional command-line arguments are introduced through the CLI, the ENTRYPOINT is not ignored. Instead, the command-line parameters are appended as arguments for the ENTRYPOINT command, i.e.:

```
$ docker run Darwin hostname
```

will execute the ENTRYPOINT, echoing **Hello, Darwin** and then displaying the hostname to return the following output:

```
Hello, Darwin 6e14beead430
```

When to use ENTRYPOINT

ENTRYPOINT instructions are suitable for both single-purpose and multi-mode images where there is a need for a specific command to always run when the container starts.

One of its popular use cases is building wrapper container images that encapsulate legacy programs for containerization, which leverages an ENTRYPOINT instruction to ensure the program will always run.

Using CMD and ENTRYPOINT instructions together

While there are fundamental differences in their operations, CMD and ENTRYPOINT instructions are not mutually exclusive. Several scenarios may call for the use of their combined instructions in a Dockerfile.

A very popular use case for blending them is to automate container startup tasks. In such a case, the ENTRYPOINT instruction can be used to define the executable while using CMD to define parameters.

Let's walk through this with the Darwin Dockerfile, with its specifications as:

```
FROM centos:7
RUN    apt-get update
RUN     apt-get -y install python
COPY ./opt/source code
ENTRYPOINT CMD
```

The image is then built with the command:

```
$ docker build -t darwin .
```

If we run the container without CLI parameters, it will echo the message Hello, Darwin.

Appending the command with a parameter, such as Username, will override the CMD instruction, and execute only the ENTRYPOINT instruction using the CLI parameters as arguments. For example, the command:

```
$ docker run Darwin User_JDArwin
```

will return the output:

```
Hello User_JDArwin
```

This is because the ENTRYPOINT instructions cannot be ignored, while with CMD, the command-line arguments override the instruction.

Using ENTRYPOINT or CMD

Both ENTRYPOINT and CMD are essential for building and running Dockerfiles—it simply depends on your use case. As a general rule of thumb:

- Use ENTRYPOINT instructions when building an executable Docker image using commands that always need to be executed.
- Use CMD instructions when you need an additional set of arguments that act as default instructions until there is explicit command-line usage when a Docker container runs.

A container image requires different elements, including runtime instructions, system tools, and libraries to run an application. To get the best out of a Docker setup, it is strongly advised that your administrators understand various functions, structures, and applications of these instructions, as they are critical functions that help you build images and run containers efficiently.

Related reading

- BMC DevOps Blog
- Docker Security: 14 Best Practices for Securing Docker Containers
- Kubernetes vs Docker: A Quick Comparison
- How To Run MongoDB as a Docker Container
- How Containers Fit in a DevOps Delivery Pipeline
- The State of Containers Today: A Report Summary