Docker is a platform that lets us run and manage applications inside containers. Before we can run an application inside a container, we must [create a Docker image](https://kodekloud.com/blog/create-docker-images/) of the application using Dockerfiles. A Dockerfile is a text file that contains instructions that are executed by Docker to create the Docker image. Each instruction is composed of a directive and an argument. CMD and ENTRYPOINT are two such directives.

In this blog post, we’ll explore CMD and ENTRYPOINT, figure out what they do, understand the key difference between them, and most importantly learn when to use which. Let’s get started!

**Prerequisites**

To follow along with the examples in this post, you need a [code editor](https://kodekloud.com/blog/top-developer-tools/#number-2-code-editor). You also need to have Docker Desktop installed and running on your system. If you don’t have Docker Desktop, you can download it from the link: [Get Docker](https://docs.docker.com/get-docker/?ref=kodekloud.com).

**What Is CMD in Dockerfile?**

Let’s take a closer look at what CMD does inside a Dockerfile by diving into an example.

To follow along, open your code editor and create a directory named docker-demo. Inside this directory, create a file called Dockerfile. You can choose any name you prefer for the directory but the file must be named Dockerfile.

Next, copy and paste the following code into the Dockerfile.

FROM alpine

CMD ["echo", "Hello World!"]

In this Dockerfile, we have two instructions. Each instruction consists of a directive followed by an argument. Note that it’s a good practice to write the directives in UPPERCASE to distinguish them from the arguments.

In our case, we have two directives: FROM and CMD. The FROM directive is followed by the argument alpine, which is the parent image that our custom image is going to be built on top of. The parent image will be downloaded from Docker Hub, a cloud-based storage location for Docker images.

Next, we have **the**CMD**directive that provides the default command** (["echo", "Hello World!"]), **which will be executed when a container is created from the custom Docker image** (which we have not created yet).

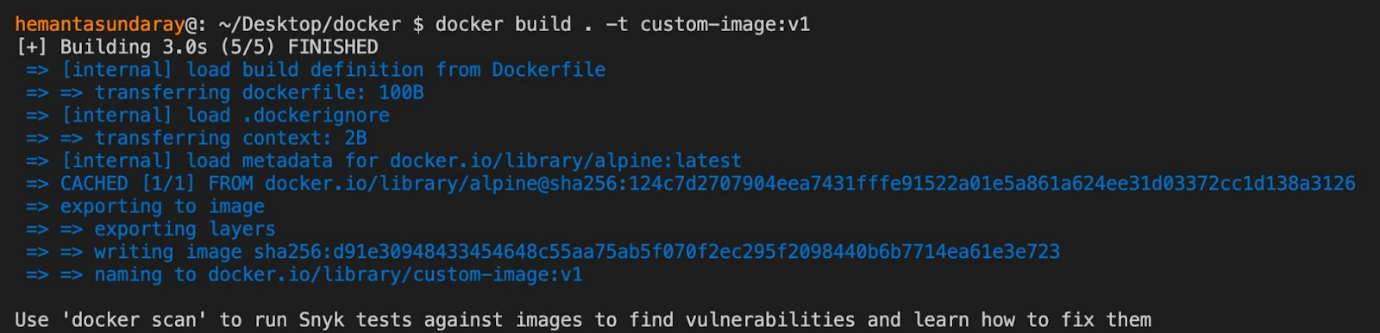
In our example, the argument to CMD is ["echo", "Hello World!"]. The first string echo represents the executable and the second string Hello World! is the argument passed to the executable. When we run a container from the custom Docker image, the command ["echo", "Hello World!"] will be executed and we’ll see Hello World! printed in the terminal.

To see the CMD directive in action, we need to build an image first and then run it as a container. While inside the docker-demo directory, run the following command to create a custom Docker image named custom-image:v1.

docker build . -t custom-image:v1

Don’t forget to use dot (.) after the build command. It indicates that the current directory (i.e. "docker-demo") should be used as the build context.

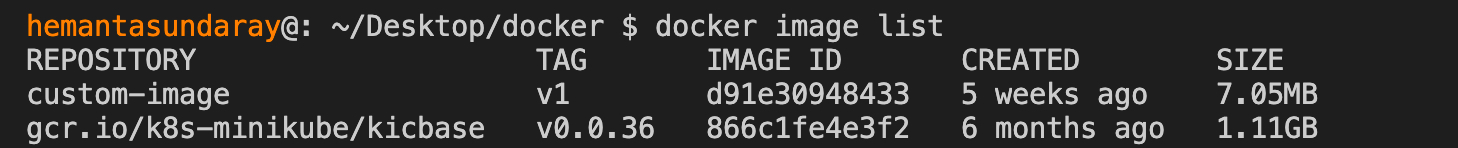
Running the command above will give you the following output:



Next, let's verify that the image has been created by listing the images in our system. Run the following command:

docker image list

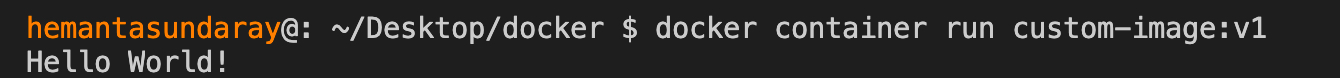
Executing the command above will give you the following output:



As you can see, our custom image, custom-image:v1, is listed. Now, let's run this Docker image as a container by running the following command:

docker container run custom-image:v1

You will see that Hello World! is printed on the console, as shown below:

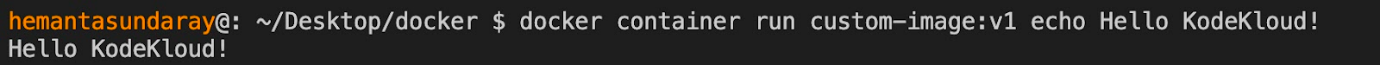


When we ran the Docker image named custom-image:v1 as a container, the command specified by the CMD directive (["echo", "Hello World!"]) was executed, resulting in the Hello World! message being displayed in the terminal.

The important thing to remember about the CMD directive is that **we can override the default command when running the container**. Let's demonstrate this by running the following command:

docker container run custom-image:v1 echo Hello KodeKloud!

After running this command, you’ll see Hello kodeKloud! printed on the terminal, as shown below:



The argument echo Hello KodeKloud! that we passed in the command line took precedence over the default argument ("Hello World!") specified in the Dockerfile.

Now that you have a clear understanding of the CMD directive and its functionality, let's explore the ENTRYPOINT directive and how it differs from CMD.

**What Is ENTRYPOINT in Dockerfile?**

Replace the code inside the Dockerfile with the following:

FROM alpine

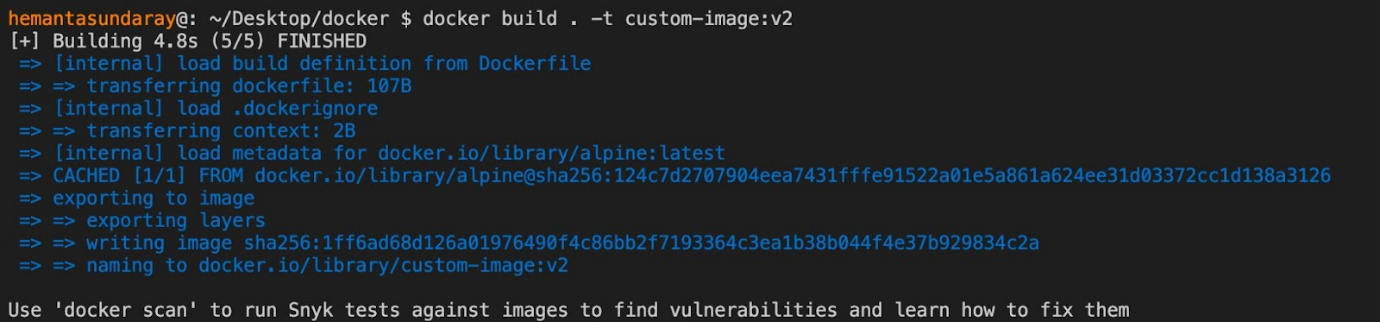
ENTRYPOINT ["echo", "Hello World!"]

As you can see, the only change we have made in the Dockerfile is replacing CMD with ENTRYPOINT.

Next, let’s build a custom Docker image named custom-image:v2 by executing the following command:

docker build . -t custom-image:v2

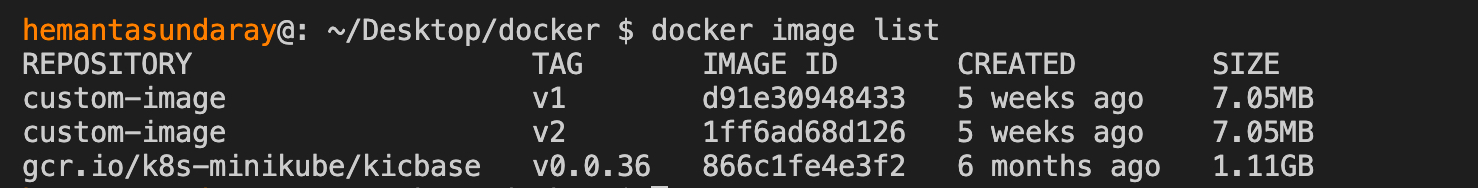
Running the command above will provide you with the following output:



Now, let's verify that the image has been created by listing the images in our system. Run the following command:

docker image list

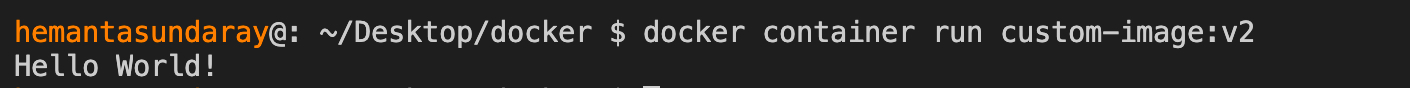
Executing the command above will display the following output:



As you can see, our custom image, custom-image:v2, is listed. Now, let's run this Docker image as a container by running the following command:

docker container run custom-image:v2

You will see that Hello World! is printed on the console, as shown below:

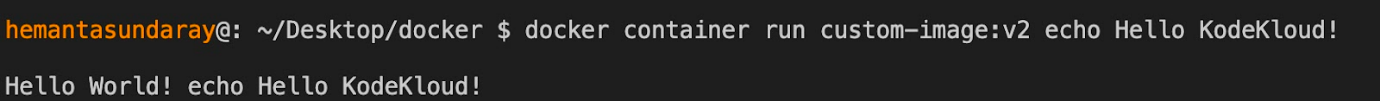


This result is identical to what we obtained with the CMD directive. So, what’s the difference between them?

**Unlike the**CMD**directive, we can’t override the default command of the**ENTRYPOINT**directive**. Let’s see this in action by executing the following command:

docker container run custom-image:v2 echo Hello kodeKloud!

After running the above command, you’ll receive an output like this:



As you can see, the argument ("echo Hello KodeKloud!") we passed in the command line is appended to the default argument ("Hello World!") specified in the Dockerfile. It doesn’t replace the default argument.

Now you should have a clear understanding of the functionality of the ENTRYPOINT directive inside a Dockerfile.

Up until now, we have used the CMD and ENTRYPOINT directives individually in the Dockerfile. However, they can also be used together. Let's explore how in the next section.

**Using CMD & ENTRYPOINT Together**

Replace the contents of your Dockerfile with the following:

FROM alpine

ENTRYPOINT ["echo", "Hello"]

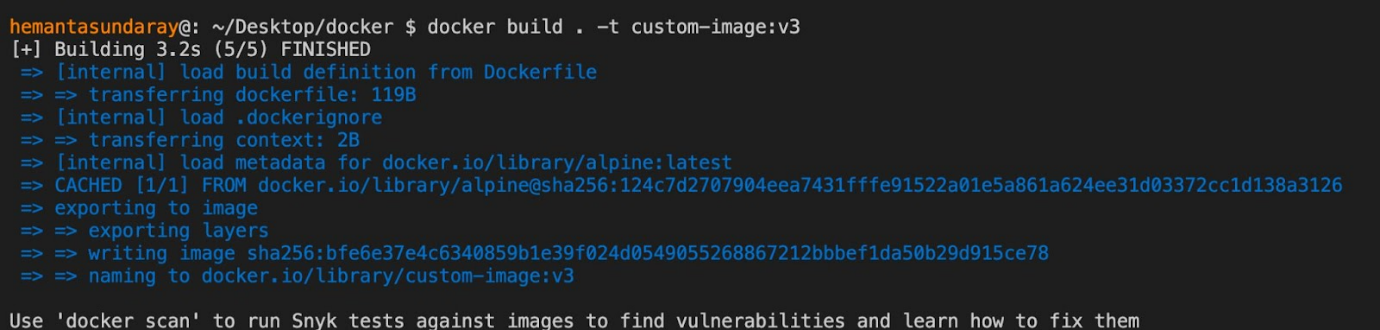
CMD ["World!"]

In this Dockerfile, the ENTRYPOINT directive specifies the default command that will get executed when a container is run from the custom Docker image. The CMD directive, on the other hand,  provides the default argument for the `ENTRYPOINT` directive.

Let’s create a Docker image from this Dockerfile by running the following command:

docker build . -t custom-image:v3

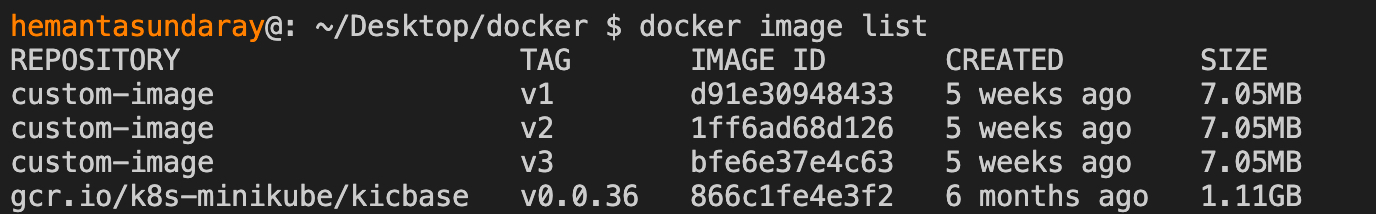
Executing the command above will generate the following output:



Now, let's verify that the image has been created by listing the images in our system. Run the following command:

docker image list

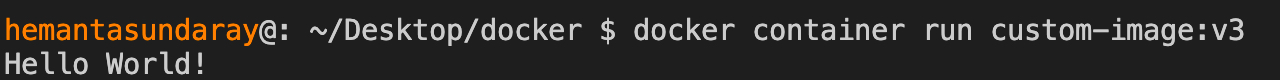
Executing the command above will display the following output:



As you can see, our custom image, custom-image:v3, is listed. Now, let's run this Docker image as a container by running the following command:

docker container run custom-image:v3

You will observe that Hello World! is printed on the console, as shown below:



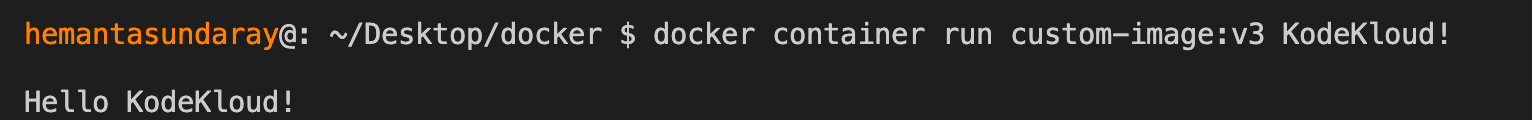
You might be wondering what’re the benefits of using CMD and ENTRYPOINT directives together in the same Dockerfile. Well, **the combination is useful when you not only want to provide a default command to run but also give users the flexibility to provide command-line arguments so that they can get the custom output they want.**

In the Dockerfile above, we can’t override the default command specified by the ENTRYPOINT directive. However, we can override the arguments specified by the CMD directive. For example, we could pass another argument from the command line to override the text World!. The argument we pass will then be appended to Hello.

To see this in action, run the following command:

docker container run custom-image:v3 KodeKloud!

Executing the command above will display the following output:



As you can see, the text World! is replaced by the text KodeKloud! and we see Hello KodeKloud! printed on the terminal.

**Difference Between CMD & ENTRYPOINT**

**Both**CMD**and**ENTRYPOINT**directives serve the same purpose. They provide default commands to be executed when a Docker image is run as a container.**However, they differ in how their arguments behave.

**The main distinction is that the argument passed to the**ENTRYPOINT**directive cannot be overridden, while the argument passed to the**CMD**directive can.**This means that the ENTRYPOINT directive sets a fixed command that cannot be modified when running the container, whereas the CMD directive allows for flexibility in specifying alternative arguments.

Choosing between CMD and ENTRYPOINT depends on your specific requirements. However, it is a common practice to use both directives together. **The**ENTRYPOINT**directive is used to define the default command, while the**CMD**directive is used to pass default arguments to the**ENTRYPOINT**directive.**

Here is a summary of the difference between CMD and ENTRYPOINT in a table form:

|  |  |  |
| --- | --- | --- |
|  | CMD | ENTRYPOINT |
| Provide default commands to be executed when the Docker image is run as a container | Yes | Yes |
| Arguments can be overridden by the user | Yes | No |

**Conclusion**

In this blog post, we learned what purpose CMD and ENTRYPOINT directives server inside a Dockerfile. We also understood how they differ in their functionality and when we should choose each. Now, you should be able to use these directives confidently in your Dockerfile when building Docker images.

Interested in learning more about Docker? Check out the following courses from KodeKloud:

* [Docker for the Absolute Beginner](https://kodekloud.com/courses/docker-for-the-absolute-beginner/?ref=kodekloud.com): This course will help you understand Docker using lectures and demos. You’ll get a hands-on learning experience and coding exercises that will validate your Docker skills.  Additionally, assignments will challenge you to apply your skills in real-life scenarios.
* [Docker Certified Associate Exam Course](https://kodekloud.com/courses/docker-certified-associate-exam-course/?ref=kodekloud.com): This course covers all the required topics from the Docker Certified Associate Exam curriculum. The course offers several opportunities for practice and self-assessment. There are hundreds of research questions in multiple-choice format, practice tests at the end of each section, and multiple mock exams that closely resemble the actual exam pattern.