Introduction

We create Docker containers using **[base]** images. An image can be basic, with nothing but the operating-system fundamentals, or it can consist of a sophisticated pre-built application stack ready for launch.

When we build images with docker, each action taken (i.e. a command executed such as **apt-get install**) forms a new layer on top of the previous one. These base images then can be used to create new containers.

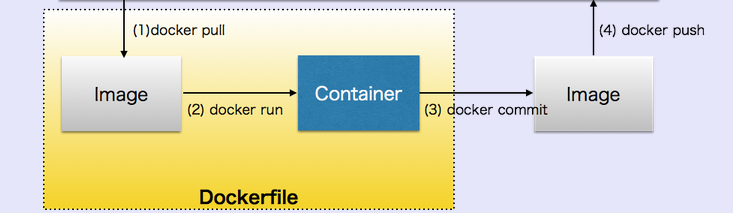


Image source: [Docker](http://hondou.homedns.org/pukiwiki/pukiwiki.php?Docker%20%A4%B6%A4%C3%A4%AF%A4%EA%B8%C0%A4%A6%A4%C8%B2%BF%3F)

In this chapter, we're going to learn how to automate this process via instructions in**Dockerfiles**. A Dockerfile is a text document that contains all the commands we would normally execute manually in order to build a Docker image. By calling docker build from our terminal, we can have Docker build our image executing the instructions successively step-by-step, layer-by-layer, automatically from a source (base) image.

The docker project offers higher-level tools which work together, built on top of some Linux kernel features. The goal is to help developers and system administrators port applications with all of their dependencies included, and get them running across systems and machines headache free.

Docker achieves this by creating safe, LXC (Linux Containers) based environments for applications called docker containers. These containers are created using docker images, which can be built either by executing commands manually or automatically through Dockerfiles.

Dockerfile

Each Dockerfile is a script, composed of various commands (instructions) and arguments listed successively to automatically perform actions on a base image in order to create (or form) a new one. They are used for organizing things and greatly help with deployments by simplifying the process start-to-finish.

Dockerfiles begin with defining an image FROM which the build process starts. Followed by various other methods, commands and arguments (or conditions), in return, provide a new image which is to be used for creating docker containers.

Sample Dockerfile

Here is our Dockerfile we're going to playing with in this chapter. We'll run instructions from this file step by step by uncommenting and commenting each line.

**FROM debian:latest**

**MAINTAINER devops@bogotobogo.com**

**# 1 - RUN**

**RUN apt-get update && DEBIAN\_FRONTEND=noninteractive apt-get install -yq apt-utils**

**RUN DEBIAN\_FRONTEND=noninteractive apt-get install -yq htop**

**RUN apt-get clean**

**# 2 - CMD**

**CMD ["htop"]**

**CMD ["ls", "-l"]**

**# 3 - WORKDIR and ENV**

**WORKDIR /root**

**ENV DZ version1**

**# 4 - ADD**

**ADD run.sh /root/run.sh**

**CMD ["./run.sh"]**

**# 5 - ENTRYPOINT (vs CMD)**

**ENTRYPOINT ["./run.sh"]**

**CMD ["arg1"]**

Dockerfile format

Here is the format of the Dockerfile:

**# Comment**

**INSTRUCTION arguments**

The Instruction is not case-sensitive, however convention is for them to be UPPERCASE in order to distinguish them from arguments more easily.

Docker runs the instructions in a Dockerfile in order. The first instruction must be FROM in order to specify the Base Image from which we are building.

Docker will treat lines that begin with # as a comment.

FROM

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. But it is especially easier to start by pulling an image from the [Public Repositories](http://docs.docker.com/userguide/dockerrepos/#using-public-repositories).

MAINTAINER

**MAINTAINER <name>**

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

Docker build & context

To build an image from a source repository, we create a description file called Dockerfile at the root of our repository. This file will describe the steps to assemble the image.

Then we call docker build with the path of our source repository as the argument:

**Usage: docker build [OPTIONS] PATH | URL | -**

It will build a new image from the source code at PATH.

Here is an example using the current directory (".") as its path:

**$ sudo docker build .**

The path to the source repository defines where to find the context of the build. The build is run by the Docker daemon, not by the CLI, so the whole context must be transferred to the daemon. The Docker CLI reports "Sending build context to Docker daemon" when the context is sent to the daemon.

Note: Avoid using root directory, /, as the root of the source repository. The docker build command will use whatever directory contains the Dockerfile as the build context(including all of its subdirectories). The build context will be sent to the Docker daemon before building the image, which means if we use / as the source repository, the entire contents of our hard drive will get sent to the daemon (and thus to the machine running the daemon). We probably don't want that. In most cases, it's best to put each Dockerfile in an empty directory, and then add only the files needed for building that Dockerfile to that directory. To further speed up the build, we can exclude files and directories by adding a .dockerignore file to the same directory.

The Docker daemon will run our steps one-by-one, committing the result to a new image if necessary, before finally outputting the ID of our new image. The Docker daemon will automatically clean up the context we sent.

Note that each instruction is run independently, and causes a new image to be created.

Whenever possible, Docker will re-use the intermediate images, accelerating docker build significantly.

**$ docker build --help**

**Usage: docker build [OPTIONS] PATH | URL | -**

**Build a new image from the source code at PATH**

**-t, --tag="" Repository name (and optionally a tag) to be applied to the resulting image in case of success**

Base image

Let's download our base image:

**$ docker pull debian:latest**

**debian:latest: The image you are pulling has been verified**

**511136ea3c5a: Pull complete**

**f10807909bc5: Pull complete**

**f6fab3b798be: Pull complete**

**Status: Downloaded newer image for debian:latest**

**k@laptop:~/Documents/demo$ docker images**

**REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE**

**debian latest f6fab3b798be 2 weeks ago 85.1 MB**

In our local working directory, we have only one file, Dockerfile:

**k@laptop:~/Documents/demo$ ls**

**Dockerfile**

Each Dockerfile is a script, composed of various commands (instructions) and arguments listed successively to automatically perform actions on a base image in order to create (or form) a new one. They are used for organizing things and greatly help with deployments by simplifying the process start-to-finish.

docker build 'FROM'

Let's look at the syntax of docker build command:

**$ docker build --help**

**Usage: docker build [OPTIONS] PATH | URL | -**

**Build a new image from the source code at PATH**

**-t, --tag="" Repository name (and optionally a tag) to be applied to the resulting image in case of success**

Dockerfiles begin with defining an image FROM which the build process starts. Followed by various other methods, commands and arguments (or conditions), in return, provide a new image which is to be used for creating docker containers.

**FROM debian:latest**

**MAINTAINER devops@bogotobogo.com**

Let's run docker build command with the two-line Dockerfile:

**k@laptop:~/Documents/demo$ docker build -t bogodevops/demo:v1 .**

**Sending build context to Docker daemon 2.56 kB**

**Sending build context to Docker daemon**

**Step 0 : FROM debian:latest**

**---> f6fab3b798be**

**Step 1 : MAINTAINER k@bogotobogo.com**

**---> Running in 4181b54ab22e**

**---> 511bcbdd59ba**

**Removing intermediate container 4181b54ab22e**

**Successfully built 511bcbdd59ba**

Now if we list the images:

**k@laptop:~/Documents/demo$ docker images**

**REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE**

**bogodevops/demo v1 511bcbdd59ba About a minute ago 85.1 MB**

**debian latest f6fab3b798be 2 weeks ago 85.1 MB**

Note that the path to the source repository defines where to find the context of the build. The build is run by the Docker daemon, not by the CLI, so the whole context must be transferred to the daemon. The Docker CLI reports "Sending build context to Docker daemon" when the context (2.56kB) is sent to the daemon as shown in the output:

**Sending build context to Docker daemon 2.56 kB**

If we send big chuck to the daemon, it will take longer to copy things. For example, if we send duplicate device files(/de/zero) with dd:

**k@laptop:~/Documents/demo$ dd if=/dev/zero of=testimage bs=4096 count=8192**

**8192+0 records in**

**8192+0 records out**

**33554432 bytes (34 MB) copied, 0.118561 s, 283 MB/s**

**k@laptop:~/Documents/demo$ ls**

**Dockerfile testimage**

**k@laptop:~/Documents/demo$ docker build -t bogodevops/demo:v1 .**

**Sending build context to Docker daemon 33.56 MB**

**Sending build context to Docker daemon**

**Step 0 : FROM debian:latest**

**---> f6fab3b798be**

**Step 1 : MAINTAINER k@bogotobogo.com**

**---> Using cache**

**---> 511bcbdd59ba**

**Successfully built 511bcbdd59ba**

Note that the size has been increased from 2.56kb to 33.56MB. That's why the [Docker document](http://docs.docker.com/reference/builder/) gives us a Warning like this:

"Warning Avoid using your root directory, /, as the root of the source repository. The docker build command will use whatever dicrectory contains the Dockerfile as the build context (including all of its subdirectories). The build context will be sent to the Docker daemon before building the image, which means if you use / as the source repository, the entire contents of your hard drive will get sent to the daemon (and thus to the machine running the daemon). You probably don't want that."

Or like this:

"Warning: Do not use your root directory, /, as the PATH as it causes the build to transfer the entire contents of your hard drive to the Docker daemon.

So, we should be aware of the context of our build directory!

Again:

"The build is run by the Docker daemon, not by the CLI. The first thing a build process does is send the entire context (recursively) to the daemon. In most cases, it's best to start with an empty directory as **context** and keep your **Dockerfile** in that directory. Add only the files needed for building the **Dockerfile**"

Also note the the difference in the Step 1 of the two cases:

In the first instance of docker build:

**Step 1 : MAINTAINER k@bogotobogo.com**

**---> Running in 4181b54ab22e**

**---> 511bcbdd59ba**

But in the second run, Docker used cache:

**Step 1 : MAINTAINER k@bogotobogo.com**

**---> Using cache**

**---> 511bcbdd59ba**

When we build a Docker image, it's using a Dockerfile, and every instruction in the Dockerfile is run inside of a container. If that returns successfully, then that container is stored as a new image.

In our case, in Step 0, we created 'f6fab3b798be' which is a hash identifier, and Step 1, we created '511bcbdd59ba' hash.

Note that in our 2nd run (the 'docker run' with 'dd'), the hash is the same. What does this mean? If the instructions in our Dockerfile are the same, Docker uses the cache:

**k@laptop:~/Documents/demo$ docker images -a**

**REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE**

**bogodevops/demo v1 511bcbdd59ba 57 minutes ago 85.1 MB**

**debian latest f6fab3b798be 2 weeks ago 85.1 MB**

**<none> <none> f10807909bc5 2 weeks ago 85.1 MB**

**<none> <none> 511136ea3c5a 17 months ago 0 B**

So, every step along the way, we create a new image. As it succeeds, we'll build a new layer on top of the previous one as we read in an instruction. As this caching allows us to build other environment similar to the previous image without rebuilding from every steps involved.

Dockerfie - RUN

This section is from <http://docs.docker.com/reference/builder/>.

RUN has 2 forms:

**RUN <command> (the command is run in a shell - /bin/sh -c - shell form)**

**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 /bin/sh.

1. **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"].
2. **Note**: The exec form is parsed as a JSON array, which means that you must use double-quotes (") around words not single-quotes (').
3. **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" ].

Dockerfile 'RUN' sample

Here is our Dockerfile we're going to playing with in this chapter. We'll run instructions from this file step by step by uncommenting and commenting each line.

FROM debian:latest

MAINTAINER devops@bogotobogo.com

# 1 - RUN

RUN apt-get update && DEBIAN\_FRONTEND=noninteractive apt-get install -yq apt-utils

RUN DEBIAN\_FRONTEND=noninteractive apt-get install -yq htop

RUN apt-get clean

We have three instructions for RUN, and each of these instruction will create a new container, and at the completion of each instruction, it will become an image.

The following enviroment setting is to block any terminal output caused by some errors:

DEBIAN\_FRONTEND=noninteractive

The 2nd instruction, htop is to monitor processes in linux system. Then, we removes all packages from the package cache using apt-get clean.

Let's run docker build with v2 instead of v1:

$ docker build -t bogodevops/demo:v2 .

Sending build context to Docker daemon 33.56 MB

Sending build context to Docker daemon

Step 0 : FROM debian:latest

---> f6fab3b798be

Step 1 : MAINTAINER k@bogotobogo.com

---> Using cache

---> 511bcbdd59ba

Step 2 : RUN apt-get update && DEBIAN\_FRONTEND=noninteractive apt-get install -yq apt-utils

---> Running in 10ffa5b21a27

...

Setting up apt-utils (0.9.7.9+deb7u6) ...

---> e6e2c03b8efc

Removing intermediate container 10ffa5b21a27

Step 3 : RUN DEBIAN\_FRONTEND=noninteractive apt-get install -yq htop

---> Running in 2fe900ff207c

...

Setting up htop (1.0.1-1) ...

---> fac6e3168cfe

Removing intermediate container 2fe900ff207c

Step 4 : RUN apt-get clean

---> Running in 990373d72cc9

---> 327d400a953c

Removing intermediate container 990373d72cc9

Successfully built 327d400a953c

Listing images:

k@laptop:~/Documents/demo$ docker images -a

REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE

bogodevops/demo v2 327d400a953c 7 minutes ago 96.16 MB

<none> <none> fac6e3168cfe 7 minutes ago 96.16 MB

<none> <none> e6e2c03b8efc 7 minutes ago 95.12 MB

bogodevops/demo v1 511bcbdd59ba 2 hours ago 85.1 MB

debian latest f6fab3b798be 2 weeks ago 85.1 MB

<none> <none> f10807909bc5 2 weeks ago 85.1 MB

<none> <none> 511136ea3c5a 17 months ago 0 B

As we discussed in the previous chapter, if we run this again, it will be completed much faster thanks to caching:

k@laptop:~/Documents/demo$ docker build -t bogodevops/demo:v2 .

Sending build context to Docker daemon 33.56 MB

Sending build context to Docker daemon

Step 0 : FROM debian:latest

---> f6fab3b798be

Step 1 : MAINTAINER k@bogotobogo.com

---> Using cache

---> 511bcbdd59ba

Step 2 : RUN apt-get update && DEBIAN\_FRONTEND=noninteractive apt-get install -yq apt-utils

---> Using cache

---> e6e2c03b8efc

Step 3 : RUN DEBIAN\_FRONTEND=noninteractive apt-get install -yq htop

---> Using cache

---> fac6e3168cfe

Step 4 : RUN apt-get clean

---> Using cache

---> 327d400a953c

Successfully built 327d400a953c

docker run - launching container

k@laptop:~/Documents/demo$ docker run -it --rm bogodevops/demo:v2 /bin/bash

root@cf6430ffba1b:/# exit

exit

If we drop the :v2 tag in the command:

k@laptop:~/Documents/demo$ docker run -it --rm bogodevops/demo /bin/bash

Unable to find image 'bogodevops/demo' locally

Pulling repository bogodevops/demo

2014/11/24 18:55:36 Error: image bogodevops/demo not found

So, to make it work, we need to build default as latest:

k@laptop:~/Documents/demo$ docker build -t bogodevops/demo .

Now, if look at the images:

k@laptop:~/Documents/demo$ docker images -a

REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE

bogodevops/demo v2 327d400a953c 32 minutes ago 96.16 MB

bogodevops/demo latest 327d400a953c 32 minutes ago 96.16 MB

we have a new bogodevops/demo image tagged as 'latest'. So, from now on, we can execute docker run without the 'tag' since it'll look for 'latest' tag by default:

k@laptop:~/Documents/demo$ docker run -it --rm bogodevops/demo /bin/bash

root@88d48b65ebd7:/#

Now, we're in our Docker container for Debian, and htop has been installed.

root@88d48b65ebd7:/# htop

htop.png

root@88d48b65ebd7:/# exit

exit

We should not see any container:

k@laptop:~/Documents/demo$ docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

No containers are hanging around!

Dockerfie - CMD

This section is from <http://docs.docker.com/reference/builder/>.

CMD has 3 forms:

CMD ["executable","param1","param2"] (exec form, this is the preferred form)

1. CMD ["param1","param2"] (as default parameters to ENTRYPOINT)
2. CMD command param1 param2 (shell form)

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

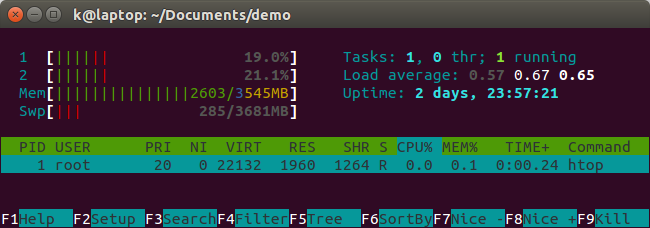
The main purpose of a CMD is to provide defaults for an executing container. These defaults can include an executable, or they can omit the executable, in which case we must specify an ENTRYPOINT instruction as well.

1. **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.
2. **Note**: The exec form is parsed as a JSON array, which means that you must use double-quotes (") around words not single-quotes (').
3. **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" ].
4. Dockerfile 'CMD' sample
5. Here is our Dockerfile that we're going to play with in this chapter. We'll run instructions from this file step by step by uncommenting and commenting each line.
6. **FROM debian:latest**
7. **MAINTAINER devops@bogotobogo.com**
8. **# 1 - RUN**
9. **RUN apt-get update && DEBIAN\_FRONTEND=noninteractive apt-get install -yq apt-utils**
10. **RUN DEBIAN\_FRONTEND=noninteractive apt-get install -yq htop**
11. **RUN apt-get clean**
12. **# 2 - CMD**
13. **CMD ["htop"]**
14. We have one instruction for CMD, and at the completion of the CMD, it will become an image.
15. **k@laptop:~/Documents/demo$ docker build -t bogodevops/demo .**
16. **Sending build context to Docker daemon 33.56 MB**
17. **Sending build context to Docker daemon**
18. **Step 0 : FROM debian:latest**
19. **---> f6fab3b798be**
20. **Step 1 : MAINTAINER k@bogotobogo.com**
21. **---> Using cache**
22. **---> 511bcbdd59ba**
23. **Step 2 : RUN apt-get update && DEBIAN\_FRONTEND=noninteractive apt-get install -yq apt-utils**
24. **---> Using cache**
25. **---> e6e2c03b8efc**
26. **Step 3 : RUN DEBIAN\_FRONTEND=noninteractive apt-get install -yq htop**
27. **---> Using cache**
28. **---> fac6e3168cfe**
29. **Step 4 : RUN apt-get clean**
30. **---> Using cache**
31. **---> 358b5cc4b9fa**
32. **Step 5 : CMD htop**
33. **---> Running in d31a73253846**
34. **---> b64547129d16**
35. **Removing intermediate container d31a73253846**
36. **Successfully built b64547129d16**

But unlike in the previous chapter where we ran htop explicitly within the container, this time, it becomes a default environment.

So, even though we issue docker run without passing in any command, we have htoprunning automatically when the container is created:

1. **k@laptop:~/Documents/demo$ docker run -it --rm bogodevops/demo**



We get the htop as soon as we're in the container. It's given us as an environment.

If we pass in /bin/bash, then we'll have bash instead of htop:

1. **k@laptop:~/Documents/demo$ docker run -it --rm bogodevops/demo /bin/bash**
2. **root@00e40007ed7d:/# exit**
3. **exit**

Before we start new thing, we need to remove 'testimage' in our directory:

1. **k@laptop:~/Documents/demo$ ls**
2. **Dockerfile testimage**
3. **k@laptop:~/Documents/demo$ rm testimage**

Then, let's switch our CMD instruction to CMD ["ls", "l"]. Here is our new Dockerfile:

1. **FROM debian:latest**
2. **MAINTAINER k@bogotobogo.com**
3. **# 1 - RUN**
4. **RUN apt-get update && DEBIAN\_FRONTEND=noninteractive apt-get install -yq apt-utils**
5. **RUN DEBIAN\_FRONTEND=noninteractive apt-get install -yq htop**
6. **RUN apt-get clean**
7. **# 2 - CMD**
8. **#CMD ["htop"]**
9. **CMD ["ls", "l"]**

Build a new image with the new CMD ["ls", "l"]:

1. **k@laptop:~/Documents/demo$ docker build -t bogodevops/demo .**
2. **Sending build context to Docker daemon 2.56 kB**
3. **Sending build context to Docker daemon**
4. **Step 0 : FROM debian:latest**
5. **---> f6fab3b798be**
6. **Step 1 : MAINTAINER k@bogotobogo.com**
7. **---> Using cache**
8. **---> 511bcbdd59ba**
9. **Step 2 : RUN apt-get update && DEBIAN\_FRONTEND=noninteractive apt-get install -yq apt-utils**
10. **---> Using cache**
11. **---> e6e2c03b8efc**
12. **Step 3 : RUN DEBIAN\_FRONTEND=noninteractive apt-get install -yq htop**
13. **---> Using cache**
14. **---> fac6e3168cfe**
15. **Step 4 : RUN apt-get clean**
16. **---> Using cache**
17. **---> 358b5cc4b9fa**
18. **Step 5 : CMD ls -l**
19. **---> Running in 717df1a3baa2**
20. **---> d2f3de97b6ef**
21. **Removing intermediate container 717df1a3baa2**
22. **Successfully built d2f3de97b6ef**

If we go in our container, it will automatically gives the output from ls -a:

1. **k@laptop:~/Documents/demo$ docker run -it --rm bogodevops/demo**
2. **total 68**
3. **drwxr-xr-x 2 root root 4096 Nov 5 21:37 bin**
4. **drwxr-xr-x 2 root root 4096 Sep 21 18:17 boot**
5. **drwxr-xr-x 4 root root 360 Nov 25 05:25 dev**
6. **drwxr-xr-x 32 root root 4096 Nov 25 05:25 etc**
7. **drwxr-xr-x 2 root root 4096 Sep 21 18:17 home**
8. **drwxr-xr-x 8 root root 4096 Nov 25 02:27 lib**
9. **drwxr-xr-x 2 root root 4096 Nov 5 21:33 lib64**
10. **drwxr-xr-x 2 root root 4096 Nov 5 21:31 media**
11. **drwxr-xr-x 2 root root 4096 Sep 21 18:17 mnt**
12. **drwxr-xr-x 2 root root 4096 Nov 5 21:31 opt**
13. **dr-xr-xr-x 253 root root 0 Nov 25 05:25 proc**
14. **drwx------ 2 root root 4096 Nov 5 21:31 root**
15. **drwxr-xr-x 5 root root 4096 Nov 5 21:37 run**
16. **drwxr-xr-x 2 root root 4096 Nov 5 21:37 sbin**
17. **drwxr-xr-x 2 root root 4096 Jun 10 2012 selinux**
18. **drwxr-xr-x 2 root root 4096 Nov 5 21:31 srv**
19. **dr-xr-xr-x 13 root root 0 Nov 25 05:25 sys**
20. **drwxrwxrwt 2 root root 4096 Nov 5 21:37 tmp**
21. **drwxr-xr-x 16 root root 4096 Nov 25 02:27 usr**
22. **drwxr-xr-x 17 root root 4096 Nov 25 02:27 var**

Dockerfie - WORKDIR & ENV

This section is from http://docs.docker.com/reference/builder/.

WORKDIR /path/to/workdir

The WORKDIR instruction sets the working directory for any RUN, CMD and ENTRYPOINT instructions that follow it in the Dockerfile.

WORKDIR & ENV - sample

Here is our updated Dockerfile:

**FROM debian:latest**

**MAINTAINER k@bogotobogo.com**

**# 1 - RUN**

**RUN apt-get update && DEBIAN\_FRONTEND=noninteractive apt-get install -yq apt-utils**

**RUN DEBIAN\_FRONTEND=noninteractive apt-get install -yq htop**

**RUN apt-get clean**

**# 2 - CMD**

**#CMD ["htop"]**

**#CMD ["ls", "-l"]**

**# 3 - WORKDIR and ENV**

**WORKDIR /root**

**ENV DZ version1**

Let's build the image:

**$ docker build -t bogodevops/demo .**

**Sending build context to Docker daemon 2.56 kB**

**Sending build context to Docker daemon**

**Step 0 : FROM debian:latest**

**---> f6fab3b798be**

**Step 1 : MAINTAINER k@bogotobogo.com**

**---> Using cache**

**---> 511bcbdd59ba**

**Step 2 : RUN apt-get update && DEBIAN\_FRONTEND=noninteractive apt-get install -yq apt-utils**

**---> Using cache**

**---> e6e2c03b8efc**

**Step 3 : RUN DEBIAN\_FRONTEND=noninteractive apt-get install -yq htop**

**---> Using cache**

**---> fac6e3168cfe**

**Step 4 : RUN apt-get clean**

**---> Using cache**

**---> 358b5cc4b9fa**

**Step 5 : WORKDIR /root**

**---> Running in 2ce95d5fede1**

**---> a205c4badd68**

**Removing intermediate container 2ce95d5fede1**

**Step 6 : ENV DZ version1**

**---> Running in 6ac629a3506b**

**---> 6f9de0a5099f**

**Removing intermediate container 6ac629a3506b**

**Successfully built 6f9de0a5099f**

**$**

Here we're using repository name (tag) for the image, and the dot('.') indicates our Dockerfile is in local directory.

What images do we have now?

**$ docker images**

**REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE**

**bogodevops/demo latest 6f9de0a5099f About a minute ago 96.16 MB**

**<none> <none> d2f3de97b6ef About an hour ago 96.16 MB**

**<none> <none> e171cd1dd9e7 About an hour ago 96.16 MB**

**<none> <none> b64547129d16 About an hour ago 96.16 MB**

**bogodevops/demo v2 358b5cc4b9fa 2 hours ago 96.16 MB**

**bogodevops/demo v1 511bcbdd59ba 7 hours ago 85.1 MB**

**debian latest f6fab3b798be 2 weeks ago 85.1 MB**

Note the images tagged with <none>. These are the images which had no tag, and left behind when a new image is tagged as 'latest'.

Now we're going to run a new container and run bash inside of it:

**$ docker run -it --rm bogodevops/demo /bin/bash**

We can check the WORKDIR and ENV settings in our Dockerfile:

**root@52a10702207c:~# pwd**

**/root**

**root@52a10702207c:~# echo $DZ**

**version1**

**root@52a10702207c:~# exit**

**exit**

OK. We've got what we expected.

Dockerfie - ADD

**ADD <src>... <dest>**

The ADD instruction copies new files, directories or remote file URLs from <src> and adds them to the filesystem of the container at the path <dest>.

Multiple <src> resource may be specified but if they are files or directories then 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 most command line uses this should act as expected, for example:

**ADD hom\* /mydir/ # adds all files starting with "hom"**

**ADD hom?.txt /mydir/ # ? is replaced with any single character**

The <dest> is the absolute path to which the source will be copied inside the destination container.

Here is our new Dockerfile:

**FROM debian:latest**

**MAINTAINER k@bogotobogo.com**

**# 1 - RUN**

**RUN apt-get update && DEBIAN\_FRONTEND=noninteractive apt-get install -yq apt-utils**

**RUN DEBIAN\_FRONTEND=noninteractive apt-get install -yq htop**

**RUN apt-get clean**

**# 2 - CMD**

**#CMD ["htop"]**

**#CMD ["ls", "-l"]**

**# 3 - WORKDIR and ENV**

**WORKDIR /root**

**ENV DZ version1**

**# 4 - ADD**

**ADD run.sh /root/run.sh**

**CMD ["./run.sh"]**

The run.sh should be referencing current working directory in our local machine.

Here is the run.sh script:

**#!/bin/sh**

**echo "The current directory : $(pwd)"**

**echo "The DZ variable : $DZ"**

**echo "There are $# arguments: $@"**

We should build the image:

**$ docker build -t bogodevops/demo .**

**Sending build context to Docker daemon 3.584 kB**

**Sending build context to Docker daemon**

**Step 0 : FROM debian:latest**

**---> f6fab3b798be**

**Step 1 : MAINTAINER k@bogotobogo.com**

**---> Using cache**

**---> 511bcbdd59ba**

**Step 2 : RUN apt-get update && DEBIAN\_FRONTEND=noninteractive apt-get install -yq apt-utils**

**---> Using cache**

**---> e6e2c03b8efc**

**Step 3 : RUN DEBIAN\_FRONTEND=noninteractive apt-get install -yq htop**

**---> Using cache**

**---> fac6e3168cfe**

**Step 4 : RUN apt-get clean**

**---> Using cache**

**---> 358b5cc4b9fa**

**Step 5 : WORKDIR /root**

**---> Using cache**

**---> a205c4badd68**

**Step 6 : ENV DZ version1**

**---> Using cache**

**---> 6f9de0a5099f**

**Step 7 : ADD run.sh /root/run.sh**

**---> b4a525cd8f8c**

**Removing intermediate container 81ed15e4425d**

**Step 8 : CMD ./run.sh**

**---> Running in 7f9dad902cff**

**---> b17ff9ebc8f8**

**Removing intermediate container 7f9dad902cff**

**Successfully built b17ff9ebc8f8**

Then, run a container with no command:

**$ docker run -it --rm bogodevops/demo**

**The current directory : /root**

**The DZ variable : version1**

**There are 0 arguments:**

If we add a command to docker run, we get this:

**$ docker run -it --rm bogodevops/demo ./run.sh Hello bogotobogo**

**The current directory : /root**

**The DZ variable : version1**

**There are 2 arguments: Hello bogotobogo**

Dockerfie - ENTRYPOINT

ENTRYPOINT has two forms:

1. ENTRYPOINT ["executable", "param1", "param2"] (the preferred exec form)
2. ENTRYPOINT command param1 param2 (shell form)

An ENTRYPOINT allows us to configure a container that will run as an executable.

For example, the following will start nginx with its default content, listening on port 80:

**docker run -i -t --rm -p 80:80 nginx**

Command line arguments to docker run <image> will be appended after all elements in an exec form ENTRYPOINT, and will override all elements specified using CMD. This allows arguments to be passed to the entry point, i.e., docker run <image> -d will pass the -dargument to the entry point. We can override the ENTRYPOINT instruction using thedocker run --entrypoint flag.

Here is our updated Dockerfile which includes ENTRYPOINT:

**FROM debian:latest**

**MAINTAINER k@bogotobogo.com**

**# 1 - RUN**

**RUN apt-get update && DEBIAN\_FRONTEND=noninteractive apt-get install -yq apt-utils**

**RUN DEBIAN\_FRONTEND=noninteractive apt-get install -yq htop**

**RUN apt-get clean**

**# 2 - CMD**

**#CMD ["htop"]**

**#CMD ["ls", "-l"]**

**# 3 - WORKDIR and ENV**

**WORKDIR /root**

**ENV DZ version1**

**# 4 - ADD**

**ADD run.sh /root/run.sh**

**#CMD ["./run.sh"]**

**# 5 - ENTRYPOINT (vs CMD)**

**ENTRYPOINT ["./run.sh"]**

**CMD ["arg1"]**

Build our image again:

**$ docker build -t bogodevops/demo .**

**Sending build context to Docker daemon 3.584 kB**

**Sending build context to Docker daemon**

**Step 0 : FROM debian:latest**

**---> f6fab3b798be**

**Step 1 : MAINTAINER k@bogotobogo.com**

**---> Using cache**

**---> 511bcbdd59ba**

**Step 2 : RUN apt-get update && DEBIAN\_FRONTEND=noninteractive apt-get install -yq apt-utils**

**---> Using cache**

**---> e6e2c03b8efc**

**Step 3 : RUN DEBIAN\_FRONTEND=noninteractive apt-get install -yq htop**

**---> Using cache**

**---> fac6e3168cfe**

**Step 4 : RUN apt-get clean**

**---> Using cache**

**---> 358b5cc4b9fa**

**Step 5 : WORKDIR /root**

**---> Using cache**

**---> a205c4badd68**

**Step 6 : ENV DZ version1**

**---> Using cache**

**---> 6f9de0a5099f**

**Step 7 : ADD run.sh /root/run.sh**

**---> Using cache**

**---> c7ecd3c5437e**

**Step 8 : ENTRYPOINT ./run.sh**

**---> Running in 2f84e971ba97**

**---> 9c6bcba955d9**

**Removing intermediate container 2f84e971ba97**

**Step 9 : CMD arg1**

**---> Running in 42b50c05e9f8**

**---> ff6f9d2ad977**

**Removing intermediate container 42b50c05e9f8**

**Successfully built ff6f9d2ad977**

Container run without any argument:

**$ docker run -it --rm bogodevops/demo**

**The current directory : /root**

**The DZ variable : version1**

**There are 1 arguments: arg1**

It still runs run.sh shell. If we pass in something like /bin/bash:

**$ docker run -it --rm bogodevops/demo /bin/bash**

**The current directory : /root**

**The DZ variable : version1**

**There are 1 arguments: /bin/bash**

Still it runs run.sh file while /bin/bash was passed in as an argument.

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. We can only use environment variables explicitly set in the Dockerfile. For example:

ENV DIRPATH /path

WORKDIR $DIRPATH/$DIRNAME

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

ENV <key> <value>

The ENV instruction sets the environment variable <key> to the value <value>. This value will be passed to all future RUN instructions. This is functionally equivalent to prefixing the command with <key>=<value>

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

Note: One example where this can cause unexpected consequences, is setting ENV DEBIAN\_FRONTEND noninteractive. Which will persist when the container is run interactively; for example: docker run -t -i image bash.