[Dockerfile - Build Docker images automatically V - WORKDIR, ENV, ADD, and ENTRYPOINT](http://www.bogotobogo.com/DevOps/Docker/Docker_Dockerfile_to_build_images_automatically_5_WORKDIR_ENV_ADD_ENTRYPOINT.php)

Continued from ...

Continued from [Dockerfile - Build Docker images automatically IV - CMD](http://www.bogotobogo.com/DevOps/Docker/Docker_Dockerfile_to_build_images_automatically_4_CMD.php)

In this chapter, we're going to learn more on how to automate this process via instructions in Dockerfiles. We'll be focusing on WORKDIR, ENV, ADD, and ENTRYPOINT.

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 ENTRYPOINTinstructions that follow it in the Dockerfile.

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.

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 -d argument to the entry point. We can override the ENTRYPOINT instruction using the docker 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.