QA Consulting

# Docker Dockerfile Instructions

**TRAINING MATERIALS - MODULE HANDOUT**

---

**Contacts**

**robert.crutchley@qa.com**
*team.qac.all.trainers@qa.com*

*www.consulting.qa.com*

# Contents



## Overview

Dockerfiles are built up of instructions that are ran in order, they help us to build and configure Docker Images in very specific ways. The format is as follows:

```
# comment
INSTRUCTION arguments
```

- It's convention to have the instruction in capital letters.
- Any Dockerfile that you create **must start with a FROM instruction**.
- Any line starting with a **#** is considered as a comment

## Instructions

### FROM

The FROM instruction creates a new build stage. Except for the ARG instruction, this instruction must be at the beginning of a Dockerfile. It is used to set the Base Image that the Docker image is going to be created from.

```
FROM [image]:[TAG]
```

```
FROM java:8
```

The FROM instruction can appear multiple times in a Dockerfile. This can be for creating multiple images or for using previous build stages as dependencies for others. For instance one image can be created for compiling the application and creating an executable, the other image for running the application.

### RUN

We use the RUN instruction to run shell commands on the intermediate containers. The default shell that's used is **/bin/sh**.

```
RUN [COMMAND]
```

```
RUN apt update
```

This instruction lends itself well to how Docker Images are built because the build progress of the image after each command you run is effectively saved, so if one fails then the build can start where it was last successful.

### CMD

To provide a default execution for a container, or main process, we use the CMD instruction. There can only be one of these in a Dockerfile, if there is more than one then the last one in the Dockerfile will be the only one that takes effect.

```
CMD [COMMAND]
```

```
CMD ["/bin/ping", "google.com"]
```

CMD can be used in conjunction with the ENTRYPOINT instruction to set the arguments for a command. This is ideal if you plan on running the same executable every time you run a container, as it allows you to have default arguments which can be altered by the docker run command if necessary.

### LABEL

Metadata can be added to a Docker image by using this instruction. Metadata for Docker Images follows a key-value pair format.

```
LABEL version="1.0"
LABEL description="A Docker Image."
```

Images can have more than one label, you can view the labels for an image by using the docker inspect command. Keep in mind that you will need to have the image in your local registry to be able to inspect it.

```
docker inspect [IMAGE]
```

```
docker inspect nginx
```

Metadata can have many uses cases, such as setting versions, descriptions etc for access by other tools. If you ever feel the need to add some extra information to a Docker Image that isn't already available then labels are good solution for it.

### EXPOSE

This is purely an informative instruction for Docker to show which port the application is going to be listening on, it doesn't actually expose or "open" a port from the container to the host, that's what publishing (-p) is for in the docker run command. By default Docker will assume that the protocol is TCP, but UDP can be specified if that's the protocol that your application is going to be using.

```
EXPOSE [PORT]/[PROTOCOL]
```

```
EXPOSE 80
```

Exposing to UDP:

```
EXPOSE 80/udp
```

If you want to expose both TCP and UDP then you can just do that over two lines:

```
EXPOSE 80/tcp
EXPOSE 80/udp
```

### ENV

Setting environment variables can be done with ENV. Once you have created an environment variable in the Dockerfile, subsequent commands afterwards will be running in the same in environment so they will be able to access it. The whole string beyond the space after the key will be taken as the value.

```
ENV [KEY] [VALUE]
```

```
ENV JENKINS_HOME /jenkins-home
```

After creating an image, the variables can be viewed with **docker inspect** command.

### ADD

New Files, Directories and even remote file URLs can be added into the filesystem of the Docker Image by providing the source and destination.

```
ADD [SOURCE] [DESTINATION]
```

```
# add a local file such as an application, relative to the context
ADD ./app.py /opt/application/app.py
```

```
# add an entire folder
ADD . /opt/application/
```

```
# add a remote file from a URL
ADD https://remote-server/file.txt /remote-file.txt
```

```
# add a tar file (relative to the context) and extract the contents to a folder
ADD ./application.tar.gz /opt/application
```

A common use case for needing to add a file to the images file system is for the application itself that you are going to want to run in the container. ADD is very similar to the COPY command,

### COPY

Files and Directories can be copied into the image file system with this instruction, sounds familiar right? So which one to use, ADD or COPY? The answer is fairly subjective however most of the time you will want to be using COPY just because it is more explicit about what it is doing, copying a file or directory.

The main difference between ADD and COPY is that ADD can add files from two more sources, a URL and a extracting a TAR file to a destination. Unless you want to extract a TAR file you may as well use COPY.

COPY works just the same as ADD, by providing a source and a destination for copying files.

```
COPY [SOURCE] [DESTINATION]
```

```
# copy a local file such as an application, relative to the context
COPY ./app.py /opt/application/app.py
```

```
# copy an entire folder
COPY . /opt/application/
```

### ENTRYPOINT

This instruction is more for running a binary on the container that isn't going to change, effectively running a container as an executable, you can use it in a similar way to CMD.

```
ENTRYPOINT [COMMAND]
```

```
ENTRYPOINT ["/bin/echo", "Hello"]
```

You provide additional arguments to the ENTRYPOINT instruction by utilising CMD, the idea is that the CMD default arguments are intended to be changed whereas the ENTRYPOINT stays the same. CMD parameters will be appended to the ENTRYPOINT.

```
# produces: Hello World
ENTRYPOINT ["/bin/echo", "Hello"]
CMD ["World"]
```

The CMD parameters can be overridden by appending them to the end of the docker run command.

```
# produces: Hello from docker run command
docker run test from docker run command
```

### VOLUME

You can create mount points in containers with a given name using this instruction, so that the data can be stored outside the container. There are many reasons for this, such as application logs that need to be accessed by a logging metrics service or any other files that need to be persisted once the container has been stopped and removed.
You can provide more than one volume, separated by spaces.

```
VOLUME [VOLUMES]
```

```
VOLUME /test
```

For the example above, any files that are put into /test, you will be able to find on the host in /var/lib/docker/volumes/[VOLUME_ID]/_data. You can find what the VOLUME_ID is by running docker inspect [CONTAINER_ID], the value will be under Mounts.

### USER

Set the user which you are running Dockerfile instructions with. If you are intending on having a user other than root run your application in the Docker Container then it will help to run commands as that user where possible to avoid issues with file permissions. The user needs to exist before you can use them, so they would be created in a previous instruction.

```
USER [USER]
```

```
USER jenkins
```

### WORKDIR

When running a Dockerfile, all instructions that interact with the filesystem in the container such as ADD, COPY, ENTRYPOINT, CMD and RUN, have a working directory of / by default. If you are going to to be running multiple instructions in the same directory on the container then you can use this to make things easier for yourself.

```
WORKDIR [DIRECTORY]
```

```
WORKDIR /opt/application
```

### ARG

Dockerfiles can be more generic and flexible by using arguments, a common property that is changes when you are building software is the version. For example, when copying an application file into a Docker image it may be named like this: application-1.0.0.py. The next time you build the application it might be at a different version, with the application file named differently also; application-1.0.1.py. We can use ARG to copy the file as application-${VERSION}.py so that the version can be provided when we build the image.

ARG instructions require at least the variable name.

```
ARG [VARIABLE_NAME]
```

```
ARG PYTHON_VERSION
```

You can set a default value for your argument when you create it.

```
ARG [VARIABLE_NAME]=[VALUE]
```

```
ARG PYTHON_VERSION=3.6
```

Arguments can be overridden with the docker build command, when you are creating the image.

```
docker build --build-arg [VARIABLE_NAME]=[VALUE] [CONTEXT]
```

```
docker build --build-arg PYTHON_VERSION=2.7 .
```

Throughout the Dockerfile after the ARG has been declared, the value can be accessed with a dollar sign, optionally surrounded by brackets: $VARIABLE_NAME or ${VARIABLE_NAME}.
The example Dockerfile below will build from python:3.6 by default, but another version could be passed as a build-arg, making it build from python:2.7 for instance.

```
ARG PYTHON_VERSION=3.6
FROM python:${PYTHON_VERSION}
ENTRYPOINT ["/bin/ping", "google.com"]
```

## Tasks

### Overview

The aim of this exercise is to get you more familiar with Dockerfile instructions, utilising most of the instructions listed in this handout. We will be creating two very basic python HTTP servers for applications to run in our Docker containers. There will be a server for versions 2 and 3 of Python that serve a simple static website. Our Dockerfile will be configured to handle both versions.

Start by creating a new directory with an empty Dockerfile and continue the rest of this exercise in that folder.

### Static Content

Create an index.html file. This is basic webpage that our application will serve.

index.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Python: Simple HTTP Server</title>
</head>
<body>
       <h3>Python HTTP Server</h3>
   <p>Running on Python Version: {{PYTHON_VERSION}}</p>
</body>
</html>
```

#### Python Servers

Create a python-server-2.7.15.py file. This is a Python script that serves the contents of the folder that it is running in on port 9000.

python-server-2.7.15.py

```
# only works with Python 2
import SimpleHTTPServer
import SocketServer

PORT = 9000
Handler = SimpleHTTPServer.SimpleHTTPRequestHandler
httpd = SocketServer.TCPServer(("", PORT), Handler)
httpd.serve_forever()
```

Create a python-server-3.6.8.py file. This script does the same thing as the first one but notice how the imports are different at the top of the script compared to the other script because it is only compatible with Python 3.

python-server-3.6.8.py

```
# only works with Python 3
import http.server
import socketserver

PORT = 9000
Handler = http.server.SimpleHTTPRequestHandler
httpd = socketserver.TCPServer(("", PORT), Handler)
print("serving at port", PORT)
httpd.serve_forever()
```

### Dockerfile

Complete the Dockerfile.

Dockerfile

```
#   an argument for the python version
#   by default this is for version 3.6.8, but it can be modified
#   by the docker build command:
#   docker build -t python-server \
#       --build-arg PYTHON_VERSION=2.7 .
ARG PYTHON_VERSION=3.6.8
# the base image to build from which is ready to run
# Python code immediately
FROM python:${PYTHON_VERSION}
# the working directory for docker instructions has
# been changed to where our application is going
# to be installed
WORKDIR /opt/python-server
# copy the correct python script to the current working directory
COPY ./python-server-${PYTHON_VERSION}.py app.py
# copy the index.html to the install folder
# the install folder is where the python application will be running
# all content from the install will be served because of this
# when a request is made to the server, the index.html will
# be served by default
COPY index.html .
# this executes a shell command to alter the contents of the index.html
# the webpage will have different content depending on the
# version of Python that is running
RUN sed -i "s/{{PYTHON_VERSION}}/${PYTHON_VERSION}/g" index.html
# the application runs on port 9000
# port 9000 on TCP has been exposed here for documentation purposes
EXPOSE 9000
# an entrypoint has been set here
# the Python binary is executed, with the correct script as an argument
ENTRYPOINT ["/usr/local/bin/python", "app.py"]
```

### Running the Application

Your exercise folder should now look something like below. Now try building the Docker Image and running it in a container. Remember that the application runs on Port 9000 and you will need to publish that port if you want to access the application via HTTP. Try changing the PYTHON_VERSION argument to 2.7.15 with the --build-arg option.

```
Dockerfile
index.html
python-server-2.7.15.py
python-server-3.6.8.py
```

To cleanup, stop and remove all containers and images.

### Build Your Own

For a project that is relevant for you, try building a Docker image of it. This could be a Java project using maven, Node, Python etc.
Upload your created Docker image to Docker hub and see if one of your peers can get it running.