# How to Create a Docker Image?

written by sysadmin | 5 November 2025

Previously, you often used a Docker image that you downloaded from Docker Hub. But now, you want to create a Docker image for your own application needs.

## Problem

How to create a Docker image?

## Solution

To create a Docker image, you must create a Dockerfile.

### A. Dockerfile

Dockerfile is a script that contains a set of instructions used to create a Docker image using the format:

```
#comment
INSTRUCTION arguments
```

You should know that Docker runs Dockerfile files sequentially from top to bottom, and this file does not have a file extension, so just write Dockerfile. To make things easier, it is best to place this Dockerfile in the same location as the files needed to create a Docker image, so that it is easier to create. That way, to create a Docker image, just run the command:

```
docker build -t image_name .
```

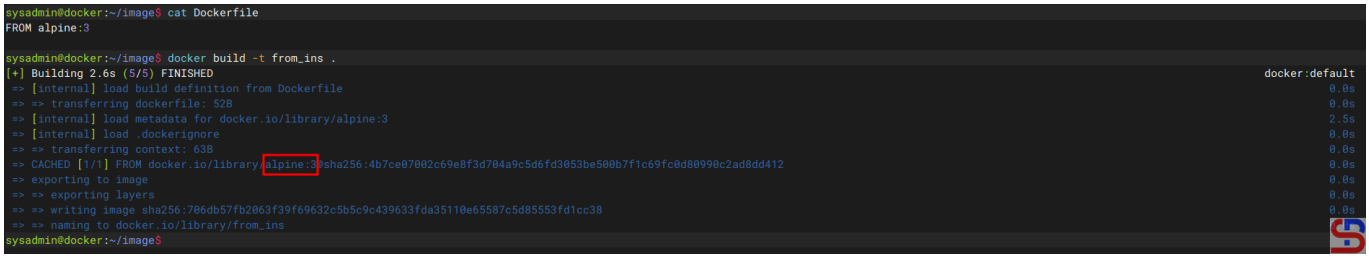
### B. Instructions

The following are the standard Dockerfile instructions:

**1. FROM instruction**

This instruction is the first command to perform a build stage in the Dockerfile with the example below:

```
FROM alpine:3
```

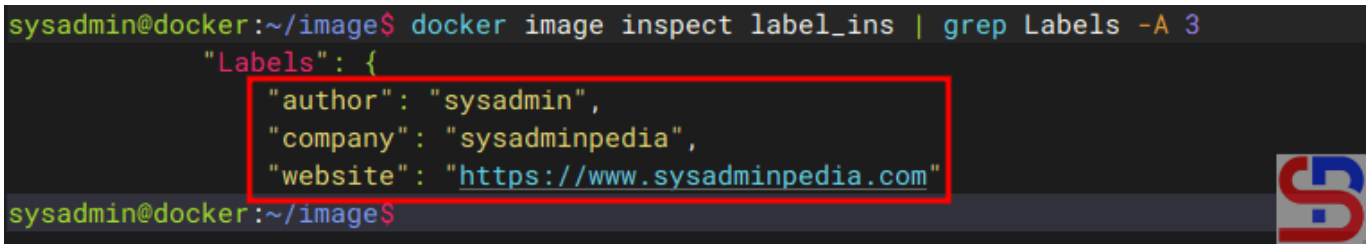



Using the FROM instruction

**2. LABEL instruction**

To add metadata to the Docker image you create, where the metadata is additional information, such as the name of the application, creator, website, and so on.

```
FROM alpine:3

LABEL author="sysadmin"
LABEL company="sysadminpedia" website="https://www.sysadminpedia.com"
```




Using the LABEL instruction

**3. WORKING DIRECTORY instruction**

This instruction specifies directories/folders to execute instructions in the container. By default, if there is no working directory, then the container will go to the / folder automatically. If the workdir does not exist, the directory will automatically be created, and then, after we determine the location of the workdir, the directory will be used as a place to execute the next instruction. If the workdir's location is a relative path, then it will

automatically enter the directory of the previous workdir. Workdir can also be used as a path for the first location when it enters the container.

```
FROM alpine:3

LABEL author="sysadmin"
LABEL company="sysadminpedia" website="https://www.sysadminpedia.com"

WORKDIR /app
```

```
sysadmin@docker:~/image$ docker run --rm -it workdir_ins
/app # pwd
/app
/app #
```

Using the WORKDIR instruction

**4. RUN instruction**

This instruction is a command in the image during the build stage, where the results of the RUN command will be committed to changes to the image, so that the RUN command will only be executed during the Docker build process, and this command will not be executed again and when you run the docker container from the image the RUN command will not be executed.

```
FROM alpine:3

LABEL author="sysadmin"
LABEL company="sysadminpedia" website="https://www.sysadminpedia.com"

WORKDIR /app

RUN mkdir hello
RUN echo "Hello World" > "hello/world.txt"
```

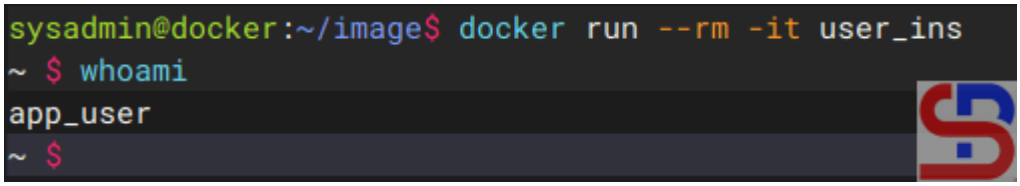
```
sysadmin@docker:~/image$ docker run --rm -it run_ins
/app # cat hello/world.txt
Hello World
/app #
```

Using the RUN instruction

**5. USER instruction**

To change the user or user group when Docker images are run, because by default, Docker will use the root user, and we can change it by using the user instruction, with the note that the user must be created first.

```
FROM alpine:3

LABEL author="sysadmin"
LABEL company="sysadminpedia" website="https://www.sysadminpedia.com"

WORKDIR /app

RUN mkdir hello
RUN echo "Hello World" > "hello/world.txt"

RUN addgroup -S app_group
RUN adduser -S -D -h /app app_user app_group
RUN chown -R app_user:app_group /app
USER app_user
```



Using the USER instruction

**6. ENTRYPOINT instruction**

An instruction in the Dockerfile specifies the main command that is executed when the container is started, and this is the main process of the container.

```
FROM alpine:3

LABEL author="sysadmin"
LABEL company="sysadminpedia" website="https://www.sysadminpedia.com"


WORKDIR /app

RUN mkdir hello
RUN echo "Hello World" > "hello/world.txt"

RUN addgroup -S app_group
RUN adduser -S -D -h /app app_user app_group
RUN chown -R app_user:app_group /app
```

```
USER app_user

ENTRYPOINT ["cat", "hello/world.txt"]
```

```
sysadmin@docker:~/image$ docker run --rm -it entrypoint_ins
Hello World
sysadmin@docker:~/image$
```

Using the ENTRYPOINT instruction

**7. COMMAND instruction**

An instruction that is used when the Docker container is running and will not be executed during the build image process. You cannot add more than one CMD instruction in an image, and if there is more than one instruction in an image, then the last CMD instruction will be executed. If there is an ENTRYPOINT instruction, then the CMD instruction becomes an argument from the ENTRYPOINT instruction. Examples of its use are as below:

```
FROM alpine:3

LABEL author="sysadmin"
LABEL company="sysadminpedia" website="https://www.sysadminpedia.com"

WORKDIR /app

RUN mkdir hello
RUN echo "Hello World" > hello/world.txt

RUN addgroup -S app_group
RUN adduser -S -D -h /app app_user app_group
RUN chown -R app_user:app_group /app
USER app_user

ENTRYPOINT ["cat"]

CMD ["hello/world.txt"]
```

The Dockerfile file above seems to be the command cat hello/world.txt, as in the picture below:

```
sysadmin@docker:~/image$ docker run --rm -it cmd_ins
Hello World
sysadmin@docker:~/image$
```

Using the CMD instruction

If there is no ENTRYPOINT, the CMD itself can be executed directly as a container command, as in the Dockerfile file below:

```
FROM alpine:3

LABEL author="sysadmin"
LABEL company="sysadminpedia" website="https://www.sysadminpedia.com"

WORKDIR /app

RUN mkdir hello
RUN echo "Hello World" > hello/world.txt

RUN addgroup -S app_group
RUN adduser -S -D -h /app app_user app_group
RUN chown -R app_user:app_group /app
USER app_user

CMD ["cat", "hello/world.txt"]
```

**8. COPY instruction**

To add files from the source to the destination folder in a Docker image folder. For example, if you want to copy all files with the extension .txt from a folder on the server, put them in the hello folder in the Docker image.

```
FROM alpine:3

LABEL author="sysadmin"
LABEL company="sysadminpedia" website="https://www.sysadminpedia.com"

WORKDIR /app

RUN mkdir hello
RUN echo "Hello World" > "hello/world.txt"

RUN addgroup -S app_group
RUN adduser -S -D -h /app app_user app_group
RUN chown -R app_user:app_group /app
USER app_user

COPY *.txt hello
```

```
CMD ["cat", "hello/world.txt"]
```

```
sysadmin@docker:~/image$ ls
Dockerfile  etc.tar.gz  hello.txt  nginx.conf  test.txt
sysadmin@docker:~/image$ docker run --rm -it copy_ins sh
~ $ ls hello/
hello.txt  test.txt  world.txt
~ $
```

Using the COPY instruction

You can see in the image that in the folder on the server, there is an nginx.conf file and a tar.gz file,  but the files that are copied are only files with the extension .txt, such as the COPY command in the Dockerfile file.

**9. ADD instruction**

To add files from the source to the destination folder in the Docker image, and this command can detect if a source file is a compressed file, such as gzip, and will automatically extract it in the destination folder, and can also support adding multiple files at once. The difference with COPY is that COPY can only copy files, while ADD, in addition to copying, can also download the source from the URL and automatically extract compressed files. The best way to practice is to use COPY as much as possible, but if you really need to extract compressed files, then use the ADD command.

```
FROM alpine:3

LABEL author="sysadmin"
LABEL company="sysadminpedia" website="https://www.sysadminpedia.com"

WORKDIR /app

RUN mkdir hello
RUN echo "Hello World" > "hello/world.txt"

RUN addgroup -S app_group
RUN adduser -S -D -h /app app_user app_group
RUN chown -R app_user:app_group /app
USER app_user

COPY *.txt hello
```

```
ADD *.gz hello

CMD ["cat", "hello/world.txt"]
```

```
sysadmin@docker:~/image$ ls
Dockerfile  etc.tar.gz  hello.txt  nginx.conf  test.txt
sysadmin@docker:~/image$ docker run --rm -it add_ins sh
~ $ ls hello/
crontab    hello.txt  hosts      test.txt   world.txt
~ $
```

Using the ADD instruction

The crontab and hosts files are extracted from the etc.tar.gz file, and this explains that when you use the COPY instruction to copy a compressed file, the file will be extracted automatically in the container.

**10. .dockerignore File**

To specify which file(s) or folder(s) we ignore when the process of copying or adding file(s) or folder(s) to the Docker image. Because when doing the copy or add process, Docker will read the file named .dockerignore first. Create a **.dockerignore** file and then fill it with example.txt and qwerty.txt. Then, create the files example.txt and qwerty.txt in that folder, and they should not be copied to the folder in the Docker image.

```
sysadmin@docker:~/image$ touch example.txt qwerty.txt
sysadmin@docker:~/image$ ls -a
.  ..  Dockerfile  .dockerignore  etc.tar.gz  example.txt  hello.txt  nginx.conf  qwerty.txt  test.txt
sysadmin@docker:~/image$ cat .dockerignore
example.txt
qwerty.txt
sysadmin@docker:~/image$ docker run --rm -it ignore_ins sh
~ $ ls hello/
crontab    hello.txt  hosts      test.txt   world.txt
~ $
```

Using the .dockerignore file

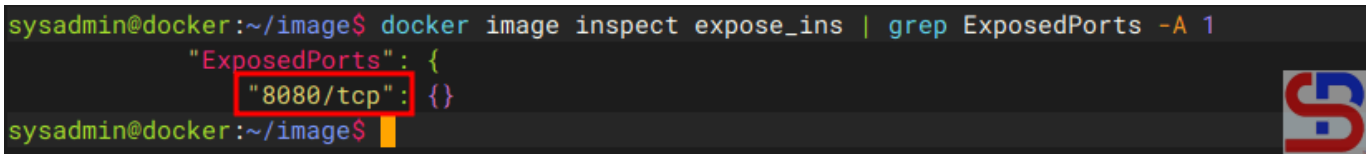
**11. EXPOSE instruction**

To tell the container which port to listen port on a specific number and protocol. Actually, EXPOSE will not publish any ports, but is only for documentation to inform the creator of the Docker container that this Docker image will use a specific port when run in a Docker container. For

example, as below:

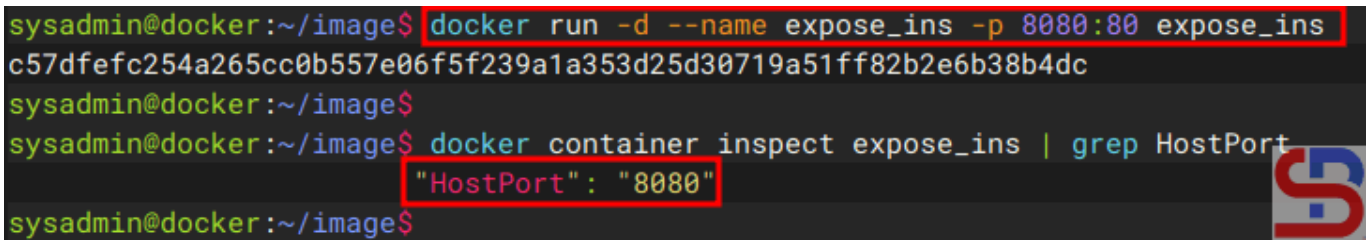
```
FROM alpine:3

LABEL author="sysadmin"
LABEL company="sysadminpedia" website="https://www.sysadminpedia.com"

WORKDIR /app

RUN mkdir hello
RUN echo "Hello World" > "hello/world.txt"

RUN addgroup -S app_group
RUN adduser -S -D -h /app app_user app_group
RUN chown -R app_user:app_group /app
USER app_user

ADD *.txt hello

COPY *.gz hello

EXPOSE 8080

CMD ["cat", "hello/world.txt"]
```

To see the open ports on this Docker image, use the command as shown in the image below:




Using the EXPOSE instruction

To create a container using this Docker image and see the open ports on this container, use the command as shown in the image below:




Display the EXPOSE instruction in the container

**12. ENVIRONMENT VARIABLE instruction**

To change the environment variables either during the build stage or when running in a Docker Container. The ENV defined in the Dockerfile can be reused using the ${ENV_NAME} syntax.

```
FROM alpine:3

LABEL author="sysadmin"
LABEL company="sysadminpedia" website="https://www.sysadminpedia.com"

WORKDIR /app

RUN mkdir hello
RUN echo "Hello World" > "hello/world.txt"

RUN addgroup -S app_group
RUN adduser -S -D -h /app app_user app_group
RUN chown -R app_user:app_group /app
USER app_user

COPY *.txt hello

ADD *.gz hello

EXPOSE 8080

ENV APP_PORT=8080
EXPOSE ${APP_PORT}

CMD ["cat", "hello/world.txt"]
```

The Environment Variable created using the ENV instruction is stored in the Docker image and can be viewed using the docker image inspect command.

```
sysadmin@docker:~/image$ docker image inspect env_ins | grep Env -A 3
            "Env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
                "APP_PORT=8080"
            ],
sysadmin@docker:~/image$
```

Using the ENV instruction

Environment variables can be changed in value when creating

a Docker container with the docker command using the **–env-key=value** option. So if you use the Dockerfile above, which uses APP_PORT=8080, but you want to use APP_PORT=9090, then you can use the command:

```
docker container create --name env_ins --env APP_PORT=9090 -p 9090:80 env_ins
```

```
sysadmin@docker:~/image$ docker container create --name env_ins --env APP_PORT=9090 -p 9090:80 env_ins
1d645f103fb2905cb31be5f772349e5d6ca86b7165396b1a115db759d555fecc
sysadmin@docker:~/image$
sysadmin@docker:~/image$ docker container inspect env_ins | grep APP_PORT
                "APP_PORT=9090",
sysadmin@docker:~/image$
```

Using the ENV instruction when creating a container

**13. VOLUME instruction**

To create the volume automatically when creating the container, and all the files contained in the volume are automatically copied to the Docker Volume, even though we didn’t create the Docker Volume when creating the Docker Container. It is suitable for cases when the application stores data in a file, so that the data can be automatically and safely stored in the Docker Volume.

```
FROM alpine:3

LABEL author="sysadmin"
LABEL company="sysadminpedia" website="https://www.sysadminpedia.com"

WORKDIR /app

RUN mkdir hello
RUN echo "Hello World" > "hello/world.txt"

RUN addgroup -S app_group
RUN adduser -S -D -h /app app_user app_group
RUN chown -R app_user:app_group /app
USER app_user

COPY *.txt hello

ADD *.gz hello

ENV APP_PORT=8080
EXPOSE ${APP_PORT}

ENV APP_DATA=/logs
```

```
VOLUME ${APP_DATA}

CMD ["cat", "hello/world.txt"]
```

```
sysadmin@docker:~/image$ docker image inspect vol_ins | grep Volume -A2
            "Volumes": {
                "/logs": {}
            },
sysadmin@docker:~/image$
```

Using the VOL instruction

Then, create a container from the Docker image and inspect it with the keyword Mounts. List the volume on the server, and it should be the same as in the image below:

```
sysadmin@docker:~/image$ docker container create --name vol_ins --env APP_PORT=9090 -p 9090:80 vol_ins
33cc9ecac5b30529feccf8f6ddce7bbd0c6edf97b70496976df40100a0de1f48
sysadmin@docker:~/image$
sysadmin@docker:~/image$ docker container inspect vol_ins | grep Mounts -A6
        "Mounts": [
            {
                "Type": "volume",
                "Name": "fef469d23766c072f239a44829516d0219ee1b98a2c71ef13a8516cb2f8763a2",
                "Source": "/var/lib/docker/volumes/fef469d23766c072f239a44829516d0219ee1b98a2c71ef13a8516cb2f8763a2/_data",
                "Destination": "/logs",
                "Driver": "local",
sysadmin@docker:~/image$
sysadmin@docker:~/image$ docker volume ls
DRIVER    VOLUME NAME
local     a685e9b17ea6c499d743dfd35d2fb799706b2f278d7d09bddfa5f0e377a86229
local     fef469d23766c072f239a44829516d0219ee1b98a2c71ef13a8516cb2f8763a2
sysadmin@docker:~/image$
```

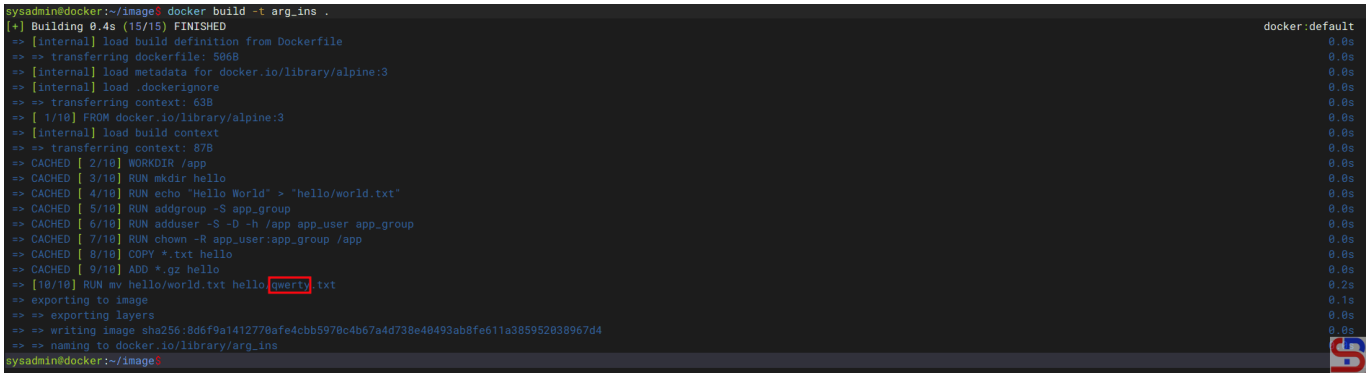
Display the volume

**14. ARGUMENT instruction**

To define variables that can be used by the user to send when performing a Docker build process, use the **–build-arg key=value** command. This instruction is only used during the build time process, which means that when running in a Docker container, this instruction will not be used any differently from the ENV used. Accessing variables from ARG is the same as accessing variables from ENV using ${variable_name}.

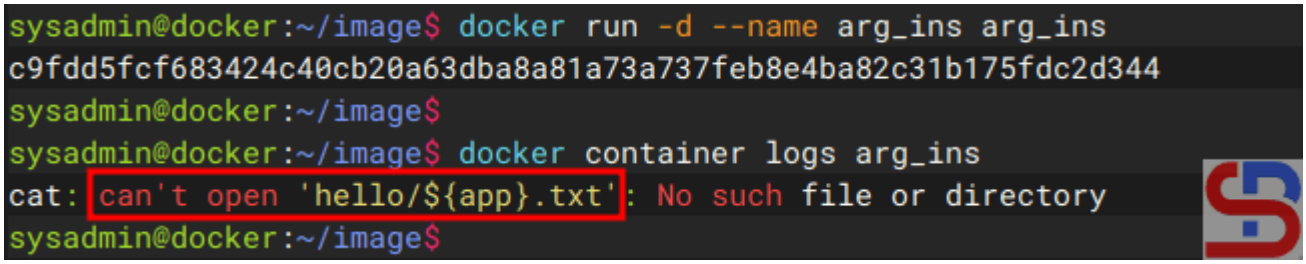
```
FROM alpine:3

LABEL author="sysadmin"
LABEL company="sysadminpedia" website="https://www.sysadminpedia.com"

WORKDIR /app
```

```
RUN mkdir hello
RUN echo "Hello World" > "hello/world.txt"

RUN addgroup -S app_group
RUN adduser -S -D -h /app app_user app_group
RUN chown -R app_user:app_group /app
USER app_user

COPY *.txt hello

ADD *.gz hello

ENV APP_PORT=8080
EXPOSE ${APP_PORT}

ARG app=qwerty
RUN mv hello/world.txt hello/${app}.txt

CMD ["cat", "hello/${app}.txt"]
```



Using the ARG instruction

As you can see in the image above, the file name changes to qwerty.txt, which corresponds to the arguments you wrote in Docker. If you want to change the argument when creating a Docker image, then you can change it by adding the **–build-arg app=pass** option so that the file becomes pass.txt, as shown in the image below:



Using the ARG when creating a Docker image

Based on the image above, you can see that the file name has changed to pass.txt. However, you have to know that when you run the container and run the log command, it will display as shown in the picture below:

```
sysadmin@docker:~/image$ docker run -d --name arg_ins arg_ins
c9fdd5fcf683424c40cb20a63dba8a81a73a737feb8e4ba82c31b175fdc2d344
sysadmin@docker:~/image$
sysadmin@docker:~/image$ docker container logs arg_ins
cat: can't open 'hello/${app}.txt': No such file or directory
sysadmin@docker:~/image$
```

Error when opening the log

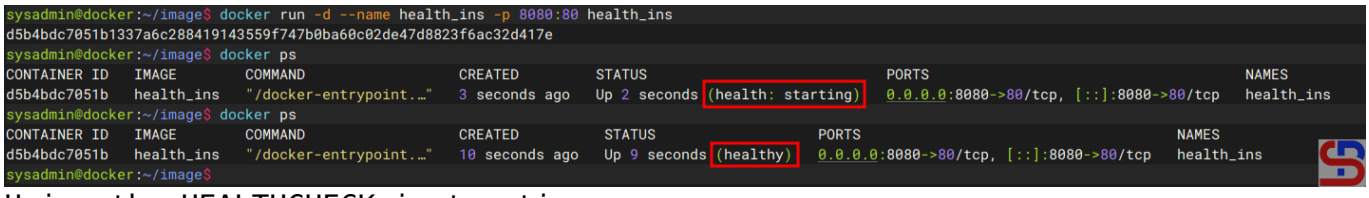
You can see in the image above that there is an error in the container log. This is because ARGs can only be accessed at build time, while CMDs are executed at runtime, so if you want to use ARG at runtime, then you need to insert the ARG into ENV, and your Dockerfile will be as below:

```
FROM alpine:3

LABEL author="sysadmin"
LABEL company="sysadminpedia" website="https://www.sysadminpedia.com"

WORKDIR /app

RUN mkdir hello
RUN echo "Hello World" > "hello/world.txt"

RUN addgroup -S app_group
RUN adduser -S -D -h /app app_user app_group
RUN chown -R app_user:app_group /app
USER app_user

COPY *.txt hello

ADD *.gz hello

ENV APP_PORT=8080
EXPOSE ${APP_PORT}

ARG app=qwerty
RUN mv hello/world.txt hello/${app}.txt

CMD ["cat", "hello/${app}.txt"]
```

**15. Health CHECK instruction**

To tell Docker if the container is still running properly or not. If there is a HEALTHCHECK, the container will automatically have a health status from the beginning with a starting value. If successful, it has a healthy value, and if it fails, it has an unhealthy value.

```
FROM nginx:alpine

RUN addgroup -S app_group \
    && adduser -S -G app_group -h /app app_user

RUN mkdir -p /app /var/cache/nginx /var/log/nginx /run \
    && chown -R app_user:app_group /app /var/cache/nginx /var/log/nginx /run

RUN echo "OK" > /app/healthz.html

RUN cat <<'EOF' > /etc/nginx/conf.d/default.conf
server {
    listen 80;

    location /healthz {
        root /app;
        try_files /healthz.html =404;
    }

    location / {
        return 200 "Hello from Nginx! Everything works fine.\n";
    }
}
EOF

USER app_user

# Healthcheck
HEALTHCHECK --interval=30s --timeout=5s --start-period=5s --retries=3 \
  CMD wget --no-verbose --tries=1 --spider http://127.0.0.1/healthz || exit 1

CMD ["nginx", "-g", "daemon off;"]
```



Using the HEALTHCHECK instruction

## Note

Actually, it was the developers who created the Dockerfile as an image for their applications to run on Docker. However, as a sysadmin, you should also understand the instructions in the Dockerfile so that you can help developers if there is an error in their Docker, or maybe also to create a Docker image for sysadmin purposes.

## References


youtube.dimas-maryanto.com
youtube.com
stackify.com
devopscube.com
geeksforgeeks.org