Docker Image

A Docker image is made up of a collection of files that bundle together such as installations, application code, and dependencies – required to configure a fully operational container environment.

You can create a Docker image by using one of two methods:

Interactive

Convert an existing container to docker image

- Run a container from existing image
- Make configurations of your application
- Convert the container to image
- Use this Image to create future containers

Dockerfile

- By constructing a plain-text file, known as a **Dockerfile**, which provides the specifications for creating a Docker image.

Dockerfile Directives

- A Docker image is created using instructions provided in a Dockerfile.
- These instructions are known as directives.
- Even though the directive is case-insensitive, it is a best practice to write all directives in uppercase to distinguish them from arguments.

Common Directives in Dockerfiles

The **FROM** directive
The **LABEL** directive
The **RUN** directive
The **CMD** directive
The **ENTRYPOINT** directive

FROM Directive

- A Dockerfile usually starts with the FROM directive.
- It specifies the base OS to be used.
- You can use an existing OS image either present in your local computer or pull from any Registry.

Parent Image

- The parent image can be an image from Docker Hub, such as Ubuntu, CentOS, Nginx, and MySQL.
- The FROM directive takes a valid image name and a tag as arguments.
- If the tag is not specified, the latest tag will be used.

```
FROM ubuntu

FROM ubuntu:latest
```

FROM ubuntu:16.04

FROM mysql (here you have an pre-built MySQL installed on any Linux OS)

Base Image

- A base image is what you create from scratch.

Lab: Building Ubuntu Base Image

Do this on any running Ubuntu 22.04 LTS VM or Laptop or Desktop or Server.

Create a directory for docker base image and change the current working directory after creating it.

```
mkdir -p /opt/docker_base_images
cd /opt/docker_base_images
```

Install debootstrap

```
apt install debootstrap
```

Run debootstrap

Here we are creating Ubuntu 22.04 LTS docker base image.

For this, we will use the Ubuntu release code name that is ‘Jammy’.

```
debootstrap jammy jammy > /dev/null
```

This command will take some time to finish.

After the command gets completed, you will see the directory relevant to the Ubuntu code name you used. In our case, a directory called ‘jammy’ is created.

You can see the release details using the command:

```
cat jammy/etc/lsb-release
```

Import the docker image in the local system

```
sudo tar -C jammy -c . | docker import - jammy
```

List docker images

You can find your imported docker image.

```
docker images
```

You can RUN a container based on this image:

```
docker run jammy cat /etc/lsb-release
```

LABEL Directive

- A LABEL is a **key-value pair** that can be used to add metadata to a Docker image.
- These labels can be used to organize the Docker images properly.

An example would be to add the name of the author of the Dockerfile or the version of the Dockerfile.

Dockerfile can have multiple labels, adhering to the preceding key-value format:

```
LABEL maintainer=raghavendra@cloudiq.in
LABEL version=1.0
LABEL environment=dev
```

Or these labels can be included on a single line separated by spaces:

```
LABEL maintainer=raghavendra@cloudiq.in version=1.0 environment=dev
```

Labels on an existing Docker image can be viewed with the **docker image inspect** command.

RUN Directive

- The RUN directive is used to execute commands during the image build time.
- The RUN directive can be used to install the required packages, update the packages, create users and groups, and so on.

In the following example, we are running two commands on top of the parent image. The **apt-get update** is used to update the package repositories, and **apt-get install nginx -y** is used to install the Nginx package:

```
FROM ubuntu
LABEL maintainer=raghavendra@cloudiq.in version=1.0 environment=dev
RUN apt-get update
RUN apt-get install nginx -y
```

Alternatively, you can add multiple shell commands to a single RUN directive by separating them with the && symbol.

```
FROM ubuntu
LABEL maintainer=raghavendra@cloudiq.in version=1.0 environment=dev
RUN apt-get update && apt-get install nginx -y
```

CMD Directive

- A Docker container is normally expected to run one process.
- A CMD directive is used to provide this default initialization command that will be executed when a container is created from the Docker image.
- A Dockerfile can execute only one CMD directive.
- If there is more than one CMD directive in the Dockerfile, Docker will execute only the last one.

```
FROM ubuntu
LABEL maintainer=raghavendra@cloudiq.in version=1.0 environment=dev
RUN apt-get update
RUN apt-get install apache2 -y
CMD ["apache2ctl", "-D", "FOREGROUND"]
```

Multiple CMDs

```
FROM ubuntu
CMD ["/bin/bash", "-c", "echo 1;echo 2;echo 3"]

#Build the image
docker build cmdtest .

#Run the container
docker run -it cmdtest
```

ENTRYPOINT directive

- Similar to the CMD directive, the ENTRYPOINT directive is also used to provide this default initialization command that will be executed when a container is created from the Docker image.
- The difference between the CMD directive and the ENTRYPOINT directive is that, unlike the CMD directive, we cannot override the ENTRYPOINT command using the command-line parameters sent with the docker container run command.

Example with CMD

Create a Dockerfile

```
FROM ubuntu
RUN apt-get update
ENTRYPOINT ["echo","Hello"]
CMD ["World"]
```

Build image

```
docker build -t test .
```

Run the container

```
docker run test
Hello World
```

You can see **Hello World** is printed executing both directives.

Now override CMD

```
docker run test india
Hello india
```

CMD is overwritten and instead india is print. ENTRYPOINT does not allow this override.

Other Directives

The **ENV** directive
The **ARG** directive
The **WORKDIR** directive
The **COPY** directive
The **ADD** directive
The **USER** directive
The **VOLUME** directive
The **EXPOSE** directive
The **HEALTHCHECK** directive
The **ONBUILD** directive

ENV Directive

- The ENV directive in Dockerfile is used to set environment variables.
- Environment variables are used by applications and processes to get information about the environment in which a process runs.
- One example would be the PATH environment variable, which lists the directories to search for executable files.

Dockerfile

```
FROM ubuntu:16.04
RUN apt-get update
RUN apt-get install -y apache2
```

```
RUN apt-get install -y openjdk-8-jre
RUN apt-get install -y ant
RUN apt-get clean
RUN apt-get update && \
    apt-get install ca-certificates-java && \
    apt-get clean && \
    update-ca-certificates -f
ENV JAVA_HOME /usr/lib/jvm/java-8-openjdk-amd64/
RUN export JAVA_HOME
EXPOSE 80
CMD ["apache2ctl", "-D", "FOREGROUND"]
```

Build image

```
docker build -t java-test .
```

Run the container

```
docker run -d java-test
```

```
root@ubuntu22:~/javatest# docker ps
CONTAINER ID   IMAGE       COMMAND                  CREATED         STATUS         PORTS     NAMES
2d2a96faa7e1   java-test   "apache2ctl -D FOREG…"   4 seconds ago   Up 2 seconds   80/tcp    lucid_hypatia
```

Connect to the Container Shell

```
docker exec -it 2d2a96faa7e1 /bin/bash
```

Find the Path Variables thar are set

```
root@2d2a96faa7e1:/# env
HOSTNAME=2d2a96faa7e1
TERM=xterm
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=3
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PWD=/
JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64/
SHLVL=1
HOME=/root
_=/usr/bin/env

```

As you can see JAVA_HOME is set as: `JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64/`

Validate that the ENV Directive has set the correct JAVA_PATH:

```
root@2d2a96faa7e1:/# echo $JAVA_HOME
/usr/lib/jvm/java-8-openjdk-amd64/
```

Another Example:

Setting Environment Variable.

```
FROM ubuntu
ENV TZ=Asia/Kolkata
RUN apt-get update
RUN apt-get install -y tzdata
RUN apt-get install -y apache2
RUN apt-get install -y apache2-utils
RUN apt-get install git -y
```

```
RUN apt update
RUN apt install -y libz-dev libssl-dev libcurl4-gnutls-dev libexpat1-dev gettext cmake gcc
RUN apt-get install -y nano && \
    apt-get install -y wget && \
    rm -fr /var/lib/apt/lists/*
RUN apt-get clean
EXPOSE 80
CMD ["apache2ctl", "-D", "FOREGROUND"]
```

Build Docker image

```
docker build -t timezone-test:1.0 .
```

Verify

```
docker images
```

Run it as a Container

```
docker run --name timezonecontainer -d -p 80:80 timezone-test:1.0
```

SSH in to the container

```
docker exec -it timezonecontainer /bin/bash
```

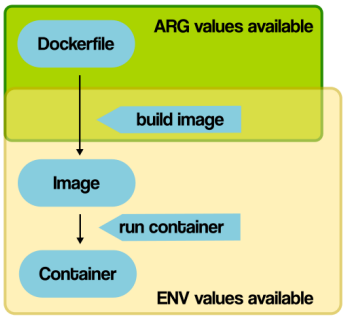
Finding the Time zone

```
date +%Z
```

```
# You see the output as UTC
IST
```

ARG directive

- ARG and env can be confusing at first
- Docker ARG vs ENV command instructions are used to set environment variables
- Both used for same purposes, but actually not same in functionality
- Both are Dockerfile instructions
- ENV is for future running containers and It works when the container is running
- ARG for building your Docker image
- ARG values are not available after the image is built
- A running container won't have access to an ARG variable value



Example

```
# ENV and ARG example
ARG TAG=latest
FROM ubuntu:$TAG
LABEL maintainer=raghavendra@cloudiq.in
ENV PUBLISHER=CloudIQ-Company
CMD ["env"]
```

Build Docker image

```
docker build -t argenv:1.0 --build-arg TAG=19.04 .
```

Here i am changing Ubuntu version to 19.04 overriding the Latest TAG (22.04)

Note the `env-arg --build-arg TAG=19.04` flag used to send the `TAG` argument to the build process.

Output

```
[+] Building 4.1s (5/5) FINISHED
 => [internal] load build definition from Dockerfile                                                0.0s
 => => transferring dockerfile: 176B                                                                0.0s
 => [internal] load .dockerignore                                                                   0.1s
 => => transferring context: 2B                                                                     0.0s
 => [internal] load metadata for docker.io/library/ubuntu:19.04                                     0.4s
 => [1/1] FROM docker.io/library/ubuntu:19.04@sha256:2adeae829bf27a3399a0e7db8ae38d5adb89bcaf1bbef378240  3.6s
 => => resolve docker.io/library/ubuntu:19.04@sha256:2adeae829bf27a3399a0e7db8ae38d5adb89bcaf1bbef378240  0.1s
 => => sha256:2adeae829bf27a3399a0e7db8ae38d5adb89bcaf1bbef378240bc0e6724e8344 1.42kB / 1.42kB      0.0s
 => => sha256:61844ceb1dd55aa110ca578bd4a042200bc64bb5d702c9a19b9fb90409565da0 1.15kB / 1.15kB      0.0s
 => => sha256:c88ac1f841b72add46f5a8b0e77c2ad6864d47e5603686ea64375acd55e27906 3.41kB / 3.41kB      0.0s
 => => sha256:4dc9c2fff01807ad6360d978aac7ce47455150e4725a1acbbbcda361ecf39e6b 27.62MB / 27.62MB    0.4s
 => => sha256:0a4ccbb242158237fe41d3dc405f13a94bf38ba3f2805ce0f7759565df405108 30.99kB / 30.99kB    0.1s
 => => sha256:c0f243bc6706a528213b7396fbd96640a848e0c65189362db1261a71c62ff3a0 861B / 861B         0.1s
 => => sha256:5ff1eaecba77a2d55818a0e8a80b324e6cf5ead6d0cbac915bc25b6d1c5d57b8 163B / 163B         0.1s
 => => extracting sha256:4dc9c2fff01807ad6360d978aac7ce47455150e4725a1acbbbcda361ecf39e6b           0.7s
 => => extracting sha256:0a4ccbb242158237fe41d3dc405f13a94bf38ba3f2805ce0f7759565df405108           0.0s
 => => extracting sha256:c0f243bc6706a528213b7396fbd96640a848e0c65189362db1261a71c62ff3a0           0.0s
 => => extracting sha256:5ff1eaecba77a2d55818a0e8a80b324e6cf5ead6d0cbac915bc25b6d1c5d57b8           0.0s
 => exporting to image                                                                              0.0s
 => => exporting layers                                                                             0.0s
 => => writing image sha256:d11e4dabbbd11fe343eef7595b54983e64ce1bc433f82f21ed5b4728a66e0c38        0.0s
 => => naming to docker.io/library/argenv:1.0
```

The Dockerfile ARG Latest is overwritten with 19.04

`load metadata for docker.io/library/ubuntu:19.04`

Note that the `19.04` tag of the ubuntu image was used as the parent image. This is because you sent the `--build-arg flag` with the value of `TAG=19.04` during the build process.

Verify

```
docker images
```

Run it as a Container

```
root@ubuntu22:~# docker container run -it argenv:1.0
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=1270479b4cbe
PUBLISHER=CloudIQ-Company
HOME=/root
```

ENV variable is as expected: `PUBLISHER=CloudIQ-Company`

Run a New Container again , but change the running ENV variable

```
root@ubuntu22:~# docker run --env PUBLISHER=ABC-corp -it argenv:1.0 env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=8cb1cfcac2d4
TERM=xterm
PUBLISHER=ABC-corp
HOME=/root
```

Container runtime changes are in effect and yo see a change in PUBLISHER:

`PUBLISHER=ABC-corp`

WORKDIR directive

- The `WORKDIR` directive is used to specify the current working directory of the Docker container.
- Any subsequent `ADD` , `CMD` , `COPY` , `ENTRYPOINT` , and `RUN` directives will be executed in this directory.

The `WORKDIR` directive has the following format:

```
WORKDIR /path/to/workdir
```

If the specified directory does not exist, Docker will create this directory and make it the current working directory, which means this directive executes both `mkdir` and `cd` commands implicitly.

There can be multiple `WORKDIR` directives in the `Dockerfile` .

If a relative path is provided in a subsequent `WORKDIR` directive, that will be relative to the working directory set by the previous `WORKDIR` directive:

```
WORKDIR /one
WORKDIR two
WORKDIR three
RUN pwd
```

In the preceding example, we are using the `pwd` command at the end of the `Dockerfile` to print the current working directory. The output of the `pwd` command will be `/one/two/three` .

Example: Build a Node.js Application with Docker

What is node.js?

- Node. js is an open-source, cross-platform JavaScript runtime environment and library for running web applications outside the client's browser.
- Developers use Node.js to build dynamic websites and web apps.
- Current Node.js examples of major websites thriving thanks to its platform include the likes of Netflix, PayPal, LinkedIn, and more.
- Azure Portal Dashboard is an example of node.js

How web applications run in node.js run?

- All files run in a Directory
- Files that require for the Web Application typically include: *.js, *.json files coded by developers. Web apps also use index.html kind of html pages.

The Project

Step 1 — Installing Your Application Dependencies

Create a directory for your project in your non-root user's home directory.

The directory in this example named `node_project`, but you should feel free to replace this with something else:

```
mkdir node_project
```

Navigate to this directory:

```
cd node_project
```

This will be the root directory of the project.

Next, create a `package.json` file with your project's dependencies and other identifying information.

Open the file with `nano` or your favorite editor:

```
nano package.json
```

```
{
  "name": "nodejs-image-demo",
  "version": "1.0.0",
  "description": "nodejs image demo",
  "author": "aghavendra <raghavendra@cloudiq.in>",
  "license": "MIT",
  "main": "app.js",
  "keywords": [
    "nodejs",
    "bootstrap",
    "express"
  ],
  "dependencies": {
    "express": "^4.16.4"
  }
}
```

Main Application = `app.js`

Free use License = `MIT`

Dependencies: Node application requires Express Web Server = `4.16.4`

First install node software

```
sudo apt install nodejs
sudo apt install npm
```

To install your project's dependencies, run the following command:

```
npm install
```

Step 2 — Creating the Application Files

Create `app.js` file with following content

```
nano app.js
```

```
const express = require('express');
const app = express();
const router = express.Router();

const path = __dirname + '/views/';
const port = 8080;

router.use(function (req,res,next) {
  console.log('/' + req.method);
  next();
});

router.get('/', function(req,res){
  res.sendFile(path + 'index.html');
});

router.get('/sharks', function(req,res){
  res.sendFile(path + 'sharks.html');
});

app.use(express.static(path));
app.use('/', router);

app.listen(port, function () {
  console.log('Example app listening on port 8080!')
})
```

Add Website index.html page in views subfolder

```
mkdir views
```

Create landing page file, `index.html`:

```
nano views/index.html
```

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>About Sharks</title>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css" integ
    <link href="css/styles.css" rel="stylesheet">
    <link href="https://fonts.googleapis.com/css?family=Merriweather:400,700" rel="stylesheet" type="text/css">
</head>

<body>
    <nav class="navbar navbar-dark bg-dark navbar-static-top navbar-expand-md">
        <div class="container">
            <button type="button" class="navbar-toggler collapsed" data-toggle="collapse" data-target="#bs-examp
            </button> <a class="navbar-brand" href="#">Everything Sharks</a>
            <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
                <ul class="nav navbar-nav mr-auto">
                    <li class="active nav-item"><a href="/" class="nav-link">Home</a>
                    </li>
                    <li class="nav-item"><a href="/sharks" class="nav-link">Sharks</a>
                    </li>
```

```
                </ul>
            </div>
        </div>
    </nav>
    <div class="jumbotron">
        <div class="container">
            <h1>Want to Learn About Sharks?</h1>
            <p>Are you ready to learn about sharks?</p>
            <br>
            <p><a class="btn btn-primary btn-lg" href="/sharks" role="button">Get Shark Info</a>
            </p>
        </div>
    </div>
    <div class="container">
        <div class="row">
            <div class="col-lg-6">
                <h3>Not all sharks are alike</h3>
                <p>Though some are dangerous, sharks generally do not attack humans. Out of the 500 species know
                </p>
            </div>
            <div class="col-lg-6">
                <h3>Sharks are ancient</h3>
                <p>There is evidence to suggest that sharks lived up to 400 million years ago.
                </p>
            </div>
        </div>
    </div>
</body>

</html>
```

Create sharks.html

```
nano views/sharks.html
```

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>About Sharks</title>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css" integ
    <link href="css/styles.css" rel="stylesheet">
    <link href="https://fonts.googleapis.com/css?family=Merriweather:400,700" rel="stylesheet" type="text/css">
</head>
<nav class="navbar navbar-dark bg-dark navbar-static-top navbar-expand-md">
    <div class="container">
        <button type="button" class="navbar-toggler collapsed" data-toggle="collapse" data-target="#bs-example-r
        </button> <a class="navbar-brand" href="/">Everything Sharks</a>
        <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
            <ul class="nav navbar-nav mr-auto">
                <li class="nav-item"><a href="/" class="nav-link">Home</a>
                </li>
                <li class="active nav-item"><a href="/sharks" class="nav-link">Sharks</a>
                </li>
            </ul>
        </div>
```

```
    </div>
</nav>
<div class="jumbotron text-center">
    <h1>Shark Info</h1>
</div>
<div class="container">
    <div class="row">
        <div class="col-lg-6">
            <p>
                <div class="caption">Some sharks are known to be dangerous to humans, though many more are not.
                </div>
                <img src="https://assets.digitalocean.com/articles/docker_node_image/sawshark.jpg" alt="Sawshark
            </p>
        </div>
        <div class="col-lg-6">
            <p>
                <div class="caption">Other sharks are known to be friendly and welcoming!</div>
                <img src="https://assets.digitalocean.com/articles/docker_node_image/sammy.png" alt="Sammy the S
            </p>
        </div>
    </div>
</div>

</html>
```

Create the CSS style sheet:

```
mkdir views/css
```

```
nano views/css/styles.css
```

```
.navbar {
    margin-bottom: 0;
}

body {
    background: #020A1B;
    color: #ffffff;
    font-family: 'Merriweather', sans-serif;
}

h1,
h2 {
    font-weight: bold;
}

p {
    font-size: 16px;
    color: #ffffff;
}

.jumbotron {
    background: #0048CD;
    color: white;
    text-align: center;
}

.jumbotron p {
```

```
    color: white;
    font-size: 26px;
}

.btn-primary {
    color: #fff;
    text-color: #000000;
    border-color: white;
    margin-bottom: 5px;
}

img,
video,
audio {
    margin-top: 20px;
    max-width: 80%;
}

div.caption: {
    float: left;
    clear: both;
}
```

To start the application, make sure that you are in your project's root directory:

```
cd ~/node_project
```

Start the application with `node app.js`:

```
node app.js
```

Navigate your browser to **http://your_server_ip:8080**

Dockerize the above Node application

Step 3 — Writing the Dockerfile

In your project's root directory, create the Dockerfile:

```
nano Dockerfile
```

```
FROM node:10-alpine

RUN mkdir -p /home/node/app/node_modules && chown -R node:node /home/node/app

WORKDIR /home/node/app

COPY package*.json ./

USER node

RUN npm install

COPY --chown=node:node . .

EXPOSE 8080
```

```
16
17 CMD [ "node", "app.js" ]
```

Save and close the file when you are finished editing.

dockerignore file

Before building the application image, add a `.dockerignore` file.

`.dockerignore` specifies which files and directories in your project directory should not be copied over to your container.

Open the `.dockerignore` file:

```
1 nano .dockerignore
```

```
1 node_modules
2 npm-debug.log
3 Dockerfile
4 .dockerignore
```

Docker Build

```
1 sudo docker build -t nodejs-image-demo .
```

Once it is complete, check your images:

```
1 sudo docker images
```

Run the following command to build the container:

```
1 sudo docker run --name nodejs-image-demo -p 8080:8080 -d nodejs-image-demo
```

COPY Directive

- During the Docker image build process, we may need to copy files from our local filesystem to the Docker image filesystem.
- These files can be source code files (for example, JavaScript files), configuration files (for example, properties files), or artifacts (for example, JAR files).
- The `COPY` directive can be used to copy files and folders from the local filesystem to the Docker image during the build process.

This directive takes two arguments. The first one is the source path from the local filesystem, and the second one is the destination path on the image filesystem:

```
1 COPY <source> <destination>
2
3 COPY package*.json ./
```

The ADD Directive

The `ADD` directive is also similar to the `COPY` directive, and has the following format:

```
1 ADD <source> <destination>
```

However, in addition to the functionality provided by the `COPY` directive, the `ADD` directive also allows us to use a URL as the `<source>` parameter:

```
ADD http://sample.com/test.txt /tmp/test.txt
```

Example

Using the WORKDIR, COPY, and ADD Directives in the Dockerfile

sudo su -

Create a new directory named `workdir-copy-add-exercise` using the `mkdir` command:

```
mkdir workdir-copy-add-exercise
```

Navigate to the newly created `workdir-copy-add-exercise` directory:

```
cd workdir-copy-add-exerciseCopy
```

Within the `workdir-copy-add-exercise` directory, create a file named `index.html`.

This file will be copied to the Docker image during build time:

```
touch index.html
```

Now, open `index.html` using nano editor:

```
nano index.html
```

```
<html>
  <body>
    <h1>Welcome to The Docker Workshop</h1>
    <img src="logo.png" height="431" width="601"/>
  </body>
</html>
```

Within the `workdir-copy-add-exercise` directory, create a file named `Dockerfile`:

```
touch DockerfileCopy
```

Now, open the `Dockerfile` using nano editor:

```
nano Dockerfile
```

Add the following content to the `Dockerfile`, save it, and exit from the `Dockerfile`:

```
# WORKDIR, COPY and ADD example
FROM ubuntu:latest
RUN apt-get update && apt-get install apache2 -y
WORKDIR /var/www/html/
COPY index.html .
ADD https://www.docker.com/wp-content/uploads/2022/03/Moby-logo.png ./logo.png
EXPOSE 80
CMD ["ls"]
```

Now, build the Docker image with the tag of `workdir-copy-add`:

```
docker image build -t workdir-copy-add .
```

Execute the `docker container run` command to start a new container from the Docker image that you built in the previous step:

```
docker container run workdir-copy-add
index.html
logo.png

```

As we can see from the output, both the `index.html` and `logo.png` files are available in the `/var/www/html/` directory.

Run this on Port 80 and feel the web page:

Delete the current container and rebuild with the following Dockerfile:

```
docker rm $(docker ps -aq)
docker rmi -f $(docker images)
```

```
# WORKDIR, COPY and ADD example
FROM ubuntu:latest
RUN apt-get update && apt-get install apache2 -y
WORKDIR /var/www/html/
COPY index.html .
ADD https://www.docker.com/wp-content/uploads/2022/03/Moby-logo.png ./logo.png
RUN chown -R root:root /var/www/html
RUN chmod -R 755 /var/www/html
CMD ["ls"]
CMD ["apache2ctl", "-D", "FOREGROUND"]
```

```
docker image build -t workdir-copy-add .
```

Run the container with Port 80 exposed

```
docker container run -d -p :80:80 workdir-copy-add
```

If you want to verify the contents of WORKINGDIR

```
docker exec -it dec1f83aba36 /bin/bash
root@dec1f83aba36:/var/www/html# ls
index.html  logo.png
```