

How to Build Docker Image : Comprehensive Beginners Guide

by [Aman Jaiswal](#) and [Bibin Wilson](#) · March 14, 2023



5
SHARES



In this article, you will learn to build Docker image from scratch, and deploy and run your application as a Docker container using `Dockerfile`

As you know Docker is a tool for packaging, deploying, and running applications in lightweight [containers](#). If you want to learn about the basics of Docker, refer to the [Docker explained](#) blog.

If you don't have Docker installation, check out the [docker installation guide](#).

Table of Contents [[show](#)]

Dockerfile Explained

The very basic building block of a Docker image is a `Dockerfile`

A `Dockerfile` is a simple text file with instructions and arguments. Docker can build images automatically by reading the instructions given in a `Dockerfile`.

In a Dockerfile Everything on left is **INSTRUCTION**, and on right is an **ARGUMENT** to those instructions. Remember that the file name is `"Dockerfile"` without any extension.



The following table contains the important **Dockerfile** instructions and their explanation.

Dockerfile Instruction	Explanation
FROM	To specify the base image which can be pulled from a container registry(Docker hub, GCR, Quay, ECR, etc)
RUN	Executes commands during the image build process.
ENV	Sets environment variables inside the image. It will be available during build time as well as in a running container. If you want to set only build-time variables, use ARG instruction.
COPY	Copies local files and directories to the image
EXPOSE	Specifies the port to be exposed for the Docker container.
ADD	It is a more feature-rich version of the COPY instruction. It also allows copying from the URL that is the source and tar file auto-extraction into the image. However, usage of COPY command is recommended over ADD. If you want to download remote files, use curl or get using RUN.
WORKDIR	Sets the current working directory. You can reuse this instruction in a Dockerfile to set a different working directory. If you set WORKDIR, instructions like RUN, CMD, ADD, COPY, or ENTRYPOINT gets executed in that directory.
VOLUME	It is used to create or mount the volume to the Docker container
USER	Sets the user name and UID when running the container. You can use this instruction to set a non-root user of the container.
LABEL	It is used to specify metadata information of Docker image
ARG	Is used to set build-time variables with key and value. the ARG variables will not be available when the container is running. If you want to persist a variable on a running container, use ENV.
CMD	It is used to execute a command in a running container. There can be only one CMD, if multiple CMD there then it only applies to the last one. It can be overridden from the Docker CLI.
ENTRYPOINT	Specifies the commands that will execute when the Docker container starts. If you don't specify any ENTRYPOINT, it defaults to <code>/bin/sh -c</code> . You can also override ENTRYPOINT using the <code>--entrypoint</code> flag using CLI. Please refer CMD vs ENTRYPOINT for more information.

Build Docker Image Using Dockerfile

In this section, you will learn to build a docker image using a real-world example. We will create an Nginx docker image from scratch with a custom index page.

Follow the steps given below to build a docker image.

Note: The Dockerfile and configs used for this article is hosted on a [Docker image examples Github repo](#). You can clone the repo for reference.

Step 1: Create the required Files and folders

Create a folder named `nginx-image` and create a folder named `files`

```
mkdir nginx-image && cd nginx-image  
mkdir files
```

Create a `.dockerignore` file

```
touch .dockerignore
```

Step 2: Create a sample HTML file & Config file

When you build docker image for real-time projects, it contains code or application config files.

For demo purposes, we will create a simple HTML file & config file as our app code and package it using Docker. This is a simple `index.html` file. You can create your own if you want.

cd in to the files folder

```
cd files
```

Create an `index.html` file

```
vi index.html
```

Copy the following contents to the `index.html` and save the file.

```
<html>
  <head>
    <title>Dockerfile</title>
  </head>
  <body>
    <div class="container">
      <h1>My App</h1>
      <h2>This is my first app</h2>
      <p>Hello everyone, This is running via Docker container</p>
    </div>
  </body>
</html>
```

create a file name default

```
vi default
```

Copy the following contents to the default file.

```
server {
    listen 80 default_server;
    listen [::]:80 default_server;

    root /usr/share/nginx/html;
    index index.html index.htm;

    server_name _;
    location / {
        try_files $uri $uri/ =404;
    }
}
```

Step 3: Choose a Base Image

We use `FROM` command in the `Dockerfile` which instructs Docker to create an image based on other images which are available on the Docker hub or any container registry configured with Docker. We call it a **base image**.

Choosing a base image depends on our application and os platform of choice. In our case, we will pick the `ubuntu:18.04` base image.

Note: Always use official base images for your applications to avoid potential vulnerabilities. Towards the end, we have added all the public registries that has verified container base images. Also, when it comes to **production use cases**, always use a [minimal base image like alpine](#) (just 5 Mib) or [distroless images](#). Distroless alpine is just 2 MiB

Step 3: Create the Dockerfile

Create a Dockerfile in the `nginx-image` folder.

```
vi Dockerfile
```

Here's the simple `Dockerfile` content for our use case. Add the content to the Dockerfile.

```
FROM ubuntu:18.04
LABEL maintainer="contact@devopscube.com"
RUN apt-get -y update && apt-get -y install nginx
COPY files/default /etc/nginx/sites-available/default
COPY files/index.html /usr/share/nginx/html/index.html
EXPOSE 80
CMD ["/usr/sbin/nginx", "-g", "daemon off;"]
```

Here's the explanation of each step:

- 1 With `LABEL` instruction, we are adding metadata about the maintainer. It is not a mandatory instruction.
- 2 `FROM` instruction will pull the Ubuntu 18.04 version Image from the Docker hub.
- 3 In the second line, we're installing Nginx.
- 4 Then we're copying the Nginx default config file from the local `files` directory to the target image directory.
- 5 Next, we're copying our `index.html` file from the local `files` directory into the target image directory. It will overwrite the default `index.html` file created during Nginx installation.
- 6 We're exposing port 80 as the Nginx service listens on port 80.
- 7 At last, we're running the Nginx server using `CMD` instruction when the Docker image launches.

For Docker containers, the `daemon off;` directive tells Nginx to stay in the foreground. It disables the self-daemonizing behavior of Nginx. the `-g` option specifies a directive to Nginx.

The reason we run the process in the foreground is to attach the console process to standard input, output, and error.

Step 4: Build your first Docker Image

The final folder & file structure would look like the following.

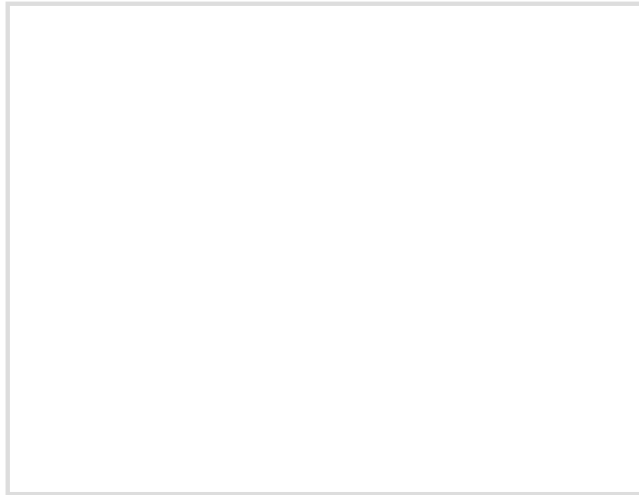
```
nginx-image
├── Dockerfile
├── files
│   ├── default
│   └── index.html
```

Now, we will build our image using the Docker command. The below command will build the image using `Dockerfile` from the same directory.

```
docker build -t nginx:1.0 .
```

- 1 `-t` is for tagging the image.
- 2 `nginx` is the name of the image.
- 3 `1.0` is the tag name. If you don't add any tag, it defaults to the tag named `latest`.

- 4 . means, we are referring to the Dockerfile location as the docker build context.

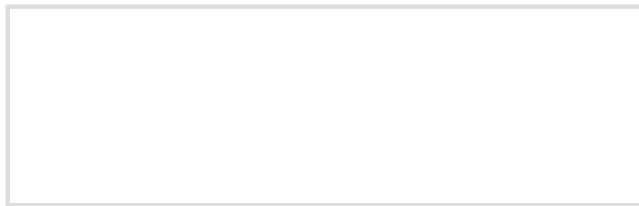


If the `Dockerfile` is in another folder then you need to specify it explicitly.

```
docker build -t nginx /path/to/folder
```

Now, we can list the images by using this command.

```
docker images
```



We can see the tag is `1.0` here. If we want to put a specific tag we can put it like this `image-name:<tag>` . If you don't specify any tag, it defaults to `latest` tag.

```
docker build -t nginx:2.0 .
```

A single image can have multiple tags. There are two approaches we generally follow to tag the image:

- 1 **Stable Tags** – We can continue to pull a specific tag, which continues to get updates. Our tags are always constant, but the image content is changed.
- 2 **Unique Tags** – We use a different and unique tag for each image. There are different ways to provide unique tags for example date-time stamp, build number, commit id, etc.

Note: When it comes to production, a recommended docker image tagging method is [semantic versioning \(Semver\)](#).

Docker caches the build steps. So if we build the image again the process will move a little faster. For example, it will not download the ubuntu 18.04 image again.

Using large images slows down the build and deployment time of containers. If you want to learn more about optimizing Docker images, check out [reduce docker image](#) guide.

Step 5: Test the Docker Image

Now after building the image we will run the Docker image. The command will be

```
docker run -d -p 9090:80 --name webserver nginx:1.0
```

Here,

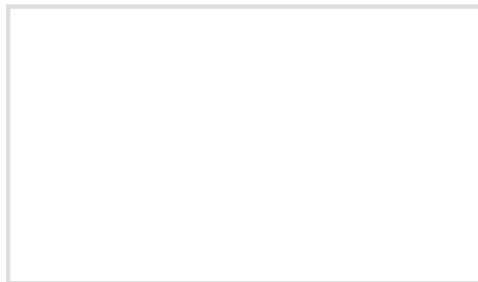
- 1 `-d` flag is for running the container in detached mode
- 2 `-p` flag for the port number, the format is local-port:container-port
- 3 `--name` for the container name, webserver in our case

We can check the container by using the below command

```
docker ps
```



Now in the browser, if you go to `http://<host-ip>:9090`, you can see the index page which displays the content in the custom HTML page we added to the docker image.

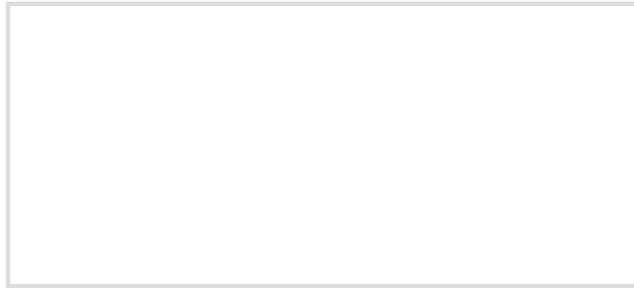


Push Docker Image To Docker Hub

To push our Docker image to the Docker hub, we need to create an account in the [Docker hub](#).

Post that, execute the below command to log in from the terminal. It will ask for a username and password. Provide the Docker hub credentials.

```
docker login
```



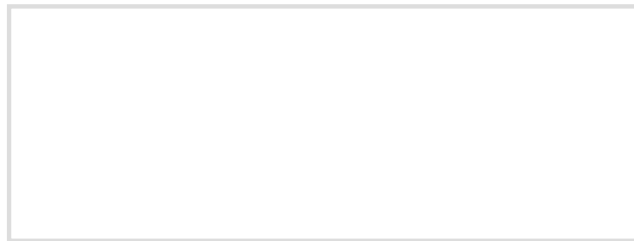
After login, we now need to **tag our image with the docker username** as shown below.

```
docker tag nginx:1.0 <username>/<image-name>:tag
```

For example, here `devopscube` is the dockerhub username.

```
docker tag nginx:1.0 devopscube/nginx:1.0
```

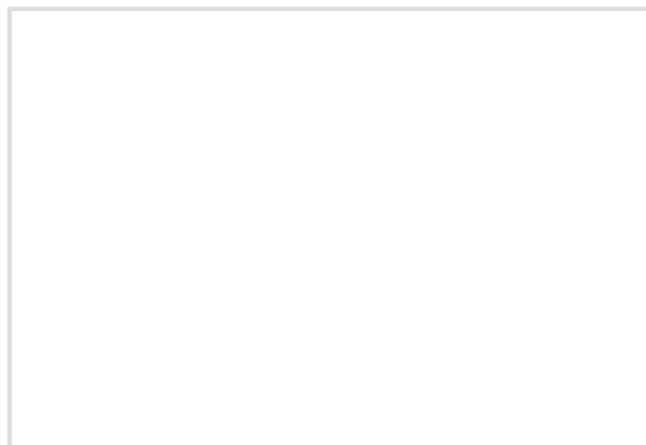
Run `docker images` command again and check the tagged image will be there.



Now we can push our images to the Docker hub using the below command.

```
docker push devopscube/nginx:1.0
```

Now you can check this image will be available in your Docker Hub account.



Using heredoc With Dockerfile

Dockerfile also supports [heredoc](#) syntax. If we have multiple `RUN` commands then we can use `heredoc` syntax as shown below.


```
RUN <<EOF
apt-get update
apt-get upgrade -y
apt-get install -y nginx
EOF
```

Also, let's say you want to execute a python script from a Dockerfile, you can use the following syntax.

```
RUN python3 <<EOF
with open("/hello", "w") as f:
    print("Hello", file=f)
    print("World", file=f)
EOF
```

You can also use the heredoc syntax to create a file. Here is an Nginx example.

```
FROM nginx

COPY <<EOF /usr/share/nginx/html/index.html
<html>
  <head>
    <title>Dockerfile</title>
  </head>
  <body>
    <div class="container">
      <h1>My App</h1>
      <h2>This is my first app</h2>
      <p>Hello everyone, This is running via Docker container</p>
    </div>
  </body>
</html>
EOF
```

Dockerfile Best Practices

Some of the `Dockerfile` practices which we should follow:

- 1 Use a `.dockerignore` file to exclude unnecessary files and directories to increase the build's performance.
- 2 Use **trusted base images** only and keep updating the images periodically.
- 3 Each instruction in the `Dockerfile` adds an extra layer to the Docker image. Minimize the number of layers by consolidating the instructions to increase the build's performance and time.
- 4 Run as a **Non-Root User** to avoid security breaches.
- 5 **Keep the image small:** Reduce the image size for faster deployment and avoid installing unnecessary tools in your image. Use minimal images to reduce the attack surface.
- 6 Use specific tags over the latest tag for the image to avoid breaking changes over time.
- 7 Avoid using multiple `RUN` commands as it creates multiple cacheable layers which will affect the efficiency of the build process.
- 8 Never share or copy the application credentials or any sensitive information in the `Dockerfile`. If you use it, add it to `.dockerignore`.
- 9 Use `EXPOSE` and `ENV` commands as late as possible in `Dockerfile`.
- 10 **Use a linter:** Use a linter like hadolint to check your Dockerfile for common issues and best practices.
- 11 **Use a single process per container:** Each container should run a single process. This makes it easier to manage and monitor containers, and helps to keep containers lightweight.

- 12 **Use multi-stage builds:** Use multi-stage builds to create smaller and more efficient images.

Possible Docker Build Issues

- 1 If there is a syntax error or an invalid argument in `Dockerfile`, `docker build` command will fail with an error message. Correct the syntax to resolve this.
- 2 Always try to give the container name using `docker run` command otherwise Docker automatically assigns a name to the container and it might lead to several problems.
- 3 Sometimes we get **Bind for 0.0.0.0:8080 failed: port is already allocated** error, this is because some other software/service is using these ports. We can check the listening ports using `netstat` or `ss` command. Use a different port to resolve this or stop that service.
- 4 Sometimes Docker failed to download the packages with this error **Failed to download package <package-name>**. This is because the container may not be able to access the internet or other dependencies issues.

Docker Image Registries

As mentioned in step 1, you should always choose verified official base images for your application.

The following table has the list of publicly available container registries where you can find officially verified base images and application images.

Registry	Base Images
Docker	Docker hub base images
Google Cloud	Distroless base images
AWS	ECR public registry
Redhat Quay	Quay Registry

Docker Image Vs Containers

A Docker image is a snapshot of the file system and application dependencies. It is an executable package of software that includes everything needed like application code, libraries, tools, dependencies, and other files to run an application. You can **compare it to a VM golden image**.

A Docker image is organized in read-only layers stacked on top of each other.

A Docker container is a running instance of a Docker image. We create VMs from VM images. Similarly way we create a container from a container image. When you create a container from a Docker image, you are creating a writable layer on top of the existing image layers.

The key difference between a Docker image and a container is the writable layer on top of the image. This means, if you have five containers running from an image, all the containers share the same read-only layers from the image and only the top writable layer is different for all five containers.

This means, when you delete the container, the writable layer gets deleted.

Images can exist without containers, whereas a container needs an image to run. We can create multiple containers from the same image, each with its own unique data and state.

Docker Image Build FAQs

How to use base images from container registries other than Docker hub?

By default docker, the docker engine is configured with the docker hub container registry. So if you mention the image name, it pulls the image from the docker hub. However, if you want to use an image from a different container registry, you need to provide the full URL of the image. For example, FROM `gcr.io/distroless/static-debian11`

What is Docker build context?

Docker build context is the docker host location where all the code, files, configs, and Dockerfile are present during the docker build process. You can specify the current build context using a dot `.` or the path of the folder. Also, you can have the Dockerfile in a different location than the build context. As a best practice, always have only the required files in the build context. Or else you will have unwanted files and a bloated Docker image.

How to build a Docker Image from a git repository?

You can use the docker build command with a git repository to build a docker image. The git repository should have the Dockerfile and required files for a successful image build.

Conclusion

In this article, we discussed how we can **build a Docker Image** and run our app as a Docker container using a `Dockerfile`.

We discussed `Dockerfile` in detail and went through some good practices to write it.

As a [DevOps engineer](#), it is important to have a solid understanding of Docker best practices before implementing them in a project. Additionally, to [learn Kubernetes](#),

you need to understand the workflow of building container images.

Podman is another container tool using which you can manage containers. To know more, check out the [podman tutorial](#).

TAGS:

Devops

Docker

Aman Jaiswal


Aman is a Software Engineer, working in the Fintech domain and has a good knowledge of DevSecOps. He is a certified Kubernetes Engineer. He is exploring and learning System Design and Cloud-Native tools. He loves to share his knowledge with the community.

[Twitter](#) [Instagram](#) [GitHub](#) [LinkedIn](#) [Medium](#)

Bibin Wilson

An author, blogger, and DevOps practitioner. In his spare time, he loves to try out the latest open source technologies. He works as an Associate Technical Architect. Also, the opinions expressed here are solely his own and do not express the views or opinions of his previous or current employer.

[GitHub](#) [Twitter](#) [Instagram](#) [YouTube](#) [GitHub](#) [LinkedIn](#) [RSS](#)

[VIEW COMMENTS \(3\)](#) 

YOU MAY ALSO LIKE



D — DEVOPS

Docker Networking Tutorial : Connect Containers Across Hosts Using Weave

by **devopscube** · June 16, 2015

Weave connects containers across hosts by creating a virtual network. It also has a DNS service which helps...



C — COMMON

Docker Machine Tutorial : Getting Started Guide

by **devopscube** · August 22, 2015

You all might have had that moment like "Ahh man! I have to execute all these commands..."



D — DOCKER

Cloud Based Docker Container Monitoring Using Datadog

by **devopscube** · August 26, 2016

2016 has been the year of Docker containers, as it has been the latest and the most advanced...

D - DOCKER

Docker ENTRYPOINT Vs CMD Explained With Examples

by **devopscube** · March 16, 2023

In this blog we will look at the key differences between Docker ENTRYPOINT vs CMD instruction using a...

C - COMMON

How to Setup and Configure Docker Swarm Cluster

by **devopscube** · September 10, 2017

In this tutorial, you will learn how to set up a four node docker swarm cluster. Prerequisites Minimum...

C - COMMON

Getting Started With Docker : Working With Containers

by **devopscube** · March 10, 2015

To understand how docker works, you can refer this article, which will give you an introduction about docker...

DevopsCube

©devopscube 2022. All rights reserved.

[Privacy Policy](#)

[About](#)

[Site Map](#)



[Disclaimer](#)

[Contribute](#)

[Advertise](#)

[Archives](#)