Docker Build: How to Build a Docker Image from Dockerfile?

by Goodness Chris-Ugari September 5th, 2023 DOCKER

Docker has reformed software development and deployment by providing a lightweight, portable, and efficient solution. The 'docker build' command is at the heart of Docker's image creation process. This article will explore the Docker build command, its syntax, usage, and the best practices to optimize your Docker builds.

Prerequisites

To follow along with the demonstration included in this guide, ensure you have:

- Docker or Docker Desktop installed

What does Docker build do?

So, what exactly is the Docker build command, and what is its role in the Docker ecosystem? Docker build is a command-line interface (CLI) command that allows you to build Docker images based on the instructions specified in a Dockerfile. It automates the image creation process, ensuring consistency and repeatability.

Syntax and usage of the Docker build command

To be able to use the Docker build command, you must first understand its syntax. Understanding the syntax and available options will help you customize the build process according to your requirements. The docker build command is run with the following syntax:

```
docker build [OPTIONS] PATH | URL
```

- `OPTIONS` : This refers to the flags and options that modify the build behavior.
- `PATH | URL` : This represents the build context or where you want to set up your docker container, which can be a local directory or a remote Git repository.

Ready to supercharge your Docker infrastructure? Scale effortlessly and enjoy flexible storage with Cherry Servers bare metal or virtual servers. Eliminate infrastructure headaches with free 24/7 technical support, pay-as-you-go pricing, and global availability.

Get Started Today

Docker build command: commonly used options and flags with examples

By understanding the syntax and available options of the Docker build command, you can tailor the build process to suit your specific needs, whether it's assigning tags, specifying alternate Dockerfile locations, passing build-time variables, or forcing a clean build.

When working with the Docker build command, there are several commonly used options and flags that provide flexibility and customization to the build process. Let's explore some of these options and flags with examples:

- `-t` or `--tag` : This option allows you to assign a tag to the built image for easy referencing and versioning. In this example, the built image will be tagged as "sampleapp" with the "latest" version.

Example usage: `docker build -t sampleapp:latest .`

- `-f` or `-file` : This option can be used to specify a different Dockerfile name or location if it is not named "Dockerfile." In this example, the image will be built using the Dockerfile named "ProductionDockerfile" in the current directory.

  Example usage: `docker build -f` `ProductionDockerfile` `.`

- `--build-arg` : The `--build-arg` option allows you to pass build-time variables to the Dockerfile, enabling dynamic customization during the build process. This command passes a build-time variable named "VERSION" with the value "10" to the Dockerfile.

  Example usage: `docker build --build-arg VERSION=10.`

- `--no-cache` : This option can be used to force Docker to ignore the cached layers and perform a fresh build. This command forces Docker to ignore the cache and perform a fresh build using all layers.

  Example usage: `docker build --no-cache .`

- `--target` : If your Dockerfile has multiple build stages defined using the `FROM` instruction, the `--target` option allows you to specify a specific target stage to build. In this example, only the stage defined as "mytarget" in the Dockerfile will be built.

  Example usage: `docker build --target mytarget .`

- `--quiet` or `-q` : The `--quiet` option suppresses the build output, displaying only the final image ID after the image has been built successfully. This command performs the build silently, showing only the resulting image ID.

  Example usage: `docker build --quiet .`

These are just a few commonly used options, and the Docker build provides more options and flags that you can explore in the official Docker documentation.

How does Docker build work?

The Docker build process creates a Docker image based on instructions in a Dockerfile. This file contains build instructions like copying files, installing dependencies, running commands, and more.

When you run `docker build`, the Docker daemon reads the Dockerfile and executes each instruction in order, generating a layered image. Each instruction creates a lightweight, read-only filesystem snapshot known as a layer. Docker caches unchanged layers to speed up subsequent builds. Layers that haven't changed since a previous build are cached and reused, avoiding the need to rebuild those layers and significantly improving build times.

Once all instructions are executed, Docker creates the final image containing the application and its dependencies, ready to be used to run containers.

How to build a Docker image from Dockerfile: Step-by-step

To build a Docker image from Dockerfile, let's see the `docker build` command in action. In the below steps, we will build a simple Docker image, a web server to serve out a web page through installation. You're going to be building, running, and testing it on your local computer.

Step 1 - Create a working directory

Create a directory or folder to use for this demonstration ("docker-demo" is used here) and navigate to that directory by running the following commands in your terminal:

```
mkdir docker-demo
cd docker-demo
```

Create a file called "index.html" in the directory and add the following content to it:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Simple App</title>
  </head>
  <body>
    <h1>Hello World</h1>
    <p>This is running in a docker container</p>
  </body>
</html>
```

This will serve as the web page to be served up by the server.

Step 2 - Select a base image

Next, select a suitable base image from Docker Hub or a local repository. The base image forms the foundation of your custom image and contains the operating system and essential dependencies. Almost every single image for Docker is based on another image. For this demonstration, you'd be using nginx:stable-alpine3.17-slim as the base image.

Step 3 - Create a Dockerfile

Now create a file named "Dockerfile". This file will define the build instructions for your image. By default, when you run the `docker build` command, docker searches for the Dockerfile.

Step 4 - Add build instructions to the Dockerfile

Open the Dockerfile in a text editor and add the following lines:

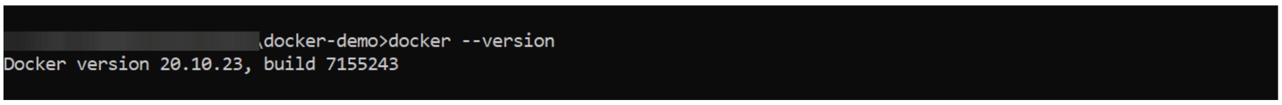
```
FROM nginx:stable-alpine3.17-slim
COPY index.html /usr/share/nginx/html

EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

The instruction above will create a Docker image that serves the provided "index.html" file through the Nginx web server when a container is launched based on that image.

The **FROM** instruction initializes a new build stage and sets the base image for subsequent instructions. The **COPY** instruction copies files or directories (usually the source code) from the source to the specified location inside the image. It copies the file to the "/usr/share/nginx/html" directory, which is the default location for serving static files in the Nginx web server. The main purpose of the **CMD** instruction is to provide defaults for executing containers. The instructions defined in Dockerfiles differ based on the kind of image you're trying to build.

Step 5 - Build the image using the Docker build command

Before building the Docker image, confirm that you have Docker installed by running the `docker --version` command.



```
[redacted]\docker-demo>docker --version
Docker version 20.10.23, build 7155243
```

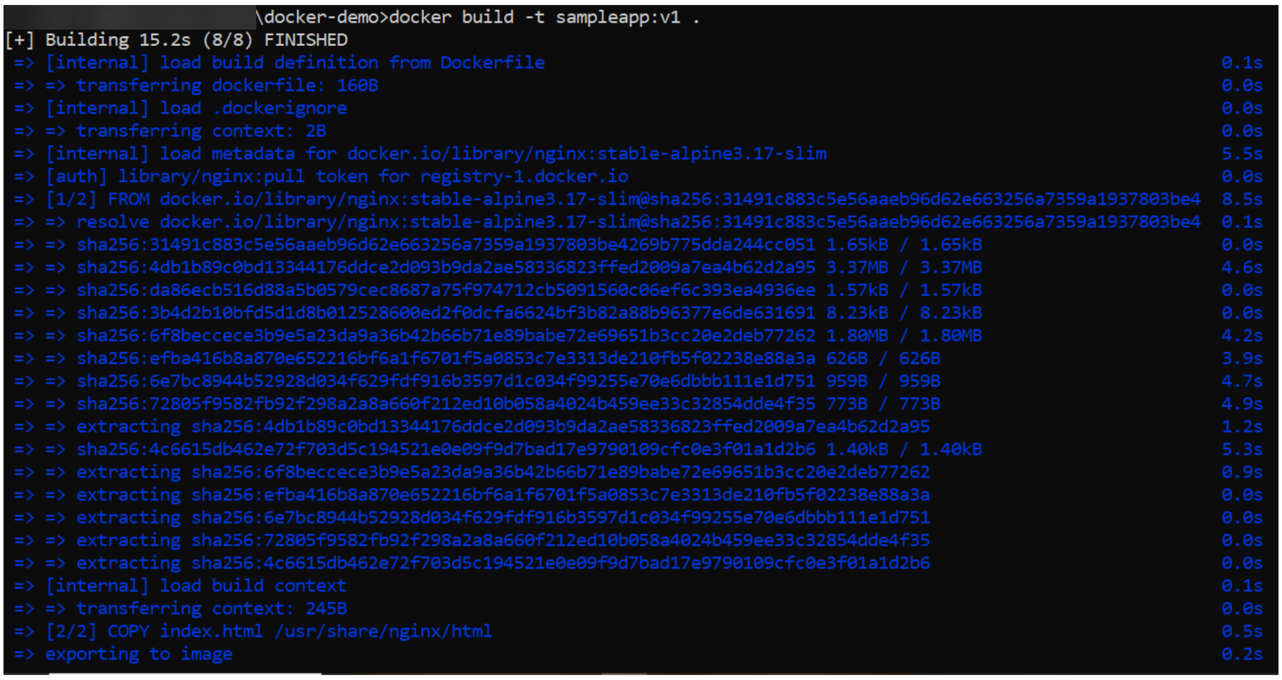
To build your container, make sure you're in the folder where you have your Dockerfile and run the following command in the terminal:

```
docker build -t sampleapp:v1 .
```

This command initiates the Docker build process to create a Docker image based on the instructions specified in the Dockerfile located in the current directory (.)

The `-t` flag specifies a tag for the image, allowing you to assign a name and version to it. In this case, the image will be tagged as "sampleapp" with the version "v1" providing a descriptive identifier for the image, making it easier to reference and manage.

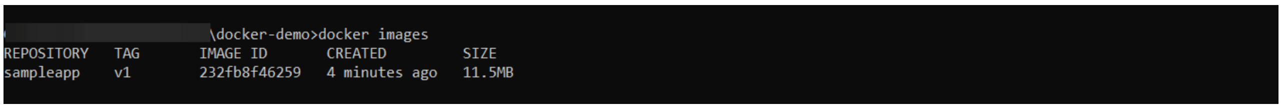
You should see the build process start and an output indicating that it has finished when it is done.



```
[redacted]\docker-demo>docker build -t sampleapp:v1 .
[+] Building 15.2s (8/8) FINISHED
 => [internal] load build definition from Dockerfile                                                       0.1s
 => => transferring dockerfile: 160B                                                                       0.0s
 => [internal] load .dockerignore                                                                          0.0s
 => => transferring context: 2B                                                                            0.0s
 => [internal] load metadata for docker.io/library/nginx:stable-alpine3.17-slim                            5.5s
 => [auth] library/nginx:pull token for registry-1.docker.io                                               0.0s
 => [1/2] FROM docker.io/library/nginx:stable-alpine3.17-slim@sha256:31491c883c5e56aaeb96d62e663256a7359a1937803be4  8.5s
 => => resolve docker.io/library/nginx:stable-alpine3.17-slim@sha256:31491c883c5e56aaeb96d62e663256a7359a1937803be4  0.1s
 => => sha256:31491c883c5e56aaeb96d62e663256a7359a1937803be4269b775dda244cc051 1.65kB / 1.65kB             0.0s
 => => sha256:4db1b89c0bd13344176ddce2d093b9da2ae58336823ffed2009a7ea4b62d2a95 3.37MB / 3.37MB             4.6s
 => => sha256:da86ecb516d88a5b0579cec8687a75f974712cb5091560c06ef6c393ea4936ee 1.57kB / 1.57kB             0.0s
 => => sha256:3b4d2b10bfd5d1d8b012528600ed2f0dcfa6624bf3b82a88b96377e6de631691 8.23kB / 8.23kB             0.0s
 => => sha256:6f8beccece3b9e5a23da9a36b42b66b71e89babe72e69651b3cc20e2deb77262 1.80MB / 1.80MB             4.2s
 => => sha256:efba416b8a870e652216bf6a1f6701f5a0853c7e3313de210fb5f02238e88a3a 626B / 626B                 3.9s
 => => sha256:6e7bc8944b52928d034f629fdf916b3597d1c034f99255e70e6dbbb111e1d751 959B / 959B                 4.7s
 => => sha256:72805f9582fb92f298a2a8a660f212ed10b058a4024b459ee33c32854dde4f35 773B / 773B                 4.9s
 => => extracting sha256:4db1b89c0bd13344176ddce2d093b9da2ae58336823ffed2009a7ea4b62d2a95                  1.2s
 => => sha256:4c6615db462e72f703d5c194521e0e09f9d7bad17e9790109cfc0e3f01a1d2b6 1.40kB / 1.40kB             5.3s
 => => extracting sha256:6f8beccece3b9e5a23da9a36b42b66b71e89babe72e69651b3cc20e2deb77262                  0.9s
 => => extracting sha256:efba416b8a870e652216bf6a1f6701f5a0853c7e3313de210fb5f02238e88a3a                  0.0s
 => => extracting sha256:6e7bc8944b52928d034f629fdf916b3597d1c034f99255e70e6dbbb111e1d751                  0.0s
 => => extracting sha256:72805f9582fb92f298a2a8a660f212ed10b058a4024b459ee33c32854dde4f35                  0.0s
 => => extracting sha256:4c6615db462e72f703d5c194521e0e09f9d7bad17e9790109cfc0e3f01a1d2b6                  0.0s
 => [internal] load build context                                                                          0.1s
 => => transferring context: 245B                                                                          0.0s
 => [2/2] COPY index.html /usr/share/nginx/html                                                            0.5s
 => exporting to image                                                                                     0.2s
```

Step 6 - Verify the built Docker image

After a successful build, verify the image by running the `docker images` command to list all the available images on your Docker host. You should see your newly created image listed with its assigned tag and other relevant details, ready to be used for running containers or pushing to a container registry for distribution.



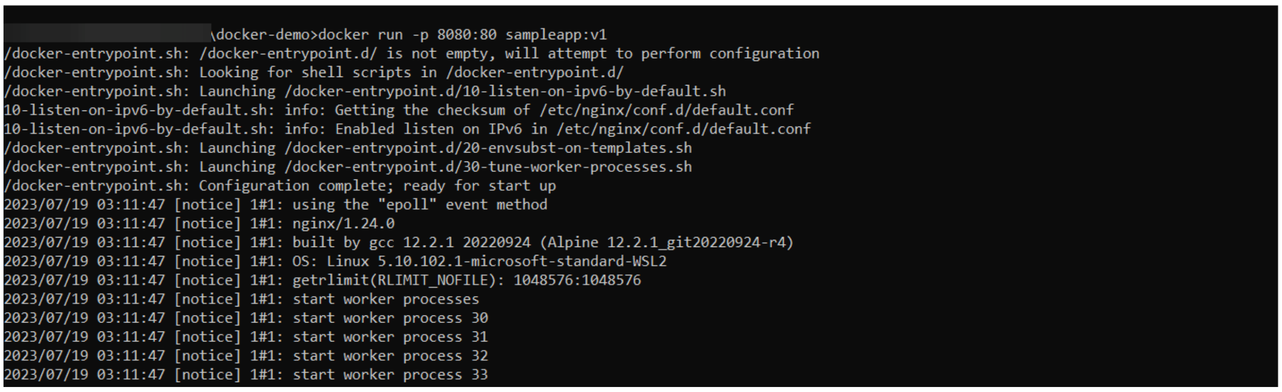
```
[redacted]\docker-demo>docker images
REPOSITORY   TAG       IMAGE ID       CREATED         SIZE
sampleapp    v1        232fb8f46259   4 minutes ago   11.5MB
```

Step 7 - Run the Docker image

Next, run the Docker image as a container using:

```
docker run -p 8080:80 sampleapp:v1
```

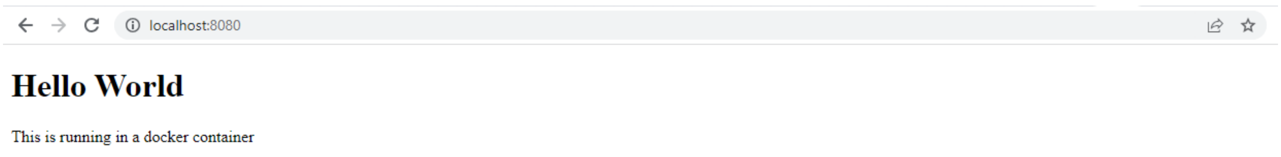
This command tells Docker to run the `sampleapp` container. The `-p` flag specifies the port mapping, which maps a port from the host machine to a port inside the container. Here, you are mapping port 8080 of the host machine to port 80 of the container. You can modify the host port as per your preference. Ensure you specify the image name and version you used when building the image.



```
[redacted]\docker-demo>docker run -p 8080:80 sampleapp:v1
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2023/07/19 03:11:47 [notice] 1#1: using the "epoll" event method
2023/07/19 03:11:47 [notice] 1#1: nginx/1.24.0
2023/07/19 03:11:47 [notice] 1#1: built by gcc 12.2.1 20220924 (Alpine 12.2.1_git20220924-r4)
2023/07/19 03:11:47 [notice] 1#1: OS: Linux 5.10.102.1-microsoft-standard-WSL2
2023/07/19 03:11:47 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576
2023/07/19 03:11:47 [notice] 1#1: start worker processes
2023/07/19 03:11:47 [notice] 1#1: start worker process 30
2023/07/19 03:11:47 [notice] 1#1: start worker process 31
2023/07/19 03:11:47 [notice] 1#1: start worker process 32
2023/07/19 03:11:47 [notice] 1#1: start worker process 33
```

Step 8 - Access the application

With the container running, you can go ahead to access the application. Open a web browser and navigate to `localhost:8080` and you should see the sample web page displayed on your web browser.



localhost:8080

Hello World

This is running in a docker container

Explore how web hosting service provider Debesis improved its service quality, performance, and reliability by migrating to Cherry Servers' bare-metal servers.

"Cherry Servers engineers always help when we need them, while their customer service quality is a blast!"

Read Their Success Story Here

Best practices for optimizing Docker builds

Here are some best practices you can follow to optimize your Docker build and ensure efficiency:

- Ensure you keep your Dockerfiles concise, organized, and easy to maintain. Break down complex tasks into smaller, reusable instructions, thereby improving readability, and simplifying troubleshooting and updates.
- Use lightweight images as base images in your Dockerfile and ensure you only install the essential things in the image. Smaller images not only improve performance but also reduce resource consumption during deployment.
- Use .dockerignore to ensure you're only adding the files you need.
- Take advantage of Docker's caching mechanism to speed up subsequent builds. Utilize layers that don't change frequently or use the `--no-cache=true` flag when necessary. This can be especially useful for large projects.
- Consider using multi-stage builds when applicable. They allow you to separate build dependencies from the final runtime image, resulting in smaller image sizes.

These practices will enhance the efficiency of your Docker builds, resulting in faster build times, smaller image sizes, improved performance, and reliable image creation. You can read more on docker build best practices from the documentation.

Troubleshooting Docker build command issues

Even with the best practices, you might encounter occasional hiccups during the Docker build process. Here are some common issues and their potential solutions:

- **Error: "Unable to find the Dockerfile":** Ensure that you are running the docker build command from the directory containing the Dockerfile.
- **Build failures or unexpected results:** Check the Dockerfile syntax and ensure all dependencies and commands are correctly specified. Review the build logs and error messages and consider using the --no-cache option to perform a clean build.

Also read: How Docker stores images

Conclusion

Docker build command empowers developers to turn your applications into portable containers that can be run anywhere. In this guide, we looked at how the Docker build process works, the

Docker build syntax and best practices, and how to build a Docker image from Dockerfile step-by-step. You should better understand the Docker build command and how to use it and utilize this knowledge to build more custom Docker images to make your applications portable.

Thanks for learning with Cherry Servers! Our flexible cloud infrastructure gives open source developers full control, stable workloads, and free technical support 24/7. We offer dedicated servers, virtual servers, anonymous hosting, GPU servers, customized services, and more.

Goodness Chris-Ugari

Cloud DevOps Engineer

Goodness is an expert in technical support, JavaScript programming, and cloud/DevOps engineering. She acquired her skills from studies in computer science and hands-on working experience. Over the years, Goodness has also honed the skills of creating, updating, and improving software documentation, writing instruction guides/manuals and technical articles for the knowledge base, and developing website content. Goodness is an expert in technical writing, DevOps engineering, Linux, Docker, containers, open-source, frontend development, and JavaScript. She also contributes to the documentation of open-source projects like Ansible and ODK-X. Goodness received her B.Sc. in Computer Science from the University of Port Harcourt and resides in Port Harcourt, Nigeria.

Related Articles

How to Install Nextcloud AIO on Docker | Step-by-Step

Apr 10, 2024 • by Goodness Chris-Ugari

This tutorial will show you how to install Nextcloud using Nextcloud AIO Docker image to help significantly reduce configuration steps.

Portainer Tutorial: How to Update [and Restart] Portainer

May 15, 2024 • by Winnie Ondara

This tutorial demonstrates how to manage Portainer: how to update Portainer, how to restart Portainer, and update existing containers.

DOCKER

Docker Swarm vs. Kubernetes – What are the Subtle Differences?

Apr 14, 2020 • by Marius Rimkus

Kubernetes and Docker Swarms are essential tools used to deploy containers inside a cluster. Learn how they differ and when to use them