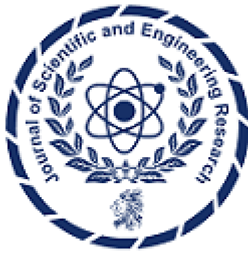
Available online www.jsaer.com

Journal of Scientific and Engineering Research, 2020, 7(1):284-292



Research Article

ISSN: 2394-2630
CODEN(USA): JSERBR

Improving the application's performance by writing effective Docker files

Pallavi Priya Patharlagadda

Engineering United States of America

Abstract Containerization is a software development approach aimed at packaging an application together with all its dependencies and execution environment in a lightweight, self-contained unit. Docker is a tool that is used to automate the deployment of applications in containers so that applications can work efficiently in different environments in isolation.

Docker images are an essential component for building Docker containers. Docker images are used to build and ship Docker containers. Docker images are created by writing Docker files. A Docker file is a text document that contains a sequence of commands a user could call on the command line to assemble an image. Docker build executes the list of instructions in the Docker file for creating a specific Docker image. So, it is important to write an effective Docker file to improve performance and reduce latency and storage. In this paper, we shed light on the best practices to be followed while writing the Docker file.

Keywords Application's Performance, Docker Files, Software Development, Docker

1. Introduction

The concept of containers started with Linux containers. Linux Containers is an operating-system-level virtualization method for running multiple isolated Linux systems on a control host using a single Linux kernel.

Docker is a tool that is used to automate the deployment of applications in containers so that applications can work efficiently in different environments in isolation. Docker images are an essential component for building Docker containers. Docker images are created by writing Docker files.

A Docker file is a text document that contains a sequence of commands a user could call on the command line to assemble an image. Each command in the Docker file makes a layer. Docker images are built in layers. Docker files enable faster and more efficient deployment of applications. Once a Docker image has been built, it can be easily deployed in any environment that supports Docker.

Docker files can be used in automation testing to build and run test environments for different applications and services. Using a Docker file, you can create an image of a specific test environment that can be easily and consistently recreated, without needing to manually set up and configure the environment on each test run.

You can easily integrate your Docker file with a continuous integration and continuous deployment (CI/CD) pipeline, which enables you to automatically build, test, and deploy your application with every code change. This helps to reduce the risk of errors and ensures that your application is always up to date. So, It is important to write effective Docker files, thereby reducing storage and improving the performance of the application.

2. Problem Statement

Even though the Docker build process is easy, many organizations make the mistake of building bloated Docker images without optimizing the container images.

In typical software development, each service will have multiple versions/releases, and each version requires more dependencies, commands, and configurations. This requires more time & resources to be used for building the Docker image before it can be shipped as a container.
The average size of our Docker images was ~300MB to ~600MB. But in some cases, the initial application image starts at 300 MB, and over time, it grows beyond 1.5 GB. This increases the storage requirement and latency and reduces performance.
Bigger Docker files take longer time to deploy as the time to build, download, transfer, and load the image into the container runtime increases. More network bandwidth is used.
Also, installing unwanted libraries can increase the chance of a potential security risk.
Therefore, it is necessary to optimize the Docker images to ensure that they are not bloated after application builds or future releases.

3. Docker configuration for performance evaluation

In this paper, I am comparing the build time and image size before and after applying the best practices. The Docker daemon has an in-built capability to display the total execution time that a Docker file is taking.

To enable this feature, follow below steps

Create a daemon.json file with the following contents: /etc/docker/

```
{
 "experimental": true
}
```

Execute the following command to enable the feature:

```
export DOCKER_BUILDKIT=1
```

After the above configuration, prepend the Docker build command with "time," and this should display the execution times at each layer on the terminal, and the total build time is displayed at the end.

```
time docker build -t <tag-name> --no-cache -f Dockerfile.
```

4. Docker file Instructions List

The table below describes different instructions that can be used in Docker file.

Instruction	Description
ADD	Add local or remote files and directories.
ARG	Use build-time variables.
CMD	Let's define the default program that is run once you start the container based on this image. Each Docker file only has one CMD, and only the last CMD instance is respected when multiple exist.
COPY	COPY <src> <dest>. Copies new files or directories from <src> and adds them to the filesystem of the container at the path <dest>.
ENTRYPOINT	Specify default executable.
ENV	Set environment variables.
EXPOSE	Describe which ports your application is listening on.
FROM	Create a new build stage from a base image.
HEALTHCHECK	Check a container's health on startup.
LABEL	Add metadata to an image.
MAINTAINER	Specify the author of an image.
ONBUILD	Specify instructions for when the image is used in a build.
RUN	Executes any commands in a new layer on top of the current image and

	commits the result. RUN also has a shell form for running commands.
SHELL	Set the default shell of an image.
STOPSIGNAL	Specify the system call signal for exiting a container.
USER	Set user and group ID.
VOLUME	Create volume mounts. Volumes are the preferred mechanism for persisting data generated by and used by Docker containers.
WORKDIR	Sets the working directory for any RUN, CMD, ENTRYPOINT, COPY, and ADD instructions that follow it in the Docker file.

5. Recommended Best Practices

Use a Smaller base Image

A Docker base image creates the foundation for structuring your Docker images. Docker provides pre-built images with the tools and libraries required to run your applications in containers. Docker provides different variants of such base images optimized for specific use cases.

Alpine Linux is a lightweight and security-focused distribution. Choosing Alpine-based images as your base image can significantly reduce the image size compared to larger distributions.

Image tag	Image size	image:[version number]-alpine size
python:3.9.13	867.66 MB	46.71 MB
node:18.8.0	939.71 MB	164.38 MB
nginx:1.23.1	134.51 MB	22.13 MB

Figure 1: Alpine Image size.

Minimize Layers

Docker images are built in layers. Each instruction in a Docker File makes a layer. so, more instructions would contribute to more layers and results in more storage.

Consider the below Docker file, where we have multiple layers.

FROM ubuntu:latest
RUN apt-get update -y
RUN apt-get upgrade -y
RUN apt-get install vim -y
RUN apt-get install net-tools -y
RUN apt-get install dnsutils -y
RUN apt-get install iputils-ping -y

Command to Build Dockerfile:

time docker build -t multiple-layers --no-cache -f Dockerfile.

Time Taken for the Build: 0m53.754s

Docker image Size: 237 MB

Consider the below Docker file, where we install all the dependencies in a single layer.

FROM ubuntu:latest
*RUN apt-get update -y && *
*apt-get upgrade -y && *
apt-get install vim net-tools dnsutils iputils-ping -y

Command to Build Docker file:

time docker build -t single-layer--no-cache -f Dockerfile.

Time Taken for the Build: 0m47.365s

Docker image Size: 233 MB
From the above two scenarios, we could see the build time is reduced from 0m53.754s to 0m47.365s and the Docker image size is reduced from 237 MB to 233 MB.

Minimise Installing Debug Tools

Developers usually install debugging tools like curl, net-tools, iputils-ping etc. inside the Docker files for debugging purposes. These tools are mostly required in the development phase. Once the development is done, include only the necessary debug tools in the Docker file. The image size will further increase because of these debugging tools.

Consider the below Docker file, where we have multiple debug tools.

FROM ubuntu:latest
*RUN apt-get update -y && *
*apt-get upgrade -y && *
apt-get install vim net-tools dnsutils iputils-ping -y

Command to Build Docker file:
time docker build -t debug-tools –-no-cache -f Dockerfile.
Time Taken for the Build: 0m48.560s
Docker image Size: 233 MB

Consider the below Docker file, where we have the minimum debug tools.

FROM ubuntu:latest
*RUN apt-get update -y && *
*apt-get upgrade -y && *
apt-get install net-tools -y

Command to Build Dockerfile:
time docker build -t min-debug-tools --no-cache -f Dockerfile_without_debug_tools.
Time Taken for the Build: 0m19.931s
Docker image Size: 117 MB

From the above two scenarios, we could see the build time is reduced from 0m48.560s to 0m19.931s and the Docker image size is reduced from 233 MB to 117 MB.

USE no-install-recommends ON apt-get install:

Adding —no-install-recommends to apt-get install -y would install only the essential dependent packages and skip other recommendations.

Consider the below Docker file, where we are installing all the recommended packages.

FROM ubuntu:latest
*RUN apt-get update -y && *
*apt-get upgrade -y && *
apt-get install vim net-tools dnsutils iputils-ping -y

Command to Build Docker file:
time docker build -t all ls --no-cache -f Dockerfile .
Time Taken for the Build: 0m45.337s
Docker image Size: 233 MB

Consider the below Docker file, where we are installing only the required packages.

FROM ubuntu:latest
*RUN apt-get update -y && *
*apt-get upgrade -y && *
apt-get install --no-install-recommends vim net-tools dnsutils iputils-ping -y

Command to Build Docker file:
time docker build -t install-recommends --no-cache -f Dockerfile_with_no_istall.
Time Taken for the Build: 0m44.337s
Docker image Size: 233 MB

From the above two scenarios, we could see the build time is reduced from 0m45.337s to 0m44.337s, and the Docker image size is the same at 233 MB. If we install more packages, then we can see the difference in the image size as well.

Add rm -rf /var/lib/apt/lists/* after apt-get:

Clean up the installation files after the package is installed. Consider the below Docker file, where we are installing all the recommended packages.

FROM ubuntu:latest
RUN apt-get update -y
apt-get upgrade -y
apt-get install –no-install-recommends vim net-tools dnsutils iputils-ping -y

Command to Build Docker file:

time docker build -t no- cleanup –no-cache -f Dockerfile.

Time Taken for the Build:0m45.716s

Docker image Size: 233 MB

Consider the below Docker file, where we are removing the installation files after packages are installed.

FROM ubuntu:latest
*RUN apt-get update -y && *
*apt-get upgrade -y && *
*apt-get install --no-install-recommends vim net-tools dnsutils iputils-ping -y && *
*rm -rf /var/lib/apt/lists/**

Command to Build Dockerfile:

time docker build -t cleanup --no-cache -f Dockerfile

Time Taken for the Build : 0m46.645s

Docker image Size: 196 MB.

From the above two scenarios, we could see the Build time is increased because of cleanup from 0m45.716s to 0m46.645s and Docker image size is decreased from 233 MB to 196 MB.

Use the. dockerignore File

Using commands such as COPY. . in Docker file instructs Docker to copy all files and folders from your local directory to the Docker container. But all the files are not required for our application to run. For example, README,.git, .git etc

Create a `.dockerignore` file to exclude unnecessary files and directories from being copied into the Docker image. A.dockerignore file instructs Docker to skip files or directories during docker build. Files or directories that match in `. dockerignore` won't be copied with any ADD or COPY statements. As a result, they never appear in the final built image. This reduces the amount of data transferred to the image and the size of the image.

Below is a sample *.dockerignore* file

node_modules
git
github
gitignore
vscode
*dist/***
README.md

Sometimes we encounter a scenario where the docker image grows in size even though the Docker file did not change. This could be because of the COPY command in the Docker file. The size of the image increases because of the stuff you copy into the image.

Using Multi Stage Build:

Multistage builds allow you to slim Docker images. With multi-stage builds, you use multiple FROM statements in your Docker file. Each FROM instruction can use a different base, and each of them begins a new stage of the build. The final image of your application is created by copying code files and dependencies from

the previous stages. This means Docker will discard any intermediate files and build artifacts that are no longer needed to create your final build.

Below is a sample of a single Stage Docker file where we take a base Golang image and create a main.go file, build the go file and execute the binary

```
FROM golang:1.13
WORKDIR /src
COPY <<EOF ./main.go
package main
import "fmt"
func main() {
fmt.Println("hello, world")
}
EOF
RUN go build -o /bin/hello ./main.go
CMD ["/bin/hello"]
```

Command to Build Dockerfile:

```
time docker build -t singlestage --no-cache -f Dockerfile.
```

Time Taken for the Build: 0m7.510s

Docker image Size: 841 MB

The below Docker file has two separate stages: one for building a binary, and another where the binary gets copied from the first stage into the next stage

```
FROM golang:1.13
WORKDIR /src
COPY <<EOF ./main.go
package main
import "fmt"
func main() {
fmt.Println("hello, world")
}
EOF
RUN go build -o /bin/hello ./main.go
FROM scratch
COPY --from=0 /bin/hello /bin/hello
CMD ["/bin/hello"]
```

Command to Build Docker file:

```
time docker build -t cleanup --no-cache -f Dockerfile.
```

Time Taken for the Build: 0m9.734s

Docker image Size: 1.8 MB

From the above two scenarios, we could see the Build time is increased from 0m7.510s to 0m9.734s and Docker image size is decreased from 841 MB to 1.8 MB. We could optimize the image size more than 80% using this Docker multi build feature.

Using Docker Build Cache:

Each instruction in the Docker file translates to a layer in your final image. You can think of image layers as a stack, with each layer adding more content on top of the layers that came before it. Whenever a layer changes, that layer will need to be re-built and the subsequent layers also need to be built. So, it is a good practice to place instructions that are less likely to change towards the top to leverage Docker's layer caching mechanism.

Layers Cache?

```
FROM ubuntu:latest                                  ✓
RUN apt-get update \
  && apt-get install build-essentials               ✓
COPY main.c Makefile /src/                          ✗
WORKDIR /src                                        ✗
RUN make build                                      ✗
```

Consider the below Docker file, where we have a go file, and then we have a RUN command to install some packages. If the go file is changed, then all the packages need to be installed again.

FROM golang:1.13
WORKDIR /src
COPY main.go .
*RUN apt-get update -y && *
*apt-get upgrade -y && *
*apt-get install --no-install-recommends vim net-tools dnsutils iputils-ping -y && *
*rm -rf /var/lib/apt/lists/**
RUN go build -o /bin/hello ./main.go
CMD ["/bin/hello"]

Command to Build Dockerfile:
time docker build -t docker_cache -f Dockerfile_cache .
Time Taken for the Rebuild: 0m33.334s
Docker image Size: 930 MB

Consider below the Dockerfile where we have main. Go to the end and package the installation at the top. If the go file is updated, then we need to build from the go file cmd. Downloading the package installation can be skipped, as it is already present in the Docker cache.

FROM golang:1.13
WORKDIR /src
*RUN apt-get update -y && *
*apt-get upgrade -y && *
*apt-get install --no-install-recommends vim net-tools dnsutils iputils-ping -y && *
*rm -rf /var/lib/apt/lists/**
COPY main.go .
RUN go build -o /bin/hello ./main.go
CMD ["/bin/hello"]

Command to Build Dockerfile:
time docker build -t docker_cache -f Dockerfile .
Time Taken for the Rebuild: 0m10.334s
Docker image Size: 930 MB

From the above two scenarios, we could see the Build time is decreased from 0m33.334s to 0m10.334s. We could optimize more than 80% using this Docker cache. Often, we rebuild the images in our development environment. So, it is a good practice to leverage the Docker caching mechanism.

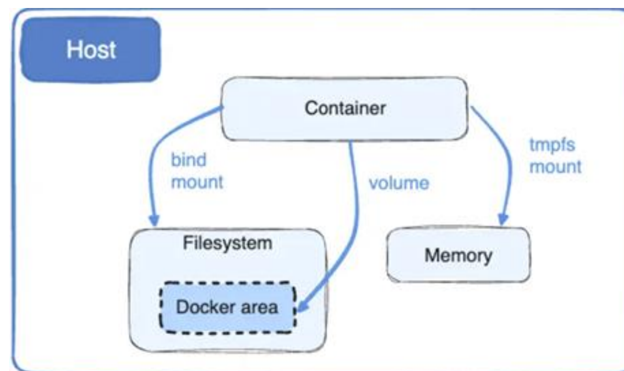
Note: The contents of build secrets are not part of the build cache. Changing the value of a secret doesn't result in cache invalidation.

Prefer volumes over the container's writable layer:

Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over binding mounts:

- Volumes are easier to back up or migrate than bind mounts.
- You can manage volumes using Docker CLI commands or the Docker API.
- Volumes work on both Linux and Windows containers.
- Volumes can be more safely shared among multiple containers.
- Volume drivers let you store volumes on remote hosts or cloud providers, encrypt the contents of volumes, or add other functionality.
- New volumes can have their content pre-populated by a container.
- Volumes on Docker Desktop have much higher performance than bind mounts from Mac and Windows hosts.

In addition, volumes are often a better choice than persisting data in a container's writable layer because a volume doesn't increase the size of the containers using it, and the volume's contents exist outside the lifecycle of a given container.



If your container generates non-persistent state data, consider using a tmpfs mount to avoid storing the data anywhere permanently and to increase the container's performance by avoiding writing into the container's writable layer.

Volumes are often a better choice than persisting data in a container's writable layer because a volume doesn't increase the size of the containers using it, and the volume's contents exist outside the lifecycle of a given container.

Decouple applications:

Limiting each container to one process is a good rule of thumb, but it's not a hard-and-fast rule. Each container should have only one concern. Decoupling applications into multiple containers makes it easier to scale horizontally and reuse containers. If containers depend on each other, you can use Docker container networks to ensure that these containers can communicate.

Sort multi-line arguments:

Whenever possible, sort multi-line arguments alphanumerically to make maintenance easier. This helps to avoid duplication of packages and make the list much easier to update. Here is an example.

```
RUN apt-get update && apt-get install -y \
bzr \
cvs \
git \
mercurial \
subversion \
&& rm -rf /var/lib/apt/lists/*
```

5. Conclusion

The values we see in the below table are based on the window's machine used and internet bandwidth available at the time of testing. These are not a benchmark but just a representation of what's possible,

Few advantages of implementing the techniques we discussed in this paper are**:**

- Docker file size can be reduced.
- The smaller the image is, the less complex it is as well.
- The less complex image contains fewer binaries and packages inside it
- There are fewer vulnerabilities in security scans.
- Faster Deployment times.
- Improved application performance.
- Low network bandwidth is used.

The table summarizes the test results with and without the recommendations.

Recommendation	Image size	Build time	Image size with Recommendation	Build time with Recommendation
Use a smaller base image	867.66 MB	0m59.504s	46.71MB	0m5.718s
Minimize Layers	237 MB	0m53.754s	233 MB	0m47.365s
Minimize installing debug tools	233 MB	0m48.560s	117 MB	0m19.931s
USE no-install-recommends ON apt-get install	233 MB	0m45.337s	233 MB	0m44.337s
Add rm -rf /var/lib/apt/lists/* after apt-get	233 MB	0m45.716s	196 MB	0m46.645s
Using Multi Stage Build	841 MB	0m7.510s	1.8 MB	0m9.734s
Using Docker Build Cache	930 MB	0m33.334s	930 MB	0m10.334s

References

[1]. https://www.ecloudcontrol.com/best-practices-to-reduce-docker-images-size/
[2]. https://devopscube.com/reduce-docker-image-size/
[3]. https://depot.dev/blog/how-to-reduce-your-docker-image-size
[4]. https://semaphoreci.com/blog/reduce-docker-image-size
[5]. https://medium.com/@reach2shristi.81/real-life-proven-strategies-to-reduce-docker-image-size-0415b756f886
[6]. https://docs.docker.com/build/building/best-practices/
[7]. https://docs.docker.com/build/building/packaging/
[8]. https://docs.docker.com/reference/dockerfile/