Tutorial: Docker for Research Artifacts

Dennis Sprokholt
March 2022

1 Introduction

Docker packs programs together with their dependencies. Through OS-level virtualization it ensures the program runs anywhere\(^1\). While a plethora of Docker tutorials exists online, most teach you to deploy web applications. Instead, I explain how Docker can build reusable research artifacts.

2 Docker Overview

We consider two of Docker’s most predominant objects – which are often confused with each other:

- **image** - An image is a read-only snapshot of a system, which contains an Operating System\(^2\), program dependencies, and your program. You can share an image with the world.

- **container** - A container is a writable copy of an image, whose function is similar to the storage drive in your machine. A container is runnable. Note that a container which is not running still differs from an image.

In the following sections I explain how to setup a Hello World program – written in C – inside a Docker container. (See hello.c in Appendix A)

3 Dockerfiles

A Dockerfile contains the build instructions for an image. Let’s construct it in steps.

- Docker images build upon other images. Often you start with an Operating System image, such as a build of the Debian distro.

  ```bash
  FROM debian:bullseye-slim
  ```

- Secondly, we install the build dependencies. In Debian, we install packages with `apt`. `libc6-dev` contains the C standard libraries, while `gcc` can compile our C program.

  ```bash
  RUN apt-get update &&
      apt-get install -y --no-install-recommends libc6-dev gcc &&
      rm -rf /var/lib/apt/lists/*
  ```

\(^1\)Assuming identical CPU architectures: https://docs.docker.com/desktop/multi-arch/

\(^2\)Actually, it virtualizes the OS: https://www.docker.com/resources/what-container
As we don’t have terminal access to the image while building, \texttt{-y} automatically confirms installations. \texttt{--no-install-recommends} prevents installing recommended dependencies. On the final line, we remove the local package cache, as it unnecessarily increases our image size.

- Thirdly, we tell Docker to execute future commands from inside the image’s \texttt{root/} directory.

\begin{verbatim}
WORKDIR root/
\end{verbatim}

- Then we copy \texttt{hello.c} from the current \texttt{host} directory into the image (at \texttt{root/hello.c}).

\begin{verbatim}
COPY hello.c .
\end{verbatim}

- Finally, we call \texttt{gcc} to build our program.

\begin{verbatim}
RUN gcc hello.c -o hello
\end{verbatim}

You can see the complete Dockerfile in Appendix B.

4 Building the image

The Dockerfile describes the build process. We still have to execute it. Inside the directory with the Dockerfile (and \texttt{hello.c}), execute:

\begin{verbatim}
$ docker build . --tag=helloworld
\end{verbatim}

This command executes the steps described in the Dockerfile and creates a repository named \texttt{helloworld} with an image tagged \texttt{latest}.

Your image should now appear in Docker’s image list:

\begin{verbatim}
$ docker image ls
REPOSITORY TAG IMAGE ID CREATED SIZE
helloworld latest d32c405f284a 10 seconds ago 237 MB
\end{verbatim}

5 Running the image

Inside our Dockerfile, we told Docker to compile our \texttt{hello.c} with \texttt{gcc} into the \texttt{hello} executable. Now our image contains the executable file \texttt{root/hello}. We execute it inside a container:

\begin{verbatim}
$ docker run -it --rm helloworld ./hello
Hello World!
\end{verbatim}

The command \texttt{docker run} spawns a container from the \texttt{helloworld} image, and executes \texttt{./hello} inside. The \texttt{-it} flag ensures the container runs in interactive terminal mode; It effectively links your terminal to the container’s stdin and stdout. The \texttt{--rm} flag removes the container (but not the image) from your storage drive upon exiting.

Now you are familiar with the basics and should be able to create images for your own artifacts. Good luck!
(Extra) Multi-stage builds

When publishing artifacts, I prefer to keep Docker images small; this avoids unnecessarily wasting time on downloads. *Multi-stage builds*\(^3\) can help with that.

For our program, **building** `hello` required `gcc`, but **executing** it does not. We can create another clean Docker image and **copy** the executable into it.

```
Multi-stage Dockerfile

FROM debian:bullseye-slim AS build
... # the same as before

FROM debian:bullseye-slim
COPY --from=build /root/hello /root/hello
WORKDIR root/
```

In this case, dropping the build dependencies reduces the image from 237MB to 80MB.

See Appendix C for the complete Dockerfile.

(Extra) Tips and Tricks

- **Pick small base images** - This ensures your final images are smaller. For instance, Debian\(^4\) has large images (e.g., `bullseye` - 124MB) and small images (e.g., `bullseye-slim` - 80MB). These “slim” variants exclude unnecessary files, such as man pages.

- **Keep images around** - Building with the same Dockerfile at different times may produce different images. If the behavior of a command changes in the future, it will have a different effect on your image. Consider:

  ```
  $ apt-get install gcc
  ```

  While it installs version 10.2.1 now, next year it could install version 11.0.0. A research artifacts should reproduce the same results for years to come, and dependencies are **not always** backward compatible. The Dockerfile does not guarantee reproducible builds. However, the image is a system snapshot. So, keep it around.

- **Export your image** - Share your artifact with a `.tar.gz` archive, which you create with:

  ```
  $ docker save helloworld | gzip --best > artifact.tar.gz
  ```

  Others then import it with:

  ```
  $ docker load -i artifact.tar.gz
  ```

  Note that it maintains its original image name `helloworld:latest`.

\(^3\)See also: https://docs.docker.com/develop/develop-images/multistage-build/
\(^4\)See also: https://hub.docker.com/_/debian
A  Hello World Program

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    printf("Hello World!\n");
    return EXIT_SUCCESS;
}
```

B  Dockerfile

```dockerfile
FROM debian:bullseye-slim
RUN apt-get update &&
    apt-get install -y --no-install-recommends libc6-dev gcc &&
    rm -rf /var/lib/apt/lists/*
WORKDIR root/
COPY hello.c .
RUN gcc hello.c -o hello
```

C  Multi-stage Dockerfile

```dockerfile
FROM debian:bullseye-slim AS build
RUN apt-get update &&
    apt-get install -y --no-install-recommends libc6-dev gcc &&
    rm -rf /var/lib/apt/lists/*
WORKDIR root/
COPY hello.c .
RUN gcc hello.c -o hello

FROM debian:bullseye-slim
COPY --from=build /root/hello /root/hello
WORKDIR root/
```