Dockerized Build Environments for C/C++ Projects

...

Dima Danilov

Who Am I?

SENIOR SOFTWARE ENGINEER

GK8 *- blockchain and digital asset security*
VMWare *- blockchain development*
LiveU *- HD live video streaming*

TECHNOLOGIES

C++, Rust, Python, Network programming, Distributed systems

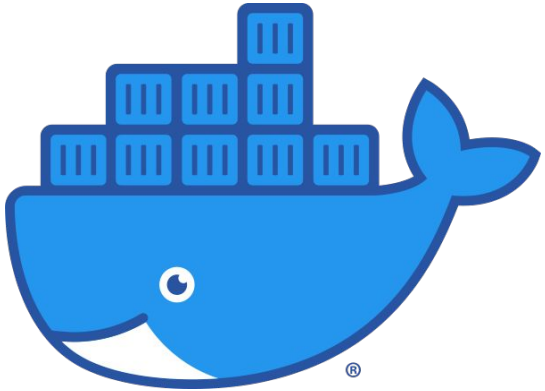
HOBBIES

Vim/Neovim fine-tuning

Common Problems when Building C/C++ Projects

- No standard dependency management
  - OS package managers
  - Manually build/install libraries as “make install”
  - Git submodules and build within a source tree
  - Conan, Hunter, Build2, etc
- Tool management
  - Which compiler version is necessary?
  - Which linker should be used?
- Differences between the CI and dev station environments
  - Dependencies are not updated automatically
  - Dirty local environment
  - Constant “works on my machine” excuses

Single Isolated Reproducible Build Environment



- **Single** – one environment used by both CI and dev stations
- **Isolated** - no influence from local packages, applications, etc
- **Reproducible** - build consistency across versions

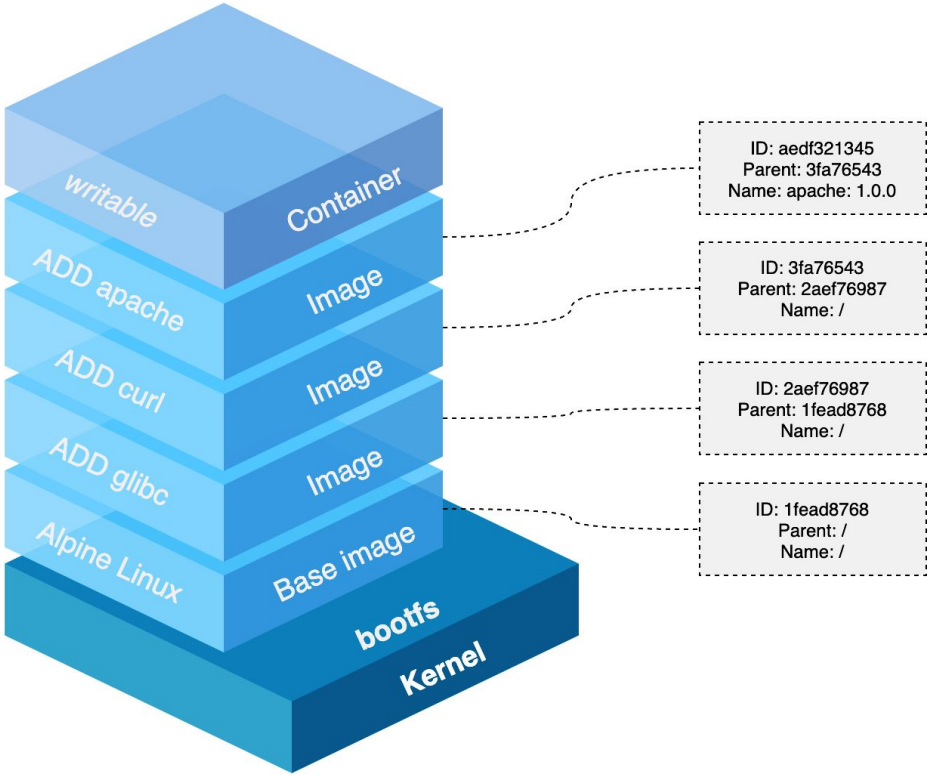
What is Docker?

Docker is a set of products that use OS-level virtualization to deliver software in packages called containers.

- Software inside a container runs on the host Linux kernel
- The processes are isolated by utilization of kernel features as follows:
  - Control groups (allows limiting an application to a specific set of hardware resources)
  - Namespaces:
    - The pid namespace: Process isolation (PID: Process ID).
    - The net namespace: Managing network interfaces (NET: Networking).
    - The ipc namespace: Managing access to IPC resources (IPC: InterProcess Communication).
    - The mnt namespace: Managing filesystem mount points (MNT: Mount).
    - The uts namespace: Isolating kernel and version identifiers. (UTS: Unix Timesharing System).
  - OverlayFS/UnionFS (file systems that operate by creating layers, making them very lightweight and fast)

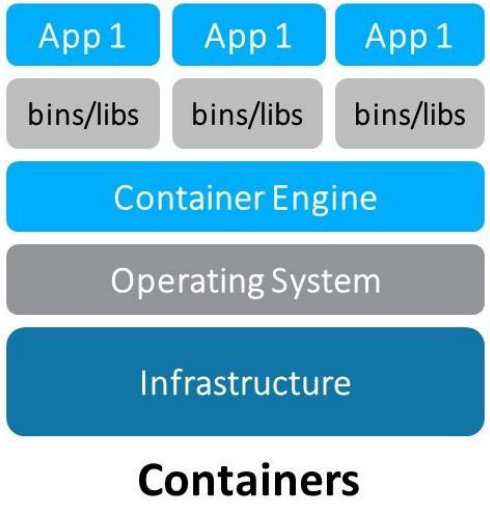
Docker Image

- Contains different layers, which are all read-only. Every layer has an ID and can contain “parent IDs” of underlying images.
- Every new layer is on top of the older layers and can “overwrite” files of the lower layers.
- Every command in the Dockerfile definition will create a new layer image.
- Can be shared between containers



Docker container

- Uses images as read-only file system
- Has a small writable runtime file system
- Runs on hosts kernel instance
- Isolated namespace



Example Application: Code

Simple application with one 3rd party library

```
#include <boost/filesystem/operations.hpp>
#include <iostream>

int main(int argc, char *argv[]) {
    std::cout << "The size of " << boost::filesystem::absolute(argv[0])
              << " is " << boost::filesystem::file_size(argv[0]) << '\n';
    return 0;
}
```

Example Application: CMake

Boost is linked statically since it is required if the target machine does not have the right version of Boost pre-installed; this recommendation applies to all dependencies pre-installed in the docker image.

```
cmake_minimum_required(VERSION 3.10.2)

project(a.out)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# Remove for compiler-specific features
set(CMAKE_CXX_EXTENSIONS OFF)

string(APPEND CMAKE_CXX_FLAGS " -Wall")
string(APPEND CMAKE_CXX_FLAGS " -Wbuiltin-macro-redefined")
string(APPEND CMAKE_CXX_FLAGS " -pedantic")
string(APPEND CMAKE_CXX_FLAGS " -Werror")

# clangd completion
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)

include_directories(${CMAKE_SOURCE_DIR})
file(GLOB SOURCES "${CMAKE_SOURCE_DIR}/*.cpp")

add_executable(${PROJECT_NAME} ${SOURCES})

set(Boost_USE_STATIC_LIBS        ON) # only find static libs
set(Boost_USE_MULTITHREADED      ON)
set(Boost_USE_STATIC_RUNTIME    OFF) # do not look for boost libraries linked ag

find_package(Boost REQUIRED COMPONENTS filesystem)

target_link_libraries(${PROJECT_NAME}
    Boost::filesystem
)
```

Prepare and Build a Docker Image

```
FROM ubuntu:18.04
LABEL Description="Build environment"

ENV HOME /root

SHELL ["/bin/bash", "-c"]

RUN apt-get update && apt-get -y --no-install-recommends install \
    build-essential \
    clang \
    cmake \
    gdb \
    wget

# Let us add some heavy dependency
RUN cd ${HOME} && \
    wget --no-check-certificate --quiet \
        https://boostorg.jfrog.io/artifactory/main/release/1.77.0/source/boost_1
        tar xzf ./boost_1_77_0.tar.gz && \
        cd ./boost_1_77_0 && \
        ./bootstrap.sh && \
        ./b2 install && \
        cd .. && \
        rm -rf ./boost_1_77_0
```

- The image is based on Ubuntu 18.04 LTS
- Contains a minimum set of tools for building a C++ project
- Boost is used as a dependency for our example application

Build the Docker Image and Project Inside

```
$ docker build -t example/example_build:0.1 -f DockerfileBuildEnv .
Here is supposed to be a long output of boost build
```

- `-t` - image name and version
- `-f` - <path to Dockerfile>
- `.` - path to the context

```
$ cd project
$ docker run -it --rm --name=example \
        --mount type=bind,source=${PWD},target=/src \
        example/example_build:0.1 \
        bash
```

- `--mount` - instructs Docker to mount the current source directory to the container

Make the Environment Re-usable

- Docker commands are hard to remember
- The idea is to wrap the docker commands in a Makefile
- Make is a tool that most usable tool by C/C++ developers

Integrate Makefile into an Existing Project

I have created a template of a Makefile working with docker.
More about its usage can be found in my blog post.

Advanced features

- Most modern IDE/text editors support docker build
  - CLion
  - Visual Studio Code
- Run tests in docker

Resources

- Example Makefile
- Example of using the Makefile
- ddanilov.me (my blog and the source for this presentation)
- Github

Thank you for listening!