Advanced Software Development: Docker

CPSC 501: Advanced Programming Techniques
Winter 2025

Jonathan Hudson, Ph.D
Assistant Professor (Teaching)
Department of Computer Science
University of Calgary

Wednesday, January 8, 2025

UNIVERSITY OF CALGARY

Containerization

History

- 1979 – Unix -> idea of process having unique root dirs (diff. views of system)

- 2000-2011 – Container ideas grow in unix type systems
  - ‘Jails’ to partition resource of system
  - Snapshot and cloning of containers
  - Google process containers become part of kernel as cgroups
  - LXC (2008) linux containers (cgroups and linux namespaces)
  - Warden (2011) API for container management

History

- 2013 – Docker – like Warden but ecosystem ended up growing fast
- 2016-2017 – strong evidence of security provided by containerizing applications, and strong support across dev. ops and cloud tools. Continuous integration and development built around using containers.
- By 2018 gold standard of modern soft. infrastructure needs understanding of containerization (Kubernetes strong growth -> auto deploy/scale containers)

Motivation Story

- You make an application
- It uses Node.js/Express.js/SQLite3 (All rather lightweight)
- But maybe Node.js is based on version that needs Python 3 and C/C++ compiler
- While Python 3 is management install C/C++ compilers environments are certainly not

Handling C/C++

- Setting-up C/C++ tool-chain is pretty easy on Linux but on Windows and Mac, it's a painful task.
- On Windows, the C++ build tools package measures at gigabytes and takes quite some time to install.
- On a Mac, you can either install the gigantic Xcode application or the much smaller Command Line Tools for Xcode package.
- Regardless of the one you install, it still may break on OS updates.
- In fact, the problem is so prevalent that there are Installation notes for macOS Catalina available on the official repository.

Are things now solved?

- What if you have a teammate who uses Windows while you're using Linux.
- Now you have to consider the inconsistencies of how these two different operating systems handle paths.
- Or the fact that popular technologies are not well optimized to run on Windows.
- Even if you get through the entire development phase, but the person responsible for managing the servers follows the wrong deployment procedure?

Solution

1. Develop and run the application inside an isolated environment (known as a container) that matches your final deployment environment.
2. Put your application inside a single file (known as an image) along with all its dependencies and necessary deployment configurations.
3. And share that image through a central server (known as a registry) that is accessible by anyone with proper authorization.

- Containerization: Putting your applications inside a self-contained package making it portable and reproducible across various environments.

Docker

Docker

- Docker is one containerization management platform
- Containers can be private or publicly stored an shared
- Docker allows you to orchestrate (access and deploy containers)
- Others include Podman, Kaniko, rkt

Kubernetes

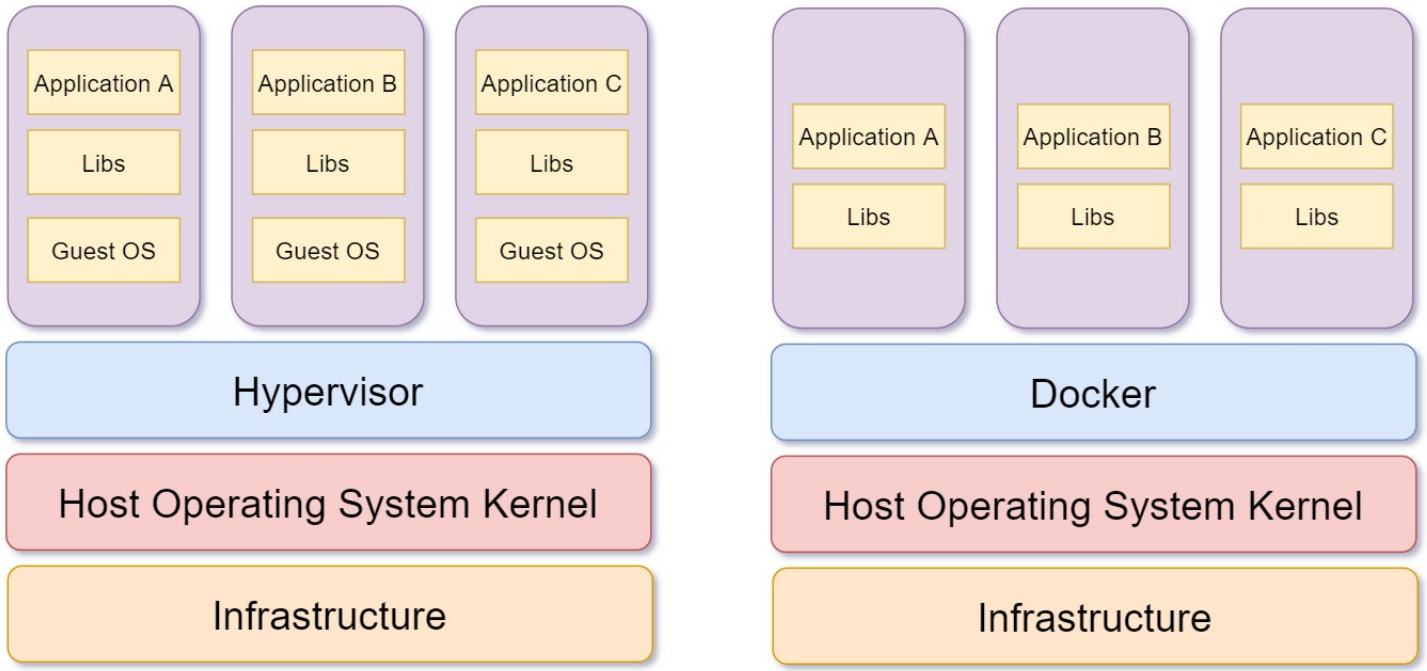
- Kubernetes is an open-source container orchestration system for automating software deployment, scaling, and management. Google originally designed Kubernetes, but the Cloud Native Computing Foundation now maintains the project.
- Amazon, Google, IBM, Microsoft, Oracle, Red Hat, SUSE, Platform9 and VMware offer Kubernetes-based platforms or infrastructure as a service (IaaS) that deploy Kubernetes.

Installation

- To use Docker you need to install the tool that handles everything
- https://docker-handbook.farhan.dev/installing-docker
- These installs can produce easy to view GUI tool to see what is live and running on your system, but can just as easily be managed through terminal which is great for remote/cloud deployment needs

Container versus VM

- Instead of having a full OS in a VM, a container instead access host OS through the limiting containerization environment (Docker)
- Still maintains isolation like a VM

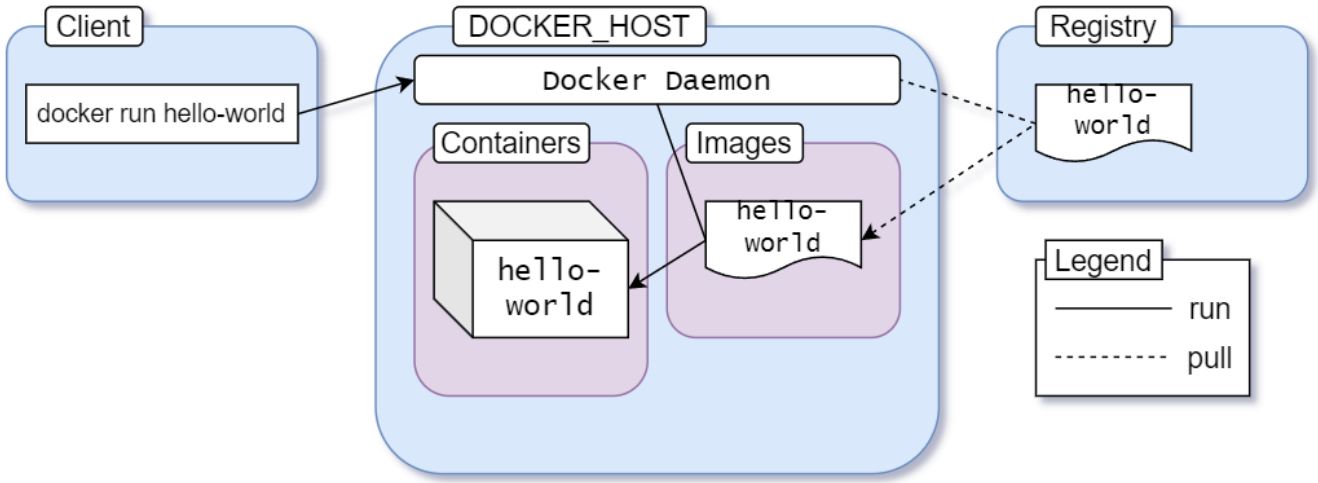


Terminology

- Image
  - Multi-layered files that act as templates to make containers
  - Frozen-read only copies of a container
  - OCI (open container initiative) as standardized this
- Containers
  - Image in a running state (writable layer on top of read-only image)
- Registry
  - Stores images (DockerHub), can download freely
  - Example there are Data Science images hosted that install 10s/100s of common packages
  - Instead of managing each individual computer install I could register a common image for a course and have everyone use it with the required tools

Terminology

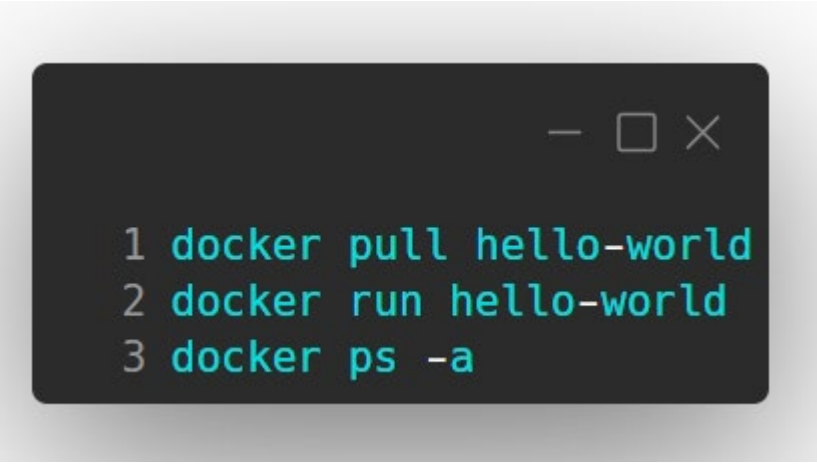
- Docker Daemon
  - Sits around in background waiting for commands to manage containers
- Docker Client
  - Takes commands from user
- REST API
  - Bridge between client and daemon



Hello, world!

Installation

- To use Docker you need to install the tool that handles everything
- https://docker-handbook.farhan.dev/installing-docker
- These installs can produce easy to view GUI tool to see what is live and running on your system, but can just as easily be managed through terminal which is great for remote/cloud deployment needs



Basics

docker <object-type> <command> <options>

- object-type indicates the type of Docker object you'll be manipulating. This can be a container, image, network or volume object.
- command indicates the task to be carried out by the daemon i.e. run command.
- options can be any valid parameter that can override the default behavior of the command i.e. the --publish option for port mapping.

docker container run <image name>

docker container run fhsinchy/hello-dock

Pull from registry and run

Isolation and Ports

Containers are isolated environments by default.

If you need outside to access inside (like connecting a DB container to another) you need to publish ports

--publish <host>:<container>

docker container run --publish 8080:80 fhsinchy/hello-dock

http://127.0.0.1:8080/

You should see The Docker Handbook and your browser

This is on your local system

Commands

Detaching/Listing/Naming

Disconnect container from the terminal that launched it

docker container run --detach --publish 8080:80 fhsinchy/hello-dock

List containers running

docker container ls

List all containers that have run or are running

docker container ls –all

Name

docker container run --detach --publish 8080:80 --name hello-dock fhsinchy/hello-dock

Rename

docker container rename <container identifier> <new name>

Stopping/Restarting/Create

Stop running container (easiest if named) (SIGTERM)

docker container stop <identifier>

With prejudice (SIGKILL)

docker container kill <identifier>

Restart previously ran container (retains previous config for ports)

docker container start <identifier>

Like previous but will stop first if it is running

docker container restart <identifier>

Create without running

docker container create --publish 8080:80 fhsinchy/hello-dock

Prune/Interactive

Remove one

docker container rm <identifier>

Remove all inactive

docker container prune

After launch leave terminal ‘inside’ of the container

docker container run --rm -it ubuntu

This command leaves us with a ubuntu container on our system with a fully functional terminal connection (interactive **it**)

-rm means remove after stopped

Commands Inside

docker container run <image name> <command>

docker container run --rm busybox sh -c "echo -n my-secret | base64“

*Access **busybox** container and run*

sh -c "echo -n my-secret | base64“

On terminal inside of container

Files

What about files?

By default the container can’t see the host file system

Need to map/bind in host locations to virtual locations to enable access

--volume <local dir>:<container dir>:<read write access>

Example

-v $(pwd):/zone

Bind present working directory to folder zone (linux variant)

What about files?

Let’s consider our container **fhsinchy/rmbyext** has a program in a folder called zone.

docker pull fhsinchy/rmbyext

The program **rmbyext** let’s us delete files in local directory with a given extension **pdf**

docker container run --rm fhsinchy/rmbyext pdf

If we run our container that program is going to run **rmbyext** whenever it is started

```
NO PDF FILES TO REMOVE.
```

However, there are no files in container folder ‘zone’ to delete

IMAGE LAYERS

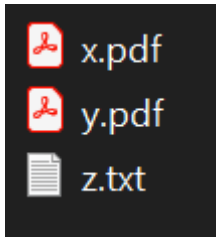
1	**ADD file ... in /**	**2.67 MB**
2	CMD ["/bin/sh"]	0 B
3	ENV PATH=/usr/local/bin:/usr/local/sbin:/usr/local/bin:/…	0 B
4	ENV LANG=C.UTF-8	0 B
5	/bin/sh -c set -eux; apk	638.37 KB
6	ENV GPG_KEY=E3FF2839C048B25C084DEBE9B26995E310250568	0 B
7	ENV PYTHON_VERSION=3.9.1	0 B
8	/bin/sh -c set -ex &&	11.02 MB
9	/bin/sh -c cd /usr/local/bin &&	229 B
10	ENV PYTHON_PIP_VERSION=20.3.3	0 B
11	ENV PYTHON_GET_PIP_URL=https://github.com/pypa/get-pip/r…	0 B
12	ENV PYTHON_GET_PIP_SHA256=6a0b13826862f33c13b614a921d362…	0 B
13	/bin/sh -c set -ex; wget	2.04 MB
14	CMD ["python3"]	0 B
15	WORKDIR /zone	93 B
16	/bin/sh -c apk add --no-cache	2.52 MB
17	ENTRYPOINT ["rmbyext"]	0 B

What about files?

```
NO PDF FILES TO REMOVE.
```

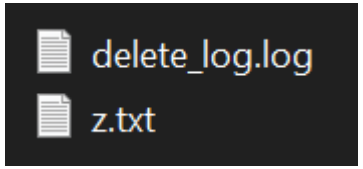
However, there are no files in container folder ‘zone’ to delete

We’ll make a host directory, add some files



then run our command from that directory (but with binding)

```
Removing: PDF
x.pdf
y.pdf
```



docker container run --rm **-v “%cd%”:/zone** fhsinchy/rmbyext pdf

docker container run --rm **-v ${PWD}:/zone** fhsinchy/rmbyext pdf

docker container run --rm **-v $(pwd):/zone** fhsinchy/rmbyext pdf

Images

Dockerfile

A docker container is created from an ‘image’ description of the steps needed to setup the container

Containers can be built on top of each other by using

FROM

Ex.

From ubuntu:latest

Will build on the latest version of the ubuntu image docker.com has

Dockerfile

```
FROM ubuntu:latest
EXPOSE 80
RUN apt-get update && \
        apt-get install nginx -y && \
        apt-get clean && rm -rf /var/lib/apt/lists/*
CMD ["nginx", "-g", "daemon off;"]
```

This is a custom DockerFile that exposes port 80

Then installs nginx and runs it (nginx is a proxy server)

Dockerfile

Now make sure this file is saved in your local directory

Build the image with

docker image build .

This will give you an image name like

Successfully built 3199372aa3fc

That you can run

docker container run --rm --detach --name custom-nginx-packaged --publish 8080:80 3199372aa3fc

Access it at http://127.0.0.1:8080

To help with naming….

docker image build --tag custom-nginx:packaged .

Dockerfile

A docker image is a multi-layer idea

Each command in DockerFile creates a new read-only layer

When you run the image it create yet another layer

This all works based on ‘union file system’ which allows the branches of file system to be overlaid yet treated as one single coherent virtual file system

This avoids data duplication, and lets you keep this layer history

Some very good information on optimizing image file size to help with deployment and portability https://docker-handbook.farhan.dev/image-manipulation-basics

IMAGE LAYERS

1	**ADD file ... in /**	**2.67 MB**
2	CMD ["/bin/sh"]	0 B
3	ENV PATH=/usr/local/bin:/usr/local/sbin:/usr/local/bin:/…	0 B
4	ENV LANG=C.UTF-8	0 B
5	/bin/sh -c set -eux; apk	638.37 KB
6	ENV GPG_KEY=E3FF2839C048B25C084DEBE9B26995E310250568	0 B
7	ENV PYTHON_VERSION=3.9.1	0 B
8	/bin/sh -c set -ex &&	11.02 MB
9	/bin/sh -c cd /usr/local/bin &&	229 B
10	ENV PYTHON_PIP_VERSION=20.3.3	0 B
11	ENV PYTHON_GET_PIP_URL=https://github.com/pypa/get-pip/r…	0 B
12	ENV PYTHON_GET_PIP_SHA256=6a0b13826862f33c13b614a921d362…	0 B
13	/bin/sh -c set -ex; wget	2.04 MB
14	CMD ["python3"]	0 B
15	WORKDIR /zone	93 B
16	/bin/sh -c apk add --no-cache	2.52 MB
17	ENTRYPOINT ["rmbyext"]	0 B

Network

Networking

Docker containers can exist in their own ‘**bridge**’ networks

Or see those whole ‘**host**’ network

Or have ‘**none**’ for no network access

Or also have externalized network access through ‘**overlay**’ or ‘**macvlan**’

Most common isolated usage is ‘**bridge**’

Networking

There is a default bridge made for all containers

docker network ls

*# c2e59f2b96bd **bridge** bridge local*

*# 124dccee067f **host** host local*

*# 506e3822bf1f **none** null local*

docker network create skynet

docker network ls

*# c2e59f2b96bd **bridge** bridge local*

*# 124dccee067f **host** host local*

*# 506e3822bf1f **none** null local*

 # 7bd5f351aa89 skynet bridge local

Networking

You can network different containers by adding them to a network

Ex.

docker network connect skynet hello-dock

Or by run running them attached to network

docker container run --network skynet --rm --name alpine-box -it alpine sh

The previous command drops us into shell where we can ping the other connected container, usefully docker will resolve the internal names we had so we don’t need to do work to determine ip of each service container we ran

ping hello-dock

Compose

Too many containers?

At a certain complexity of docker management you'll find you have to type a number of commands each time to set up the network of containers

Docker-compose is an application that reads **docker-compose.yml** files to run multiple docker commands

docker-compose.yml

Two different containers are made in this file, with the SQL todo-mysq stored in a volume made for it

```
version: "3.8"
services:
  app:
    …
  mysql:
    …
volumes:
  todo-mysql-data:
```

DB container

Database **db** image based on **mysql**, we use a versioned imaged, indicate a mapping for database, and environment variables for database

```
mysql:
  image: mysql:5.7
  volumes:
    - todo-mysql-data:/var/lib/mysql
  environment:
    MYSQL_ROOT_PASSWORD: secret
    MYSQL_DATABASE: todos
```

APP container

```
app:
  image: node:12-alpine
  command: sh -c "yarn install && yarn run dev"
  ports:
   - 3000:3000
  working_dir: /app
  volumes:
   - ./app:/app
  environment:
   MYSQL_HOST: mysql
   MYSQL_USER: root
   MYSQL_PASSWORD: secret
   MYSQL_DB: todos
```

Our **app** container is much more complicated, we could actually put all the commands in Dockerfile as well but instead put it here directly as shell command

We setup environment variables to match database

Map volumes for the app data

And expose port 3000

Networking?

Networking isn’t declared in that docker-compose.yml file

Docker-compose actually creates a default bridge when docker-compose is run for the contained services automatically

However, there is support for networks (can isolate internal services)

At general .yml description

networks:

frontend:

name: <name of network>

driver: bridge

In a service

networks:

- frontend

Commands?

Start up the set of services (in detached mode)

docker-compose --file docker-compose.yml up –detach

See running

docker-compose ps

Execute service specific command

docker-compose exec <service name> <command>

Done?

docker-compose down --volumes

Why again?

Why? (just a few of reasons)

Development

- Coding application and need to compile/test/dev against different operating systems
- Use containers to represent these environment builds and compile within each to create artifacts or verify dev work

Deployment

- No longer concern yourself with ‘installing’ application
- Download docker image, deploy docker image with application already setup
- No worry about breaking local environment or different variants of installs

Scale

- Image can define a service ability
- As more load balancing is needed for service, use deployment tool like Kubernetes to initiate more docker instances in cloud environment to assist

Onward to … Continuous Integration/Development.

Jonathan Hudson
jwhudson@ucalgary.ca
https://pages.cpsc.ucalgary.ca/~jwhudson/

UNIVERSITY OF CALGARY