

Containers

Software Architecture

Brae Webb

March 6, 2023



Question

What is a *container*?

Question

What is a *container*?

Answer

A way of *packaging software* and its dependencies such that the software can be run in numerous environments.

Okay...

How hard could that be?

Packaging software

```
1 #!/usr/bin/env python3  
2  
3 import numpy as np  
4 import re  
5  
6 my_arr = np.array([5, 2, 9, 7, 3])  
7 max_element = np.max(my_arr)  
8  
9 duplicated_max = re.sub(".*", f"{max_element}", "X")  
10 print(sum(int(x) for x in duplicated_max))
```

```
> ./program.py
```

18

demo

Transferring this software to Richard.

```
> ./program.py  
/usr/bin/env: 'python3': No such file or directory
```

```
> ./program.py  
/usr/bin/env: 'python3': No such file or directory
```

No Python interpreter installed, have to install Python and all it's dependencies.


```
> ./program.py
File "./program.py", line 9
    duplicated_max = re.sub('.*', f'{max_element}', "X")
                                         ^
SyntaxError: invalid syntax
```

f-strings aren't supported in Python 3.5! Have to upgrade to Python 3.6.

```
> ./program.py
Traceback (most recent call last):
  File "./program.py", line 3, in <module>
    import numpy as np
ModuleNotFoundError: No module named 'numpy'
```

```
> ./program.py
Traceback (most recent call last):
  File "./program.py", line 3, in <module>
    import numpy as np
ModuleNotFoundError: No module named 'numpy'
```

A Python dependency used by our code isn't installed. Have to install numpy (hopefully the right version...).

```
> ./program.py  
9
```

```
> ./program.py  
9
```

???

Question

Not so easy... what do we need?

A wall

A big wall around our environment so that we *know what software* we are actually depending upon.





A package

A way to box up all your software and dependencies so that it can be *transferred* and *run* in a different environment.

*§ A History of Containers*¹

¹This is a very linux focused history — container technology also exists in the Windows world.

1979

Unix Version 7

Introducing... *chroot*



demo

Exploring *chroot*

Exploring *chroot*

```
> mkdir ./jail
> cd jail
> mkdir bin
> cp /bin/ls bin
> chroot . /bin/ls
chroot: failed to run command '/bin/ls': No such file or directory
```

Exploring *chroot*

```
> ldd /bin/ls
    libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0
                           x00007f0097135000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f0096f0d000)
    libpcre2-8.so.0 => /lib/x86_64-linux-gnu/libpcre2-8.so.0 (0
                           x00007f0096e76000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f0097189000)
> cp --parents /lib/x86_64-linux-gnu/libselinux.so.1 /lib/x86_64-
    linux-gnu/libc.so.6 /lib/x86_64-linux-gnu/libpcre2-8.so.0 /lib64/
    ld-linux-x86-64.so.2 .
> ls
bin lib lib64
> ls lib/x86_64-linux-gnu/
libc.so.6 libpcre2-8.so.0 libselinux.so.1
```

Exploring *chroot*

```
> chroot . /bin/ls
bin lib lib64
> chroot . /bin/ls /
bin lib lib64
> chroot . /bin/ls ..
bin lib lib64
> chroot . /bin/ls /bin
ls
```

Chroot Limitations

- *Only filesystem isolation* — processes, network, etc. still accessible.
- Not very *user friendly*.
- Not very *portable*.
- *Jailbreak* is possible.

1992



Plan 9

Introducing...
layered filesystem

Layered filesystem

- Projection on *read*.
- Copy on *write*.

Projection on *read*



```
> ls  
passwords.txt help.md diary.md
```

Copy on *write*

```
> echo "1234" >> passwords.txt
```



demo

Exploring a *layered filesystem*

Exploring a *layered filesystem*

```
> mkdir lower upper worker merged  
> echo "password1234" >> lower/passwords.txt  
> touch lower/help.md upper/diary.md  
> mount -t overlay -o lowerdir=lower,upperdir=upper,workdir=worker  
none merged
```

Exploring a *layered filesystem*

```
> ls merged
diary.md help.md passwords.txt
> ls upper
diary.md
> ls lower
> cat lower/passwords.txt
password1234
> echo "1234" >> merged/passwords.txt
> cat merged/passwords.txt
password1234
1234
> ls upper
diary.md passwords.txt
> cat lower/passwords.txt
password1234
```

2002

Linux kernel 2.4.19

Introducing...
namespaces



Linux Namespaces

2002 *Mount* namespace.

2006 *Unix Time-Sharing* namespace.

2006 *Inter-process Communication* namespace.

2008 *Process ID* namespace.

2009 *Network* namespace.

2013 *User* namespace.

2016 *Control group* namespace.

2008

LinuX Containers (LXC)



2013



PyCon 2013

Introducing...
Docker

Docker was the magic that made Linux containers usable for mere mortals.

- *Nigel Poulton*

§ The Language of Containers

Definition 1. Container

A *running process* created from a container image.
Typically isolated from the host system.

Definition 2. Container Image

A set of files that can be used to *create a container*.

Container Image



```
> docker run -it ubuntu /bin/bash  
root@f2b0b0c0b0b0:/# ls  
bin dev home lib64 mnt proc run srv  
root@f2b0b0c0b0b0:/# exit
```

```
> docker run -it ubuntu /bin/ls  
bin dev home lib64 mnt proc run srv
```

Definition 3. Container Engine

A tool to *create* and *manage* containers. Often also manages container images.

Container Engines

- Docker
- rkt
- LCX
- runC
- Containerd
- CRI-O
- Podman

Creating a container image

To create a container image, we need to *create a collection of image layers*.

Fortunately, this is no longer a manual process...

Creating a container image

To create a container image, we need to *create a collection of image layers*.

Fortunately, this is no longer a manual process...

Instead we use *build file*, or image blueprints.

Definition 4. Build File

A file of the *instructions* for *creating a container image*.

Build File

```
>> cat Dockerfile  
1 FROM ubuntu  
2 RUN apt-get update  
3 RUN apt-get install -y cowsay  
4 CMD ["/usr/games/cowsay", "Hello World"]
```

Build File

```
>> cat Dockerfile  
1 FROM ubuntu  
2 RUN apt-get update  
3 RUN apt-get install -y cowsay  
4 CMD ["/usr/games/cowsay", "Hello World"]
```

```
> docker build -t cowsay .
```

Build File

```
>> cat Dockerfile  
1 FROM ubuntu  
2 RUN apt-get update  
3 RUN apt-get install -y cowsay  
4 CMD ["/usr/games/cowsay", "Hello World"]
```

```
> docker build -t cowsay .
```

```
> docker run cowsay
```

```
>> cat Dockerfile  
  
1 FROM ubuntu  
2 RUN apt-get update  
3 RUN apt-get install -y cowsay  
4 CMD ["/usr/games/cowsay", "Hello World"]
```

apt-get install cowsay

apt-get update

ubuntu

```
>> cat Dockerfile

1 FROM ubuntu
2 RUN apt-get update
3 RUN apt-get install -y cowsay
4 RUN rm -rf /var/lib/apt/lists/*
5 CMD ["/usr/games/cowsay", "Hello World"]
```

```
rm -rf /var/...
```

```
apt-get install cowsay
```

```
apt-get update
```

```
ubuntu
```

```
>> cat Dockerfile

1 FROM ubuntu
2 RUN apt-get update && \
3     apt-get install -y cowsay && \
4     rm -rf /var/lib/apt/lists/*
5 CMD ["/usr/games/cowsay", "Hello World"]
```

update && install && rm

ubuntu

Question

Where did *ubuntu* come from?

Question

Where did *ubuntu* come from?

Answer

Our final definition — a *container registry*.

Definition 5. Container Registry

A file sharing platform that *hosts container images*. Container images are *pulled* (downloaded) from registries.

§ *Virtual Machines*

App 1	App 2
File System	File System
Guest OS	Guest OS
Hypervisor	
Operating System	
Hardware	

App 1	App 2
File System	File System
Guest OS	Guest OS
Hypervisor	
Operating System	
Hardware	

App 1	App 2	Docker Daemon
File System	File System	
Operating System		
Hardware		

Isolation

Virtual machines are used for *machine isolation*.

Containers are used for *process isolation*.

Size Comparison

I want *10* flask servers running on Ubuntu 22.

Ubuntu 22 $\simeq 3.8GB$

Python 3.6 $\simeq 232MB$

Flask $\simeq 11.1MB$

My App $\simeq 12K$

Virtual Machine

$$\begin{aligned}\text{Image Size} &= 3.8GB + 232MB \\ &\quad + 11.1MB + 12K \\ &= 4.04GB\end{aligned}$$

$$\begin{aligned}\text{Total Space} &= 4.04GB * 10 \\ &= 40.4GB\end{aligned}$$

Container

$$\begin{aligned}\text{Image Size} &= 12K \\ \text{Layer Size} &= 3.8GB + 232MB + 11.1MB \\ &= 4.04GB\end{aligned}$$

$$\begin{aligned}\text{Total Space} &= (12K * 10) + 4.04GB \\ &\simeq 4.04GB\end{aligned}$$

Question

When would I want a *virtual machine*?

Question

When would I want a *virtual machine*?

Answer

- Running a *different operating system*.
- *Unique hardware requirements* such as emulating old computer hardware.
- Where *security* is crucial virtual machines can offer greater isolation.

Question

When would I want a *container*?

Question

When would I want a *container*?

Answer

- Running a *single application*.
- *Lightweight* and *fast* to startup.
- Running *many containers* on the same system.

Combined Use Cases

Often virtual machines and containers are *combined*.

For example, if you deploy containers on Google Kubernetes Engine, the containers run inside of virtual machines on Google's hardware.

§ Use Cases

Dependency Management

Containers provide a reliable, if brute force, way to *manage dependencies*.

Wrap the whole machine state up and ship it.

Continuous Integration

Containers allow developers to locally *replicate the same test environment* as the CI system.

Continuous Delivery

Containers allow teams to package, deploy, and manage applications more efficiently.

Containers can be used to *deploy* on cloud platforms or on-premise servers with *minimal manual configuration*.

Scaling

Containers allow applications to be *scaled up or down quickly and efficiently*.

Microservices

Containers make it easy to deploy and manage
individual services independently.

Serverless

Containers are the basis for *serverless computing*

\S *Docker*



```
$ docker build [context]
```

Summary

Run each instruction in the *blueprint (Dockerfile)* to *build* each layer resulting in the top-level layer (*image*).

Key parameters

- f The Dockerfile to use (default: [context]/Dockerfile)
- t The tag (name) of the image to build

```
$ docker run [image]
```

Summary

Run a *container* from the specified *image*.

Key parameters

- d Run the container in the background
- p Publish a container's port to the host
- v Mount a volume
- e Set environment variables
- i Keep STDIN open even if not attached
- t Allocate a pseudo-TTY

```
$ docker exec [container]
```

Summary

Run a command in a *running container*.

Key parameters

- d Run the command in the background
- e Set environment variables
- i Keep STDIN open even if not attached
- t Allocate a pseudo-TTY

```
$ docker ps
```

Summary

List running containers.

Key parameters

- a Show all containers (default shows just running)
- f Filter output based on conditions provided

```
$ docker stop [container]
```

Summary

Stop a running container.

Key parameters

- t Seconds to wait for stop before killing it

```
$ docker rm [container]
```

Summary

Remove a container.

Key parameters

- f Force the removal of a running container (uses SIGKILL)
- v Remove the volumes associated with the container

```
$ docker images
```

Summary

List images.

Key parameters

- a Show all images (default hides intermediate images)
- f Filter output based on conditions provided

```
$ docker rmi [image]
```

Summary

Remove an image.

Key parameters

- f Force removal of the image

```
$ docker pull [image]
```

Summary

Pull an image or a repository from a registry.

```
$ docker push [image]
```

Summary

Push an image or a repository to a registry.

\S Examples

Structurizr

```
> git clone git@github.com:CSSE6400/software-architecture.git  
> cd software-architecture/slides/microkernel/c4_model  
> docker run -it --rm -p 8080:8080 -v $(pwd):/usr/local/structurizr  
    structurizr/lite
```

Open in browser: <http://localhost:8080>

GitLab

```
> mkdir gitlab
> export GITLAB_HOME=$(pwd)/gitlab
> docker run -it --rm -d -p 223:80 --shm-size 256m -v ${GITLAB_HOME}/
    config:/etc/gitlab -v ${GITLAB_HOME}/logs:/var/logs/gitlab -v ${
    GITLAB_HOME}/data:/var/opt/gitlab gitlab/gitlab-ee:latest
> cat ./gitlab/config/initial_password
```

Open in browser: <http://localhost:223>

Doom

```
> docker run -it --rm -p 224:6901 -e VNC_PW=password kasmweb/doom  
:1.12.0
```

Open in browser: <http://localhost:224>

Username: kasm_user

Password: password

§ Docker Compose

Exercise

We want to create *multiple* containers that
work together.

Exercise

We want to create *multiple* containers that
work together.

But we don't want to remember all the
commands to start and manage the
containers and get them to talk to each other...

When faced with tedium

Script it!

```
1 >> cat start.sh  
2  
3 docker build -t frontend ./frontend  
4 docker build -t backend ./backend  
5  
6 docker run -p 3000:3000 -v ./frontend:/app -e ... -d frontend  
7 docker run -p 8081:8081 -v ./backend:/app -e ... -d backend  
8 docker run -p 80:80 -v ./nginx.conf:/etc/nginx/nginx.conf -d nginx
```

This turns out to be very common. . .

This turns out to be very common...
Introducing... *Docker Compose*

```
>> cat start.sh  
1 docker build  
2     -t frontend  
3     ./frontend  
  
5 docker run  
6     -p 3000:3000  
7     -v ./frontend:/app  
8     -e ...  
9     -d  
10    frontend
```

```
>> cat docker-compose.yml
```

```
version: '3'  
services:  
    frontend:  
        build: ./frontend  
        ports:  
            - "3000:3000"  
        volumes:  
            - ./frontend:/app  
        environment:  
            - ...  
    backend:  
        build: ./backend  
        ports:  
            - "8081:8081"  
        volumes:
```

```
$ docker-compose up
```

Summary

Create and run containers.

Key parameters

- d Detached mode: Run containers in the background, print new container names
- build Rebuild containers if necessary.

```
$ docker-compose down
```

Summary

Stop and remove containers, networks, images, and volumes.

Key parameters

- v Remove named volumes declared in the ‘volumes’ section of the Compose file and anonymous volumes attached to containers.
- t Specify a shutdown timeout in seconds.

```
$ docker-compose ps
```

Summary

List containers.

```
$ docker-compose logs
```

Summary

View output from containers.

Key parameters

- f Follow log output.

```
$ docker-compose exec [service]
```

Summary

Run a command in a running container.

Key parameters

- d Detached mode: Run command in the background
- T Disable pseudo-tty allocation. By default ‘docker-compose exec’ allocates a TTY.
- e Set environment variables

```
$ docker-compose build
```

Summary

Build or rebuild services.

Key parameters

`-no-cache` Do not use cache when building the image.

`-pull` Always attempt to pull a newer version of the image.

In the practical this week...



MAN, DOCKER IS
BEING USED FOR
EVERYTHING.
I DON'T KNOW HOW
I FEEL ABOUT IT.
STORY TIME!



ONCE, LONG AGO,
I WANTED TO USE
AN OLD TABLET AS
A WALL DISPLAY.



I HAD AN APP AND A CALENDAR
WEBPAGE THAT I WANTED TO SHOW
SIDE BY SIDE, BUT THE OS DIDN'T
HAVE SPLIT-SCREEN SUPPORT.
SO I DECIDED TO BUILD MY OWN APP.



I DOWNLOADED THE SDK
AND THE IDE, REGISTERED
AS A DEVELOPER, AND
STARTED READING THE
LANGUAGE'S DOCS.



...THEN I REALIZED IT
WOULD BE WAY EASIER
TO GET TWO SMALLER
PHONES ON EBAY AND
GLUE THEM TOGETHER.



ON THAT DAY, I
ACHIEVED SOFTWARE
ENLIGHTENMENT.

BUT YOU NEVER LEARNED
TO WRITE SOFTWARE.

NO, I JUST LEARNED HOW
TO GLUE TOGETHER STUFF
THAT I DON'T UNDERSTAND.

I...OK, FAIR.

