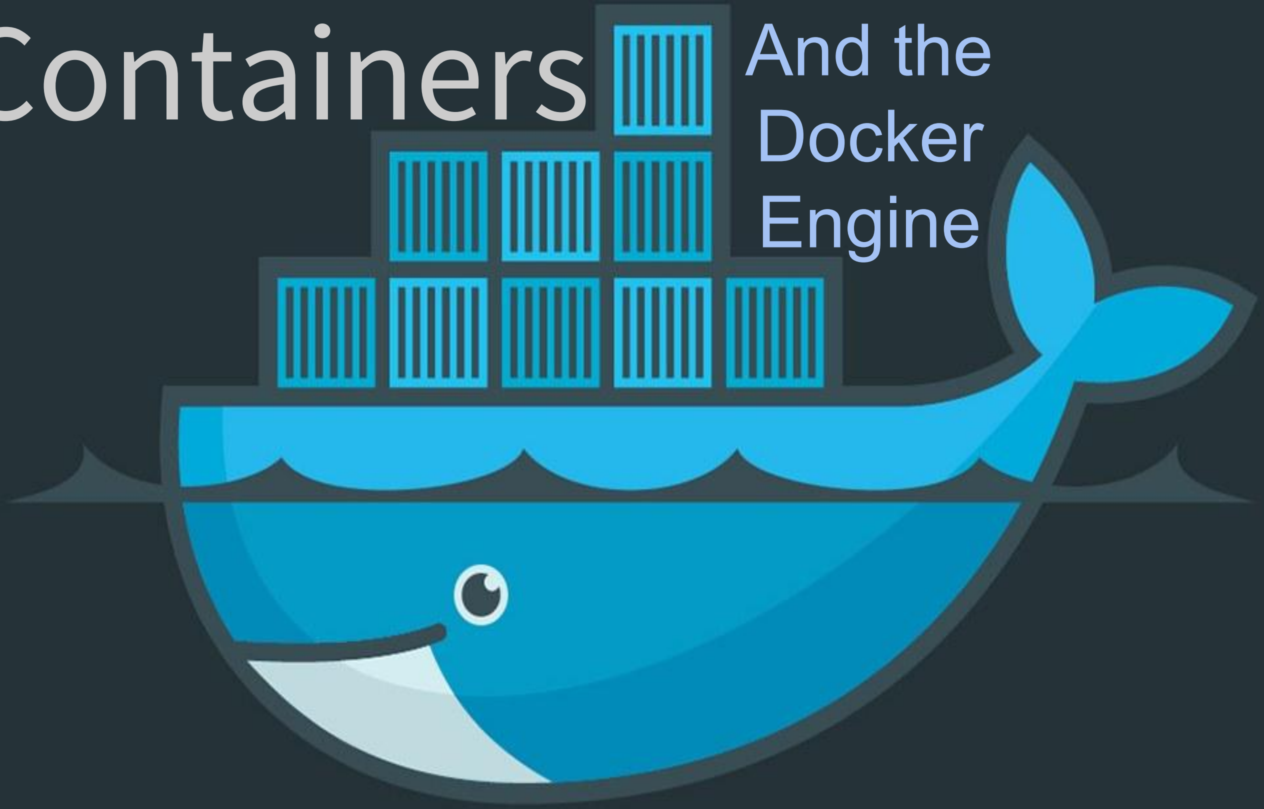
Docker Containers

And the Docker Engine



Michael DeFrancesco

wired.com

What is Docker?

What is a Container?

It is an isolation of a processe’s view of its operating environment. This is done through Linux namespaces.
Examples of resource names that can exist in multiple spaces, so that the named resources are partitioned, are process IDs, hostnames, user IDs, file names, and some names associated with network access, and interprocess communication.

And Docker?

Docker is an application-centric container runtime. It is ephemeral by nature and facilitates rapid deployment of services.

Docker grew in popularity by leveraging LXC to enable containers, or images, to share and run off of the host operating system’s kernel. As a result Docker images are extremely lightweight and run inside of their own ‘pid namespace’ on the host.
The leveraging of AuFS enables guest data to be accessed from the host without NAT’ing or bridging the Virtual system.

Because Docker Containers are so lightweight a host can easily scale from 100 to 1000’s of virtual operating systems without a significant slow down.

Implementation of Docker

Docker runs on top of the LXC api. LXC leverages Linux native PID namespaces and cgroups to be able to virtualize just the application environment.
This means that applications can run off of host container by sharing it's kernel and core processes. This dramatically reduces the hardware resources necessary to run an application or a service.

Filesystem

Docker implements a Union Filesystem in order to support a container's filesystem.
https://blog.docker.com/2015/10/docker-basics-webinar-qa/

Storage

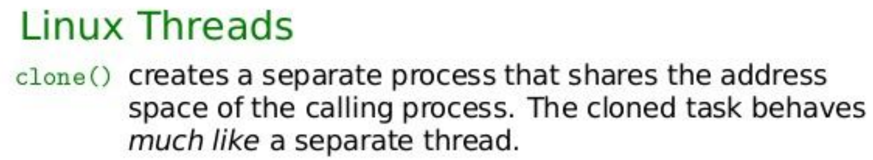
Docker Volumes enables containers to store persistent data outside of the container itself. This helps maintain the Docker Philosophy of 'single purpose ephemeral containers' while still satisfying the need Database and File System needs to facilitate web servers.

PID Namespaces?

A new PID namespace is created by calling clone() with the CLONE_NEWPID flag.

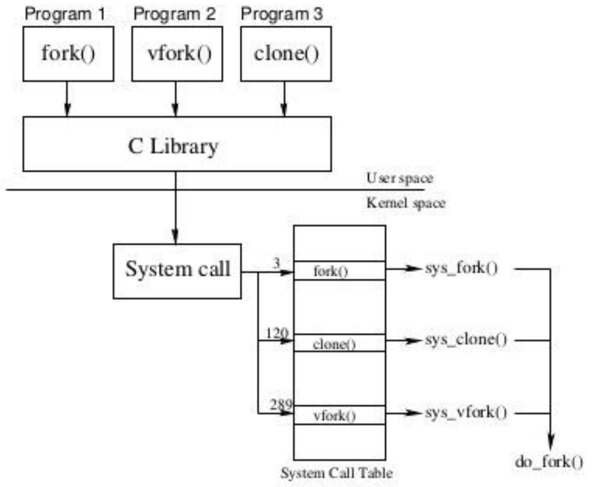
> Unlike fork(2), **clone**() allows the child process to share parts of its execution context with the calling process, such as the virtual address space, the table of file descriptors, and the table of signal Handlers. [Linux Manpage]

clone() allows the child and parent process to share memory, however, the child process cannot execute in the same stack as the parent process. The parent process must set up memory space for the child stack *inside* of its execution stack and pass a pointer to this space to **clone**() using the *child_stack* argument

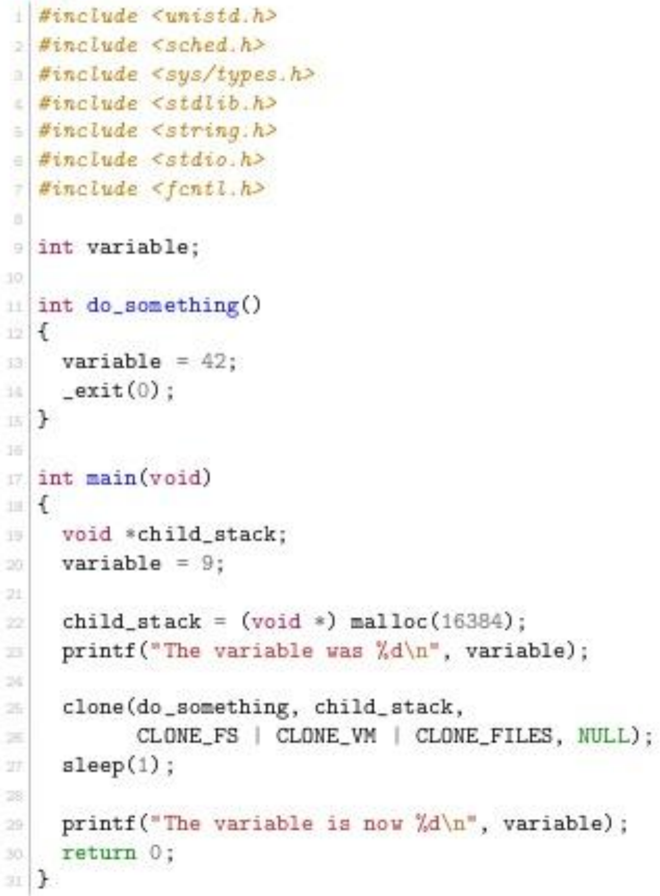


Linux Threads

`clone()` creates a separate process that shares the address space of the calling process. The cloned task behaves *much like* a separate thread.



clone() Example



```c
#include <unistd.h>
#include <sched.h>
#include <sys/types.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <fcntl.h>

int variable;

int do_something()
{
  variable = 42;
  _exit(0);
}

int main(void)
{
  void *child_stack;
  variable = 9;

  child_stack = (void *) malloc(16384);
  printf("The variable was %d\n", variable);

  clone(do_something, child_stack,
        CLONE_FS | CLONE_VM | CLONE_FILES, NULL);
  sleep(1);

  printf("The variable is now %d\n", variable);
  return 0;
}
```

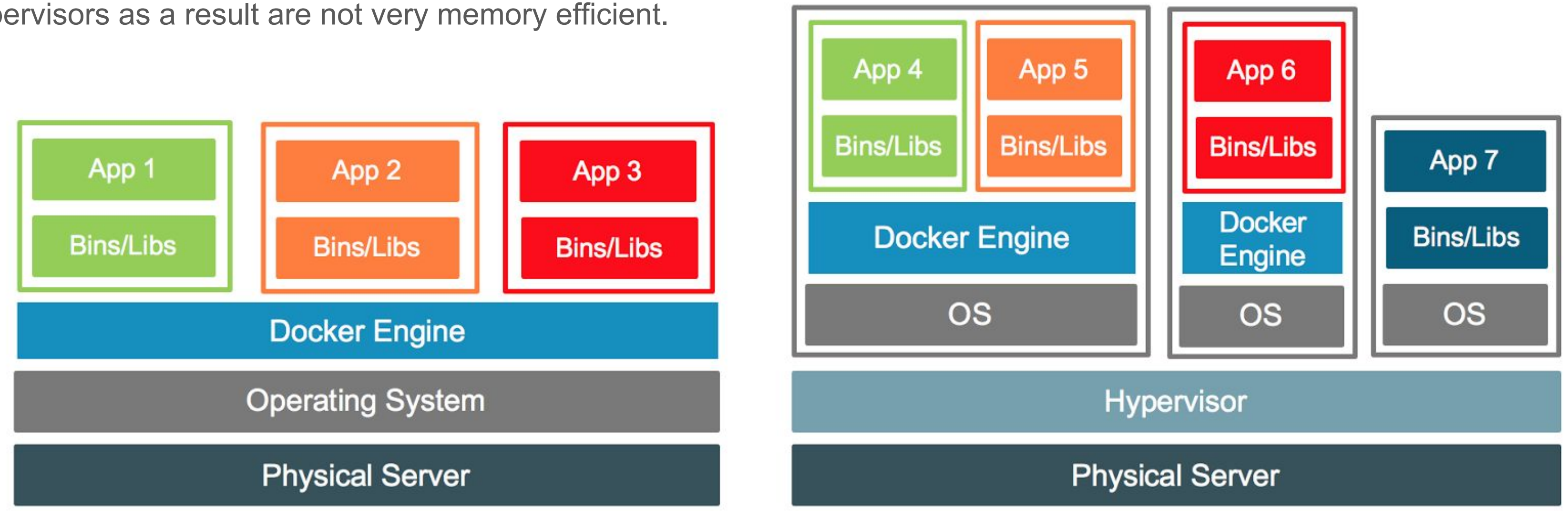
# The null pointer to child_stack is created

# **Clone()** is called and passed the function to execute and the pointer to memory location the function will occupy.

Life Before Docker

Before Docker, data centers used complex hypervisors such as ESXi, Xen, or Hyper-V in order to leverage virtualized resources.
Hypervisors introduced another layer of complexity by virtualizing hardware and drivers to support a guest kernel. If someone wanted to run a service inside of a virtual machine they would need sufficient hardware resources to run an abstraction layer, an entire operating system, its processes, and finally the service you intended to run.

Hypervisors as a result are not very memory efficient.



https://blog.docker.com/2016/04/containers-and-vms-together/

Advantages of Docker

Docker uses *containers*, which emulate only the environment needed to support applications and services.

Container hosted applications run on host machine by sharing it’s kernel and core processes. This dramatically reduces the hardware resources necessary to support an application or a service.

A host that was only capable of running a few 100 VM’s can now host 1000’s of Docker Containers.

Docker Machine

Docker took the SaaS market by storm as it enabled sysadmins and devops engineers very rapidly scale up their infrastructures and maintain a consistent development environment across several different platforms. In order to satisfy demand for Windows and MacOS users Docker Machine was created.

Implementation

Docker Machine functions differently than Native Docker on Linux. Docker Machine provides a set of cli tools to boot a virtual Alpine Linux image inside of a standard tier-2 hypervisor. The Docker Machine tools could then be used to execute Docker commands inside the VM from the host machines shell.

Reduced Performance

Docker Machine satisfied the need for portability to the Proprietary platforms however these users suffered from a performance disadvantage and a more complex implementation than the Linux Native Docker runtime.

As a result the Docker team sought out to develop Windows and Mac specific implementations of the Docker runtime in order to get more performance out of each of these platforms.

Implementation of Docker: Windows

Implementation

Because Docker is built so tightly around Linux native technologies Docker for Windows is currently functioning by running an Alpine Linux VM inside of Hyper-V. Support is planned to directly use Hyper-V containers in the future.

Storage

The Union File system is emulated when using Docker for Windows and instead uses virtual NTFS pools with several symlinks to the Windows host's system paths

Implementation of Docker: MacOS

Implementation

The MacOS implementation of Docker is very similar to the Windows Docker implementation. MacOS *also* relies on an Alpine Linux VM as a shim for the Docker Daemon.
However, MacOS uses the xhyve hypervisor to support the Alpine image. Xhyve has the advantage of letting the Alpine image talk directly to the Darwin kernel using hypervisor.Framework api calls. This nets a slight performance advantage to MacOS but does not achieve the performance of Linux native containers.

Storage

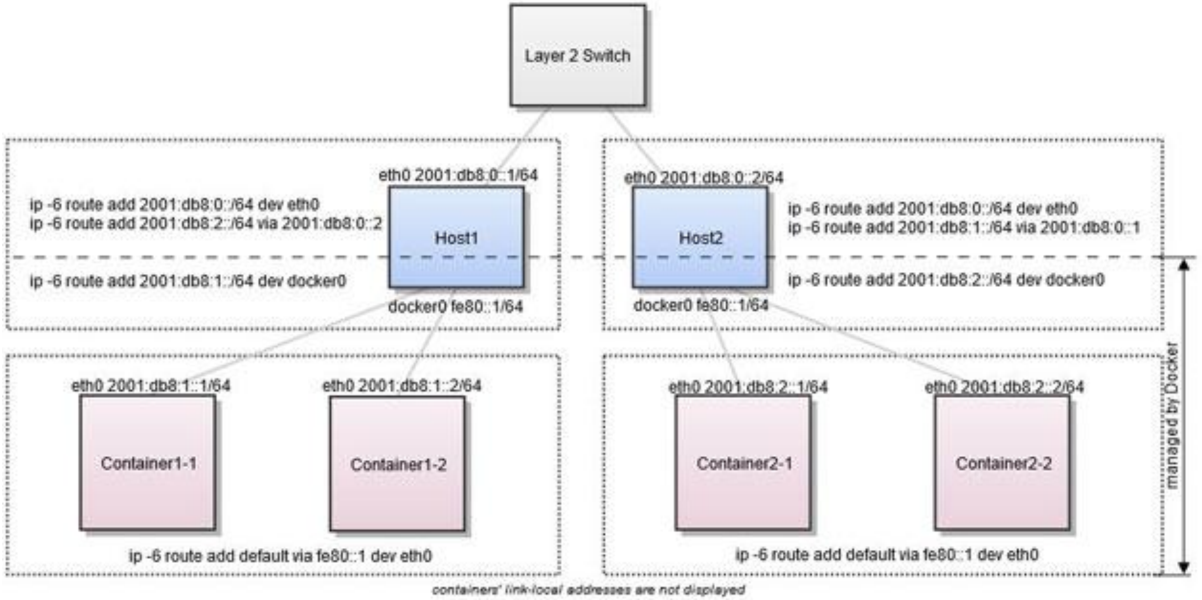
In order to support storage Docker for Mac utilizes osxfs shared volumes to facilitate filesystem writes from the container. This implementation offers very little I/O throughput (250MB/s 130μs latency) and is usually bundled with a filesystem cache.

Docker Networking

When the Docker daemon initializes a new container it is assigned a network address and connected to a virtual Ethernet bridge (docker0).

All containers communicate with each other through the docker0 bridge which automatically forwards IP packets through the docker subnet.

This is similar to how Distributed Computing networks are constructed.

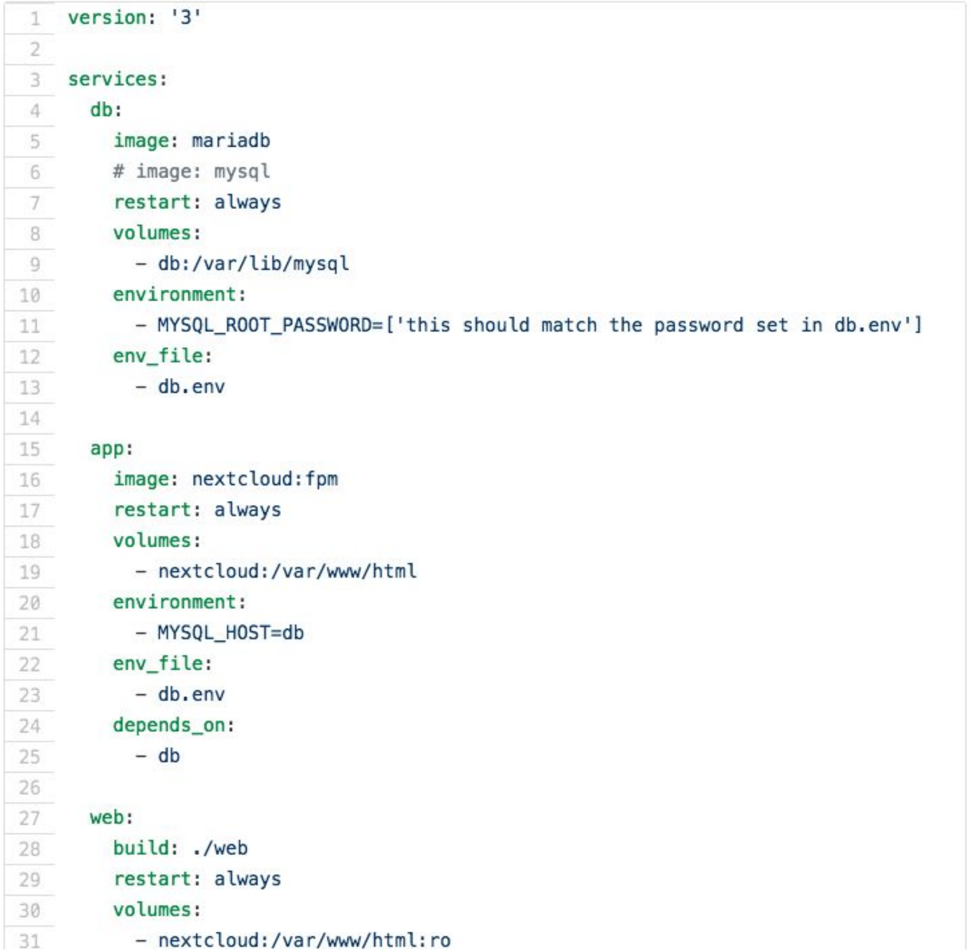

Layer 2 Switch
eth0 2001:db8:0::1/64
Host1
ip -6 route add 2001:db8:0::/64 dev eth0
ip -6 route add 2001:db8:2::/64 via 2001:db8:0::2
ip -6 route add 2001:db8:1::/64 dev docker0
docker0 fe80::1/64
eth0 2001:db8:0::2/64
Host2
ip -6 route add 2001:db8:0::/64 dev eth0
ip -6 route add 2001:db8:1::/64 via 2001:db8:0::1
ip -6 route add 2001:db8:2::/64 dev docker0
docker0 fe80::1/64
eth0 2001:db8:1::1/64
Container1-1
eth0 2001:db8:1::2/64
Container1-2
ip -6 route add default via fe80::1 dev eth0
eth0 2001:db8:2::1/64
Container2-1
eth0 2001:db8:2::2/64
Container2-2
ip -6 route add default via fe80::1 dev eth0
managed by Docker
containers' link-local addresses are not displayed

Docker Compose

Docker Compose was created in order to facilitate complex environments and network infrastructures. These environments require persistent services and data.

With Docker compose you can spin up multiple containers, bridges, and storage volumes using yaml directives. IP's, hostnames, container names and environment variables can all be declared with Docker compose. When a new container is spun up it will automatically migrate the Volumes from its predecessors.

Docker compose is also very useful for DevOps as entire development environments can be effortlessly migrated from Testing to QA, to Production.



```yaml
version: '3'

services:
  db:
    image: mariadb
    # image: mysql
    restart: always
    volumes:
      - db:/var/lib/mysql
    environment:
      - MYSQL_ROOT_PASSWORD=['this should match the password set in db.env']
    env_file:
      - db.env

  app:
    image: nextcloud:fpm
    restart: always
    volumes:
      - nextcloud:/var/www/html
    environment:
      - MYSQL_HOST=db
    env_file:
      - db.env
    depends_on:
      - db

  web:
    build: ./web
    restart: always
    volumes:
      - nextcloud:/var/www/html:ro
```

Docker Swarm

Docker facilitates inter-container networking by functioning as a virtual subnet. This means that Docker is also a *Software Defined Network*. More Info
Abstracting Hardware Networking away from Docker enables container networks to be easily migrated between or to span between multiple platforms. This also opens up new possibilities such as Docker Swarm.

What is Docker Swarm?
Docker Swarm is a platform that allows Docker to be used in cluster computing environments. Docker Swarm is decentralized and lets containers float across multiple hosts. Docker Swarm based off of the low level Docker Orchestration features implemented in Docker Compose.

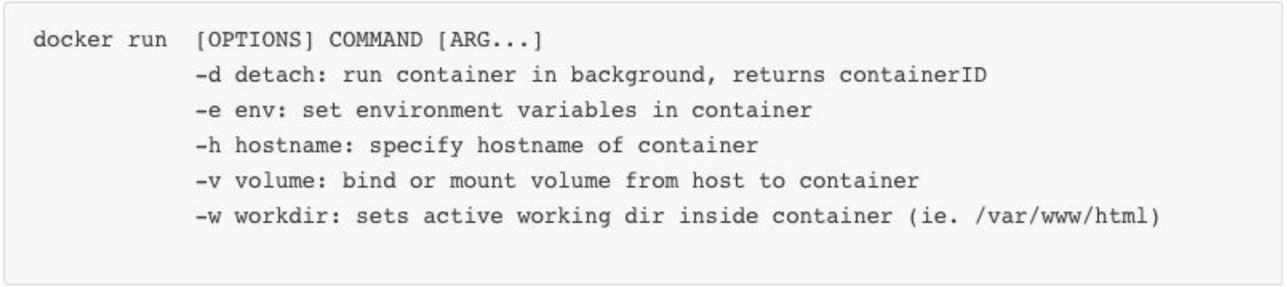
Why Not Just Use Compose?
Docker Swarm lets users modify container configuration, volumes and exposed ports *without restarting the containers*. This capability is imperative for clustered networks where there needs to be a consistent uptime. Additionally, Docker Swarm managers are the *only* processes that can *modify* the Swarm and its components. Standard Docker processes can participate but cannot modify the Swarm. This extra layer of isolation provides a security advantage in environments where users can spin up containers without being a threat to the overall system (such as when a user rents a VPS from a hosting company). More Info

Using Docker:

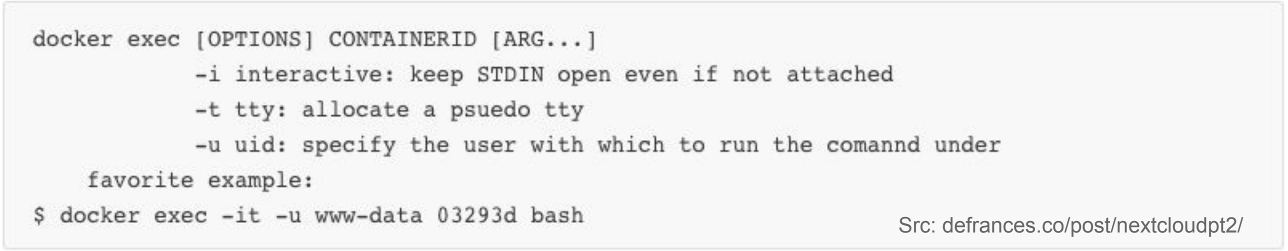
Docker Run
By default, Docker pulls from the DockerHub registry. This means if you attempt to use a container image that does not exist locally Docker will download it from DockerHub onto your machine.

Docker uses a very structured command line syntax

Docker Exec
Docker exec runs arbitrary commands from within the container. This enables scripting of containers from the host and maintenance of the containers from the host.



```
docker run  [OPTIONS] COMMAND [ARG...]
            -d detach: run container in background, returns containerID
            -e env: set environment variables in container
            -h hostname: specify hostname of container
            -v volume: bind or mount volume from host to container
            -w workdir: sets active working dir inside container (ie. /var/www/html)
```



```
docker exec [OPTIONS] CONTAINERID [ARG...]
            -i interactive: keep STDIN open even if not attached
            -t tty: allocate a psuedo tty
            -u uid: specify the user with which to run the comannd under
    favorite example:
$ docker exec -it -u www-data 03293d bash
```

Src: defrances.co/post/nextcloudpt2/

Where does Docker fall short?

Docker satisfies a lot of niche needs in the industry. It makes the Software Development Lifecycle dramatically easier. Docker makes services portable across multiple platforms so it is no surprise that it has been extremely well received.

Steep Learning Curve
Not only does Docker itself have a lot of depth, and thus, a decent learning curve but it requires a deep understanding of the Linux/Unix architecture in order to be self sufficient working with Docker Hosts.

Performance hits on Proprietary Platforms
Linux has and still is the premiere platform for the Docker Engine. Because Docker relies on Linux at such a low level users on MacOs or Windows platforms will not get the same raw performance as Linux users. Docker as a service is more suited for Linux hosts, where as development testing environments should be fine on the ported platforms.

You are Stuck With the Host's Kernel
Because the Docker Engine itself, and the containers it runs, all exist as their own processes users are stuck with the Host's containers. Therefore, User's cannot expect Docker to completely replace standard Hypervisors. Docker is not and will never be the answer to gaming on Linux.

Conclusion

Get
into
DevOps.

Docker has revolutionized the server and DevOps industry with its surging popularity and portability.

When LXC was created it was viewed as a promising API. At first it only seemed like it would serve as an extension of hypervisors.

However, Docker is responsible for proving Linux Containers can be used for rapid deployments and leveraged Linux’s little known but unique advantages to truly revolutionize the SaaS industry.