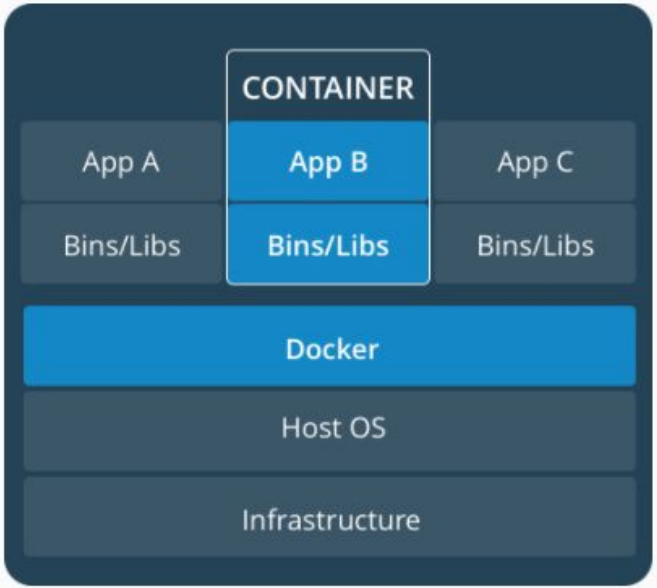
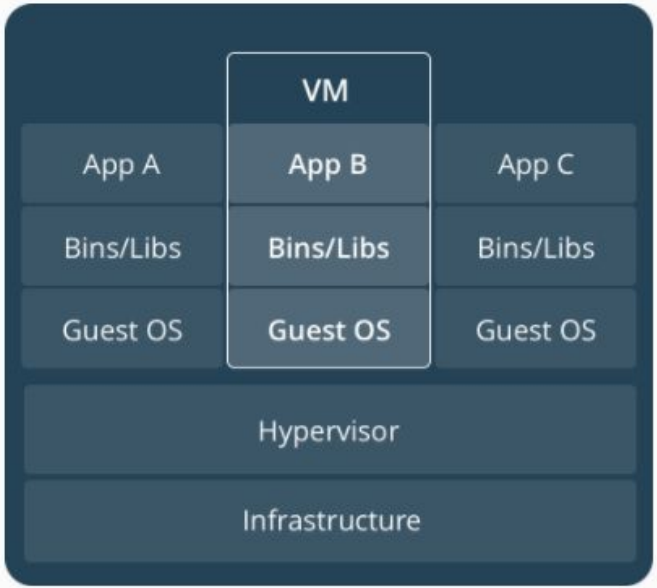
Containers & Orchestration

Taking layers of abstraction to a whole new level


IT WORKS ON MY MACHINE
THEN WE'LL SHIP YOUR MACHINE
AND THAT IS HOW DOCKER WAS BORN





CONTAINERS

Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), and start almost instantly.

VIRTUAL MACHINES

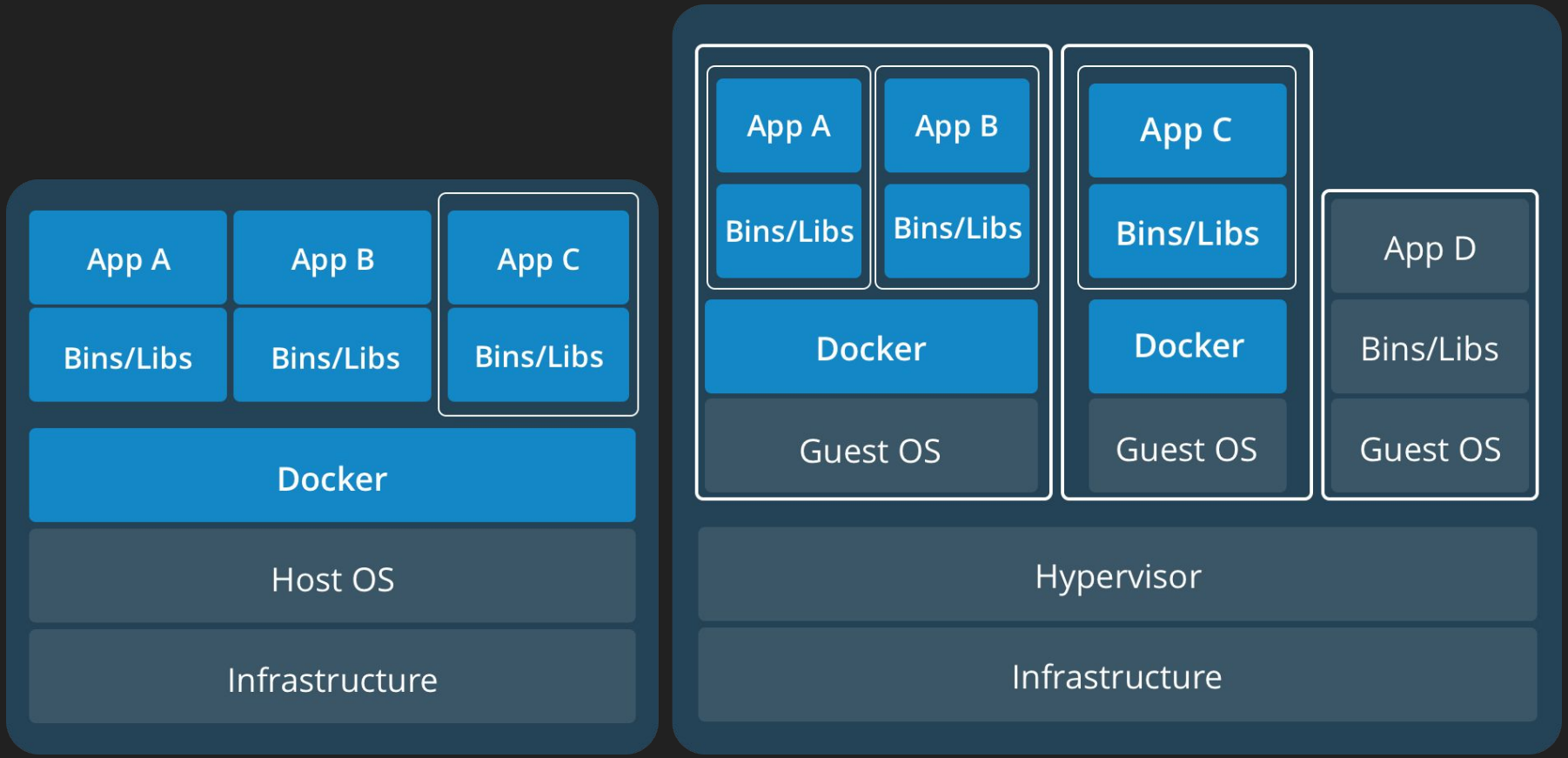
Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, one or more apps, necessary binaries and libraries - taking up tens of GBs. VMs can also be slow to boot.

Containers

What's the Diff: VMs vs Containers

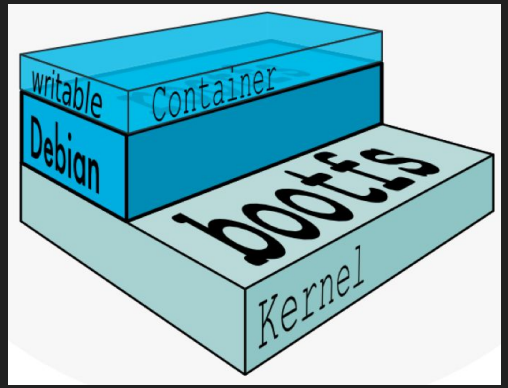
VMs	Containers
Heavyweight	Lightweight
Limited performance	Native performance
Each VM runs in its own OS	All containers share the host OS
Hardware-level virtualization	OS virtualization
Startup time in minutes	Startup time in milliseconds
Allocates required memory	Requires less memory space
Fully isolated and hence more secure	Process-level isolation, possibly less secure

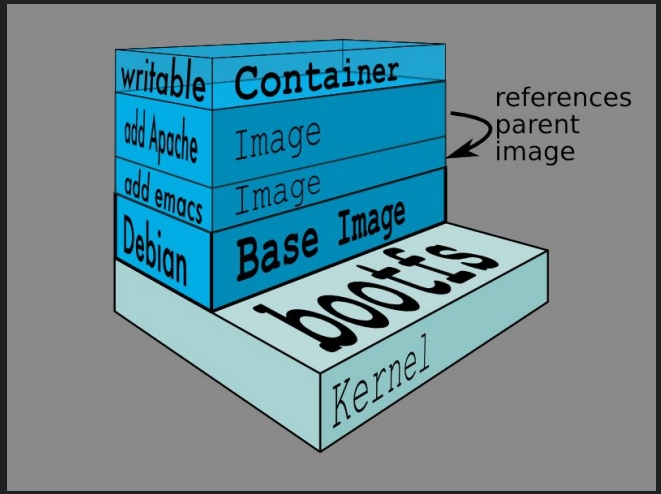
Containers in DEV vs in PROD



Container Images

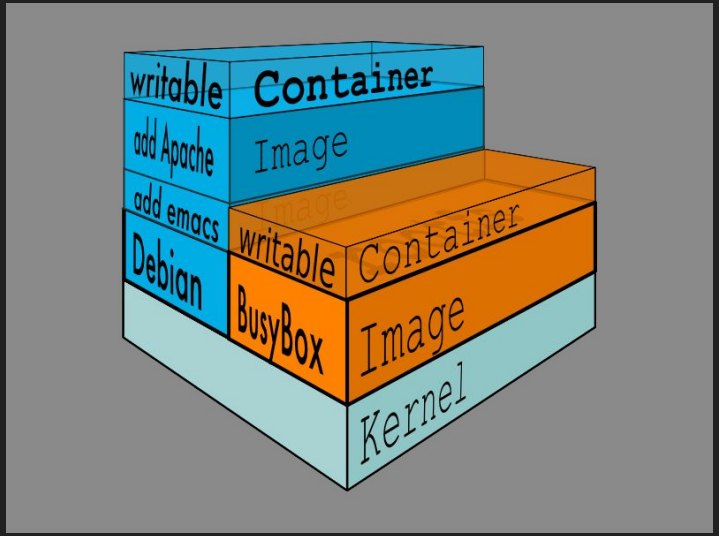
- NOT A VHD
- NOT A FILESYSTEM
- Comprised of read-only layers
- Do not have state
- Basically a tar file
- Has a hierarchy
- Stored in the Docker Registry





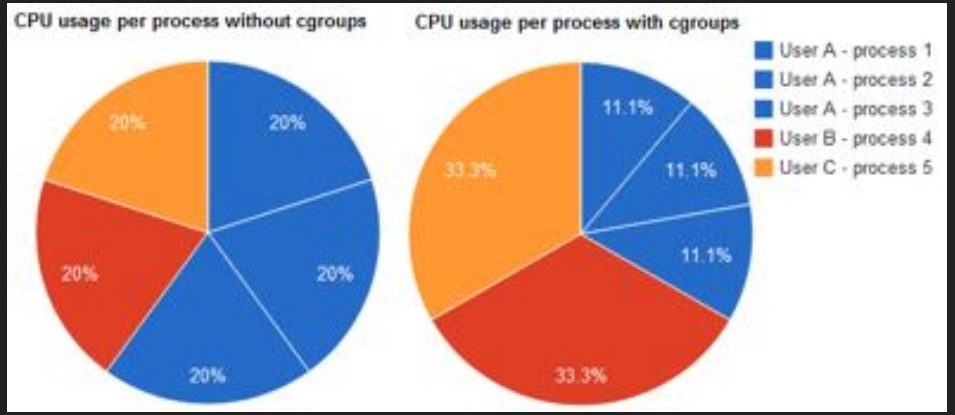
Containers

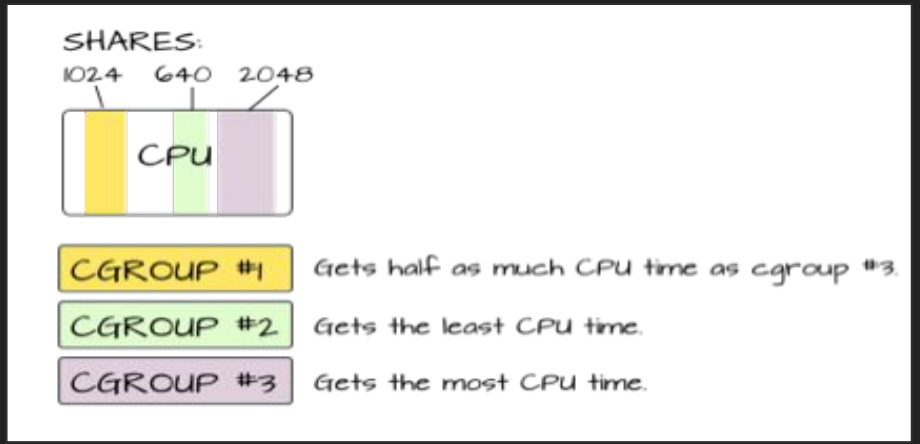
- Deployment Artifacts
  - What actually runs in Dev, Stage, QA, Prod
- Can run everywhere
  - Regardless of kernel version
  - Regardless of host distro
  - (Container and host architecture must match*)
- Can run anything *
  - If it can run on the host, it can run in the container
  - i.e., If it can run on a Linux kernel, it can run



CGroups - revisited

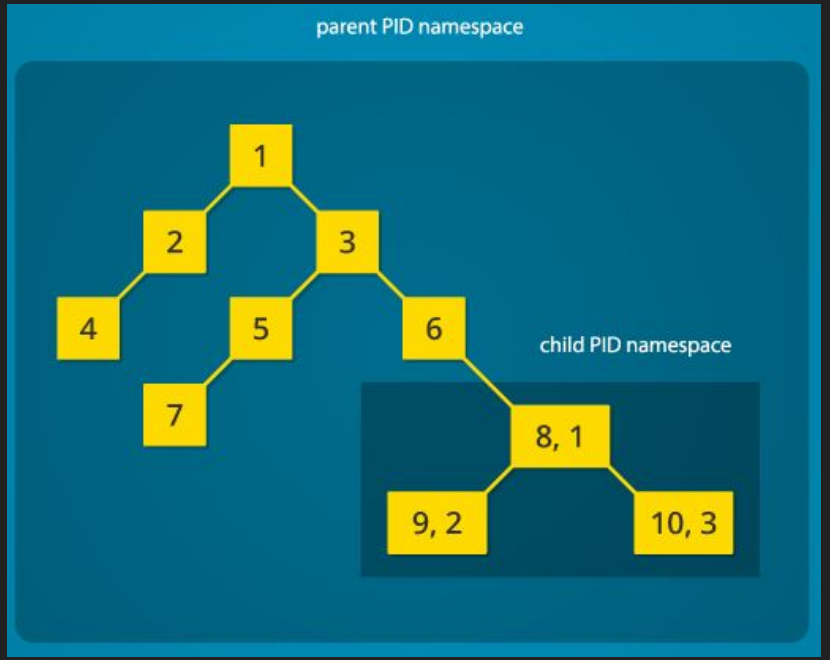
- CGroups are a Kernel Feature
- Groups of processes
- Control resource allocations
  - CPU
  - Memory
  - Disk
  - I/O
- Can be nested within each other





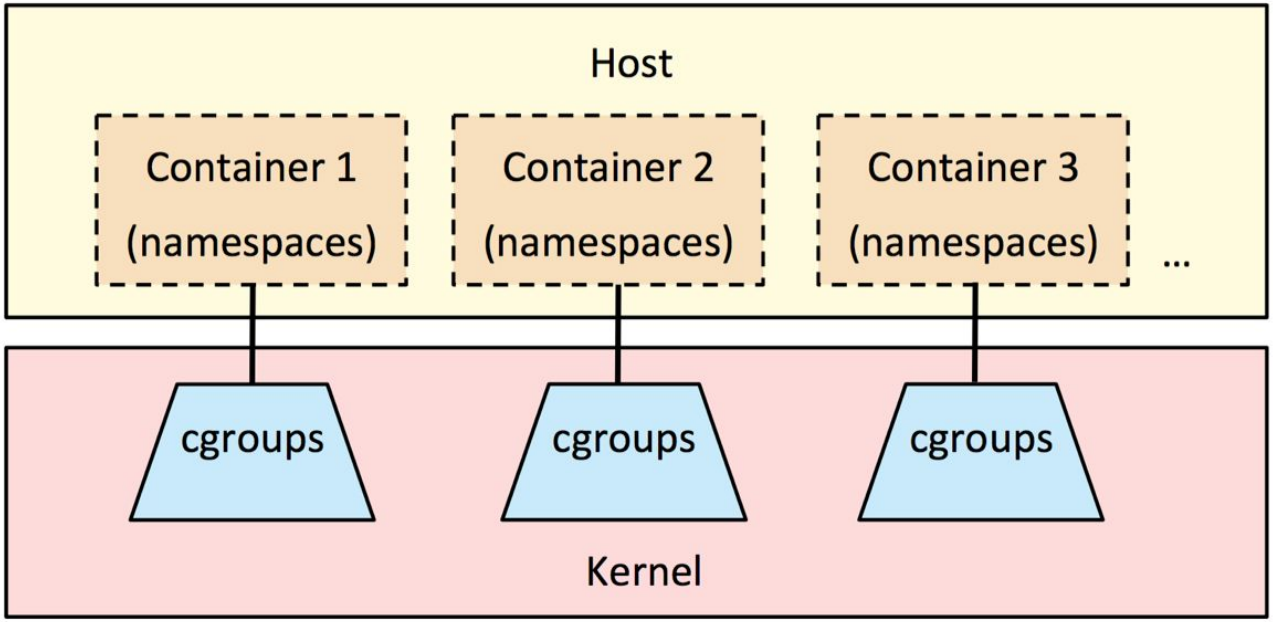
Linux Namespaces

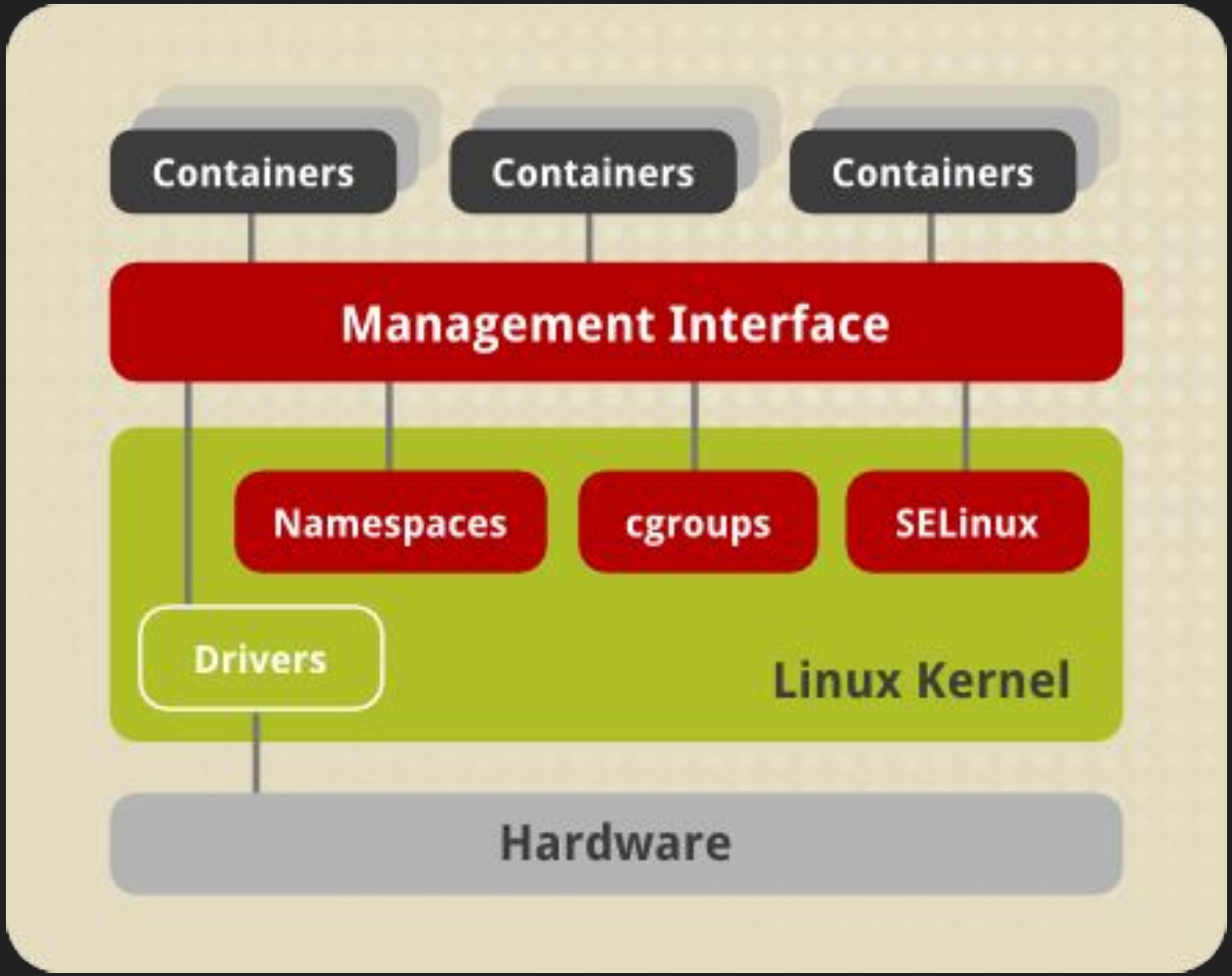
- Kernel Feature
- Restrict a process view of the system
  - Mounts (CLONE_NEWNS)
  - UTS (CLONE_NEWUTS)
  - IPC (CLONE_NEWIPC)
  - PID (CLONE_NEWPID)
  - Networks (CLONE_NEWNET)
  - User (CLONE_NEWUSER)
    - Has privileged/unprivileged modes
- May be nested



Linux Containers

Container = combination of namespaces & cgroups




Containers
Containers
Containers
Management Interface
Namespaces
cgroups
SELinux
Drivers
Linux Kernel
Hardware

That’s probably enough theory - time to use it

Running your first container

- To spin up your first container, simply run the following

  $ docker run busybox echo hello world

This uses one of the smallest, simplest images available: busybox.

- Busybox is typically used in embedded systems (phones, routers...)
- We ran a single process within the container and echo'd hello world.

How about something a bit more useful?

- To spin up something a bit more useful for development, etc try running an Ubuntu container

    $ docker run -it ubuntu /bin/bash

This is a brand new container which runs a very runs a bare-bones, no-frills ubuntu system.

When I say bare bones - I mean really stripped down.

    $ dpkg -l | wc -l
    94

There are only 94 base packages installed on the container - compared to the usual 521 of a base install

So let’s run something

For instance, let’s curl google

$ curl -L www.google.com
bash: curl: command not found

Yeah...it doesn’t have curl by default

So let’s install it

$ apt update && apt install curl

Now we should be good to go

$ curl -L www.google.com

Exiting the container

To exit a container, it's the same as quitting your shell in terminal (as in the previous example, we were running a shell)

$ exit

At this point, our container is now in a stopped state.
It still exists on disk, but all compute resources have been freed up

So what if we need to spin it up again and re-run our curl?

$ docker run -it ubuntu /bin/bash
$ curl -L www.google.com
bash: curl: command not found

What the heck?

Exiting/Stopping the container

Turns out, when we use *docker run* it creates a new container off of the image which doesn't have any of our changes. So, how can we get back in?

When we existed our container, it remained on disk - just in the stopped state. So all we need to do is spin it back up and re-attach to it.

$ docker start -ia <container-id>

$ docker start -ia d8360119ba8d

A bit later in the lecture we will cover how to persist changes by creating new images to run from

Image Registry and DockerHub

So at this point we have spun up a few containers, but where did they come from?

Docker Images are stored in what is known as a registry - by default the public DockerHub is used.

When specifying an image there are three levels of the hierarchy

- Root-like
  - ubuntu
  - busybox
  - mysql
- User/Organization
  - daviddob/CI-CD
- Self-Hosted
  - registry.example.com/private-image

Image Registry and DockerHub

If you try to run an image you do not have locally, docker will automatically search the registry for an image and automatically pull it down

If you want to search for an image manually you can use

$ docker search <query>

And to pull it down manually (without running it) you can use

$ docker pull <image-name>

And to see what images you already have on you local instance you can use

$ docker images

Image Registry and DockerHub

Lastly, let’s talk about tags

Each docker image has tags associated with it that define variants of the same image

- Examples of tags could be
  - httpd:latest
  - httpd:alpine
  - httpd:2
  - httpd:2.4.39-alpine

The latest tag is usually just that - the latest build. Usually when running in prod you want to be a specific as possible - don’t use latest. Otherwise it may update without you realizing and break your deployment.

Background/Daemon Containers

So up until now, our containers have been interactive - we have been using them pretty much as a shell - however most containers running in deployments are non-interactive daemons

So how do we spin up a non-interactive container? Here is an example if a simple clock container

$ docker run jpetazzo/clock

This will print out the time every second - however its running in the foreground and we can still interact with it (i.e using ^C to kill it)

$ docker run -d jpetazzo/clock

The -d flag has it spin up in daemon mode - no output to the terminal - background process

Background/Daemon Containers

So if we have our container running as a daemon, how do we get its output and check the logs?

Thankfully docker handles storing the logs for us and presenting them in a fairly clean way, however to get the logs we need the ID of the container

To see running containers and their IDs we can use

$ docker ps

And to see stopped containers and their size on disk you can specify the *-as* flags

$ docker ps -as

Container Logs

Now that we have the ID of our running container, we can check the logs

To do this we use

$ docker logs <container-ID>

This will output the entire log history for the container - this can be a TON of output so we need some way to truncate it

For that, we can use the built in tail feature for docker logs

$ docker logs --tail 3 <container-ID> - Will show last 3 lines of the log
$ docker logs --tail 1 --follow <container-ID> - Will show the last line of the log and then continue to print out new lines to the console

Stopping Daemon Containers

Now that we have our daemon running ang logging properly, what if we want to stop it?

Similar to processes in linux, we can stop or kill containers

$ docker stop <container-ID>

Stop is more of a graceful way to stop a container - it will send it the TERM signal and give it 10 seconds to stop gracefully before following up with a KILL signal

$ docker kill <container-ID>

Kill is pretty much what you would expect, the process in the container gets sent a KILL signal which cannot be intercepted and the container is forcefully exited

Stopping Daemon Containers

Now that we have our daemon running ang logging properly, what if we want to stop it?

Similar to processes in linux, we can stop or kill containers

$ docker stop <container-ID>

Stop is more of a graceful way to stop a container - it will send it the TERM signal and give it 10 seconds to stop gracefully before following up with a KILL signal

$ docker kill <container-ID>

Kill is pretty much what you would expect, the process in the container gets sent a KILL signal which cannot be intercepted and the container is forcefully exited

Persisting Changes to Images

At this point we can spin up containers from images, start and stop them, and track their logs. But what if we need something to persist for future containers?

There are two main approaches:

- Write the changes into a Dockerfile and build a new image
- Make the changes interactively and *commit* them to a new image

Making changes via a Dockerfile is usually the recommended way to do this, as it allows for rebuilding the image on demand even if the entire image is lost, however we will go over both methods.

Interactive Changes to Containers

First, let’s look at updating containers interactively:

Start with running a base image:

$ docker run -it ubuntu /bin/bash

Then make any changes you require interactively as you would on a traditional server

$ apt-get update && apt-get install -y curl wget htop

Now that we have performed any modification we need, it’s time to check our changes and commit them to a new image

Interactive Changes to Containers

To check what we changed in a given container we can use the *docker diff* command

$ docker diff <container-id>

This will print out all of the filesystem changes that were made (Create, Modify, Delete, etc) similar to git but for the container

If the changes are what you expect them to be, they can be committed to a new image via

$ docker commit <container-id>

Which will output the new container ID that was created

Interactive Changes to Containers

Once we have our new image created, it is often helpful to tag an image for ease of use or to make it publically available via a registry

$ docker tag <newImageId> myuser/mydistro:mytag

After tagging, the image can be pushed to the registry

$ docker push myuser/mydistro:mytag

It can then be referenced locally by that name for other docker commands

$ docker run -it myuser/mydistro /bin/bash

Building via Dockerfile

While building new images interactively can be easy, it has the downside of not being easily rebuilt from scratch. This means updates to the base image cannot easily be rolled out.

This is where Dockerfiles come in. They can be used to build and customize a container from scratch and have some powerful commands.

Here is an example of a simple Dockerfile

```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y curl htop
```

Building via Dockerfile

To build our new image we can use the *docker build* command

$ docker build -t myuser/mydistro:mytag .

-t specifies the tag to assign to the newly created image
. is the directory for context when building the new image

The build context is the directory where the Dockerfile is stored as well as the base for adding files/etc to the image

A full list of Dockerfile directives can be found here - https://docs.docker.com/engine/reference/builder/

Explain example go-webapp Dockerfile

Docker Networking

We aren’t going to dive too deep into container networking - just the basics of allowing traffic from a host to enter a container.

The first method is the *EXPOSE* directive in a dockerfile. It is specified similar to below

```
EXPOSE 80
EXPOSE 1194/udp
```

Be default, EXPOSE will assume you intend for TCP traffic - if you wish to expose both tcp and udp you must specify both in the dockerfile

Any ports that are exposed will be assigned random ephemeral ports on the host side when published during *docker run*

Docker Networking

To publish exposed ports when running a container you must specify the *-P* flag

$ docker run -P -d httpd

This will map all exposed ports to random high order ports. If this is not desired behavior (i.e. you want the port to be static) then you can specify port mappings manually on container start whether or not the port is exposed in the dockerfile.

$ docker run -d -p 80:80 -p 1194:1194/udp myuser/myapp:latest

There is so much more you can do with the docker engine, however this is a good introduction to the basics of building and running containers.

Questions?