



# DOCKER FUNDAMENTALS



# CONTAINERS AND DOCKER



# CONTAINERS



# CONTAINERS — WHAT ARE THEY?

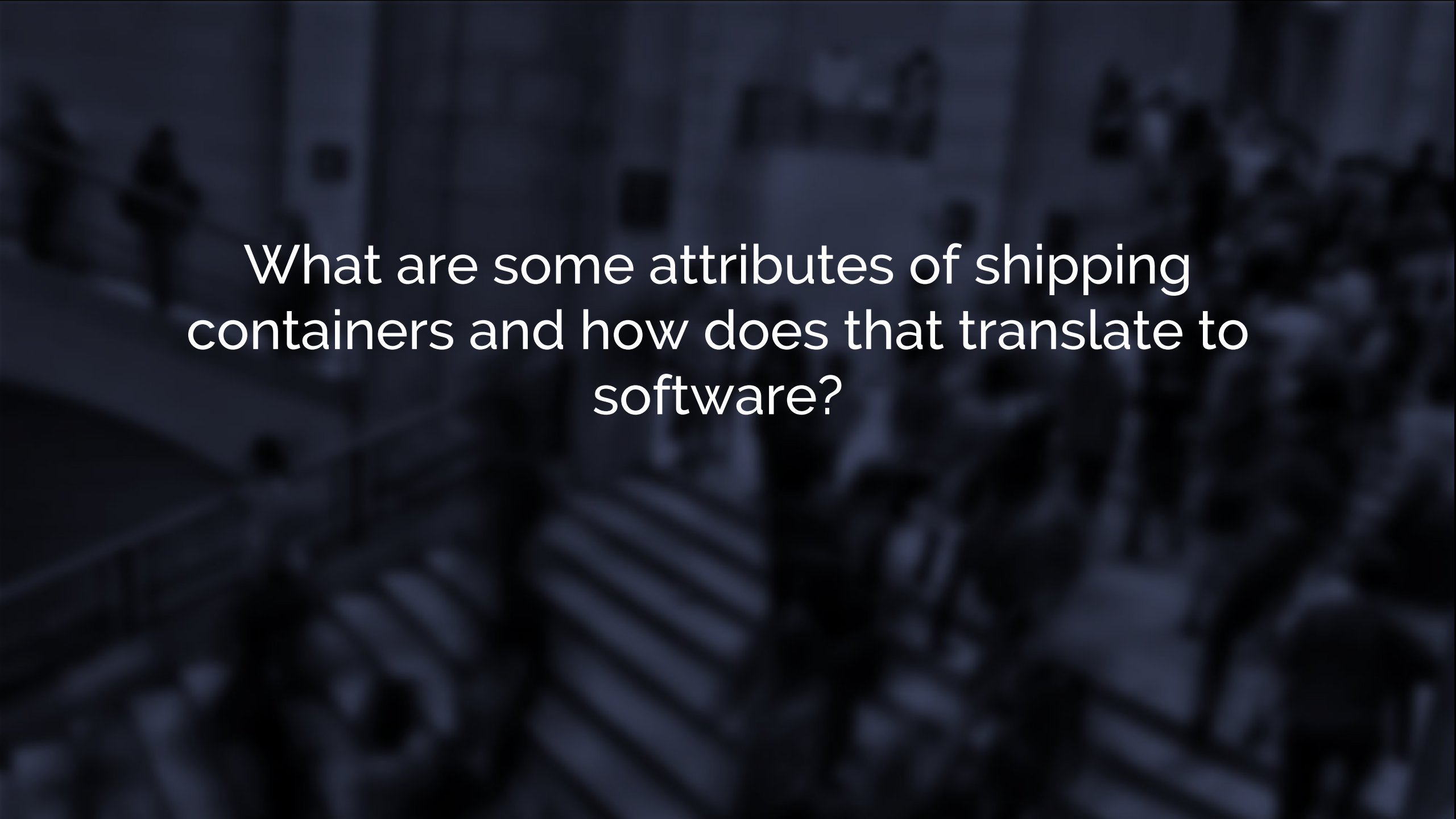
- Containers are portable, entirely self contained machines that run on a host OS, whether virtual or bare metal.
- Containers run the same everywhere and can be run on different flavors of OS in the same way
- They can be clustered together on machines of different sizes.



# CONTAINER ANALOGY



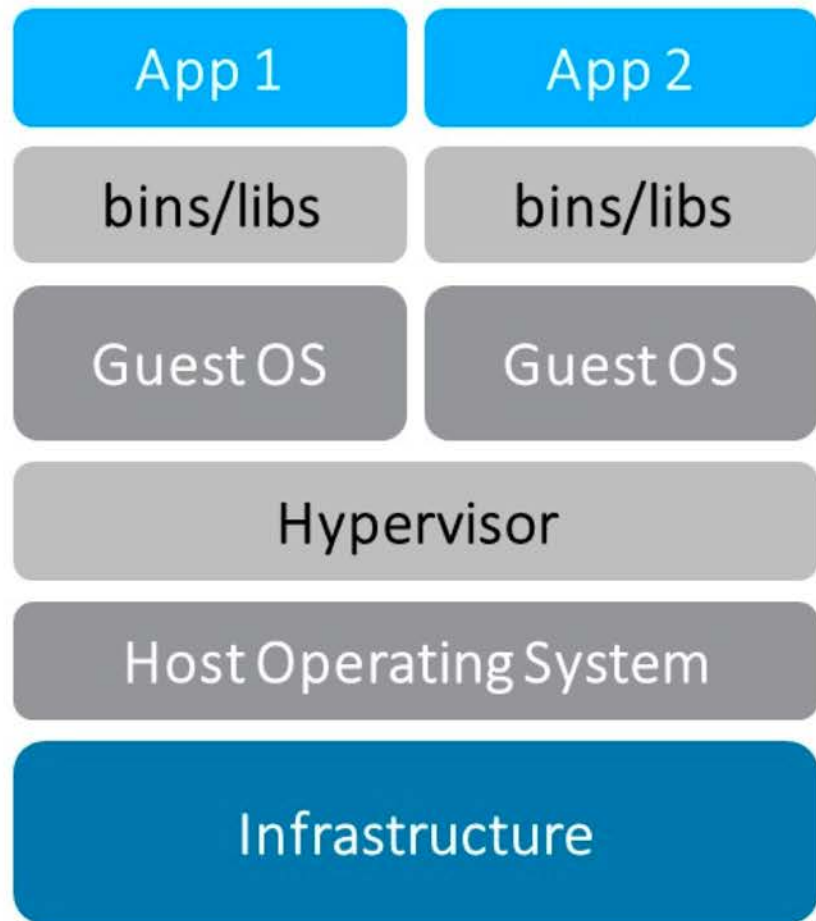




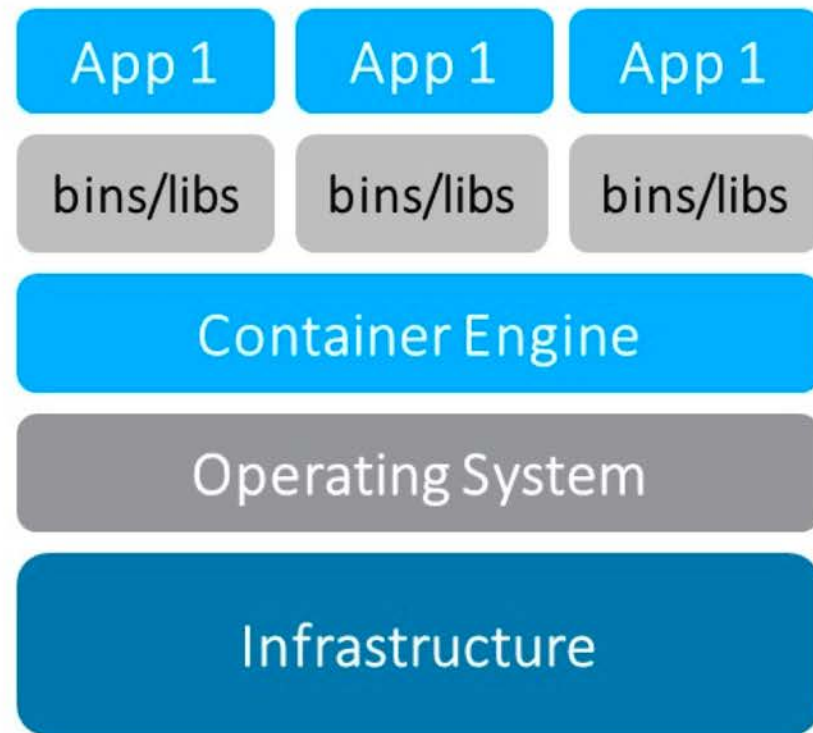
What are some attributes of shipping containers and how does that translate to software?

# CONTAINERS — LIKE A VM, BUT NOT A VM

- Containers are nothing more than a process that is isolated from other processes.
- They masquerade as virtual machines and act very much like them, but are nothing more than programs.
- Linux cgroups and namespaces provide the isolation.



**Virtual Machines**



**Containers**

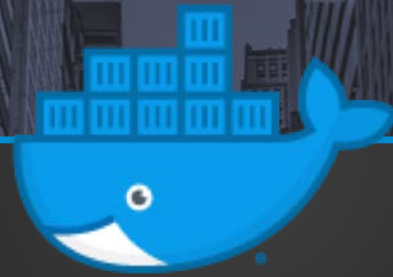


# CONTAINERS – WHAT ARE THEY USED FOR?

Containers have a lot of uses!

- Web applications running in clusters (UI, REST services, gateways, databases, queue services, etc.)
- Supporting services (logging, monitoring, alerting, etc.)
- Batch processing (containers get created and process immediately)
- Developer utilities (CLIs, software installs)
- DevOps utilities (build tools like Jenkins)

# DOCKER AND THE



# CONTAINER ECOSYSTEM



# ECOSYSTEM – OVERVIEW

The Docker container ecosystem is made up of many parts.

- containerd – the container runtime for Docker. This is what makes a container run the same everywhere from your laptop to a production server.
- Docker Engine – manages the lifecycle of containers on a host.
- Docker Community Edition – the open source tools you'll use on your workstation. It is meant for developers.
- Docker Enterprise Edition – the commercial offerings by Docker.
- Swarm – orchestration of Docker containers.
- Kubernetes – production grade container orchestration. Recently integrated with Docker.

# DOCKER





# DOCKER – FUNDAMENTAL CONCEPTS

- Docker isn't the only container runtime. But it is the one that changed the game by bringing a vastly superior ecosystem to make it user friendly to use containers.
- Containers do one job and one job only, with few exceptions
- Containers run the same everywhere and can be run on different flavors of OS in the same way
- Containers are intended to be zero overhead.
- Containers are ideal for creating a pool of machines that can host a mix of different app containers.

# DOCKER – IMAGES AND LAYERS

I'm a web server image made up of:

Linux OS: Ubuntu	Layer 1 - 800mb
Security patches and fixes	Layer 2 - 3mb
Nginx web server software	Layer 3 - 100 mb
Nginx plugins	Layer 4 - 50 mb
My custom web site files	Layer 5 - 15 mb

Key concepts:

- Images are inherited from base images and can be many levels deep
- Image is the definition of what a container is created from
- Images are immutable. If you make changes, a new image must be built.
- Images are made of layers
- When pushing/pulling an image to a repository (more to come), only the changed layers are pushed to save bandwidth.



# DOCKER — IMAGE INHERITANCE

- All images are based on other images as their parent.
- You must choose a parent, or FROM image.
- Scratch is the base empty image supplied by Docker.
- Base images include all settings, files, what command runs at startup, etc.

# DOCKER — IMAGE NAMES AND TAGS

- Docker images are given a unique ID by default, but you can make them easier to use with names and tags.
- The name is the unique identifier for a class of images
- Every image gets a tag which is like the version of that image name.
- Image and tag are separated by a colon.
  - For example, myimagename:mytag
- There is a reserved tag name of `latest`
  - If you omit the tag, then `latest` is assumed
  - For example, `docker pull nginx` is equivalent to `docker pull nginx:latest` from Docker Hub



# DOCKER – CONTAINERS

I'm a web server container

Ports

Env

Vol

I'm a web server container

Ports

Environment variables

Volumes

I'm a web server container

Ports

Environment variables

Volumes

I'm a web server container

Ports

Environment variables

Volumes

# DOCKER — PROCESS LIFECYCLE

- All containers have a single ENTRYPOINT, or process that starts when the container starts.
- The lifecycle of the container is tied to this process. If the process ends, then the container ends.
- We can run additional processes against a container after it is started, but they do not affect the container lifecycle.



# DOCKER – REGISTRIES/REPOSITORIES



mygninx:1.0

I'm a web server image made up of:

Linux OS: Ubuntu	Layer 1
Security patches and fixes	Layer 2
Nginx web server software	Layer 3
Nginx plugins	Layer 4
My custom web site files	Layer 5



Registry  
mygninx:1.0  
mynginx:2.0  
myapp:v1  
myapp:v2  
myapp:v3



Laptop



Server

Registries can be:  
Public (Docker Hub) – available for anyone  
Private (your own or Docker Hub)

# DOCKER — PRIVATE REGISTRIES

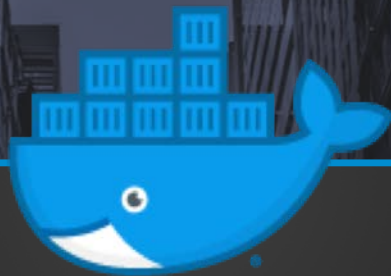
- By default, all images are pushed to and pulled from Docker Hub
- To push to a different registry, just include the URL to it in your image name. This tells Docker where to push or pull from.
  - For example, `mycompanyregistry.com/mynginx:1.0`
  - If the registry is secured, then use a `docker login` command to gain access



# DOCKER — A BLACK BOX

- Docker containers run the same based on the image every time. They are intended to be fully encapsulated.
- You can provide some basic runtime configurations
  - Port mapping from container to host
  - Volume mapping from container FS to host FS
  - Environment variables
  - Resource allowances (CPU, memory)
  - Network info
  - And much more

# DOCKER CLI





# DOCKER CLI

Docker command line interface gives us everything we need to build, interact, inspect, and run containers and images. Many of the commands mimic Linux commands. All commands start with “docker”.

Reference: <https://docs.docker.com/edge/engine/reference/commandline/docker/>

- Containers:
  - ps – list running containers (include -a to see stopped containers)
  - rm – remove container
  - inspect – inspect running container
  - start – start a stopped container
  - stop – stop running container
  - restart – restart a container
  - cp – copy files to/from a container to local filesystem
  - logs – show the console output from a container

# DOCKER CLI

- Images:
  - build – build an image from Dockerfile
  - images – show images
  - rmi – remove image
  - run – run a container from an image
    - -d – detached
    - -p – port mapping
    - -it – interactive (gets you a command line in the container)
    - --name – assign a name
    - -e – environment variable
    - -v – volume mapping
  - push – push the image to a repo
  - pull – pull an image from a repo
  - tag – tag an image



# DOCKER – RUNNING CONTAINERS

One command you'll use a lot is docker run. Let's talk about a few important concepts.

- Mappings – the left side is the host, the right is the container. This applies to ports, volumes, etc.
- Detached and interactive modes.
- Naming containers
- If an image doesn't exist on the host yet, it will be pulled before running.
- Run is a shortcut operation for:
  - Docker pull <image>
  - Docker create <image>
  - Docker start <container>
- `docker run -d -p <host>:<container> --name=<name> <image>`

# DOCKER: LAB



JENKINS



# DOCKER – JENKINS LAB

Let's get a crash course on spinning up Docker containers. I have not provided the commands to do this. You'll find it in the image page on Docker Hub!

Let's run a container of the Jenkins image on our local machines.

- Find the image on Docker Hub – review the instructions
- Run the image locally
- Bring up the UI in your browser
- Inspect the logs for the key to login
- Now spin up a new Jenkins container while the old one is still running. Then try to connect to it as well with a browser.

# DOCKER: LAB



MYSQL



# DOCKER – MYSQL LAB

Let's see some additional runtime parameters that can be passed in to a container.

Let's run a container of the Mysql image on our local machines.

- Find the image on Docker Hub – review the instructions
- Run the image locally (use tag 5.7). Pass in environment variables to allow access to the database and also volume mapping.
- If you have Mysql tools already installed on your workstation, try to connect, add some data.
- Once done Tim will demo it, changing data and then killing the container to demonstrate how the volume lives on.

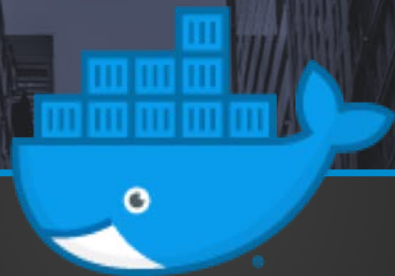
# DOCKER — MYSQL LAB

Command to run with port mapping, environment, and volume:

```
docker run --name some-mysql -v /Users/tsolley/Desktop/Temp/mysql:/var/lib/mysql -e  
MYSQL_ROOT_PASSWORD=my-secret-pw -d -p 3306:3306 mysql:5.7
```



# DOCKERFILES



# DOCKERFILE – OVERVIEW

- There are two ways to create an image in Docker:
  - With a Dockerfile
  - From an existing container
- A Dockerfile is an example of infrastructure as code. It represents a repeatable build of your Docker image. It can be stored in your code repo, versioned, and reviewed.
- Dockerfile is the preferred way to build images. You should have a good reason to do it the other way.
- Every line in a Dockerfile builds a new layer in your image.
- Commands are run from top to bottom, and each is independent from others and run from the directory where the Dockerfile is located.
  - In other words, unlike a bash script, you wouldn't cd to a directory in one line and then use that directory on the next.



# DOCKERFILE – COMMANDS

There are just a handful of simple commands:

- FROM – which image this inherits from
- MAINTAINER – An owner (no functionality)
- WORKDIR – Change to a directory until changed again. Works like cd
- USER – Change to a user
- RUN – Run a Linux command in a new layer. The command is dependent on which shell is in the image.
- EXPOSE – Tells what ports this container will listen on. Does not actually publish a port, but is rather like documentation.
- ADD – copy files into the container
- COPY – copy files into the container. Same as ADD.
- ENTRYPOINT – The command to be run when container is started. This is inherited from a base image.
- CMD – provides defaults to be run after the ENTRYPOINT. Also inherited from the base image.

# DOCKERFILE — DOCKERFILE WALKTHROUGH

Let's look at the Dockerfile for one of the containers that we'll be using later in the RV Store hackathon.



# DOCKERFILE — BUILDING THE IMAGE

Building an image is a simple CLI command:

```
docker build -t <name>:<tag> <location of Dockerfile>
```

```
docker build -t myimage:v1 .
```

This builds from the Dockerfile and tags it in one operation. The tag is optional but typically used. You would need to do a separate command to tag from the image ID if you omit it.

# DOCKERFILE: LAB



NGINX



# DOCKERFILE — NGINX LAB

1. Create a new Docker image using a Dockerfile that you'll write.
2. Use nginx:latest in your FROM statement
3. Create a new simple HTML. Any simple content will do.
4. In your Dockerfile, copy the file into the container at /usr/share/nginx/html
5. Build the image, tag it, and run it.
6. Check that you see your custom HTML in your browser. Nginx runs on port 80.

# DOCKERFILE: LAB



## INHERITANCE



# DOCKERFILE – INHERITANCE LAB

1. Let's see image inheritance in action.
2. Create a new directory
3. Write a new HTML file with a new name from the last lab.
4. Now write a new Dockerfile, and reference your updated Nginx image in the FROM line.
5. In your new Dockerfile copy the new HTML file into `/usr/share/nginx/html`
6. Build your new image and run it.
7. In a browser you should now be able to see your original HTML file from the last lab, and the new one.

# DOCKER – SINGLE PROCESS

- Docker is intended to run a single process per container. This is important as if the process faults, then the whole container can be thrown out and a new one created.
- You can get around this by using systemd or supervisord as the single process, which monitors and manages multiple other processes.
- This is not something you should make a habit of. There should be a very good reason to do this and you should avoid it if possible. Design your containers to run a single process.





WRAP UP