

Docker

Container platform

LOGISTICS REQUIRED

- ❖ Familiarity with linux operating system
- ❖ Linux boxes on our local machines with Internet connectivity
- ❖ Aware of basic application development
- ❖ Basic Knowledge on Networking(CIDR blocks, subnet etc.)
- ❖ Target Audience
 - System administrators
 - Software developers in a DevOps role
 - Anyone who wants to learn!!

ABOUT YOU

- ❖ I have not received any pre-course questionnaire!

So, please tell me about yourself:

- ❖ Your name
- ❖ Your background
- ❖ What is the purpose of the course?
- ❖ Where and how you will be using this knowledge?
- ❖ What do you currently know about Docker, Kubernetes, Ansible?

ABOUT ME

- ❖ My name is Deepak Gupta(@hellodk01)
- ❖ Total Industry Experience: 7+ Years
- ❖ Certifications:
 1. Blockchain for Developers
 2. Interfacing with the Raspberry Pi
 3. Big Data, Cloud Computing, & CDN Emerging Technologies

ABOUT ME

❖ Industry Roles

- Devops Lead, MoveinSync
- Systems Engineer, Myntra Designs
- Devops Engineer, Knowlarity Communications
- Software Engineer, Wipro Technologies

❖ Hobbies:

- Photography(hellodk01)
- Travelling
- Trekking

TRAININGS DELIVERED

- ❖ Cloud Computing : AWS Solutions, Azure DevOps
- ❖ Container Technologies : Docker, Kubernetes
- ❖ Monitoring Tools : Sensu, Zabbix, Nagios, Icinga2
- ❖ SQL Databases : MySQL, PostgreSQL, MariaDB
- ❖ NoSQL Databases : MongoDB, Cassandra, Redis, Gemfire
- ❖ Web Server : Nginx Setup and Configurations
- ❖ Messaging Tools : RabbitMQ, Kafka
- ❖ Configuration Management : Ansible, Chef, Puppet, Saltstack
- ❖ Architecture : Microservices, DevOps, DevSecOps
- ❖ Programming : Java, Python, Golang, haskell

COURSE

ORGANISATION

- ❖ Course starts at 14:00 hrs IST and finishes at 18:00 hrs IST
- ❖ Course ends on 23rd of November 2018
- ❖ We would be using Ubuntu 18.04 as our OS
- ❖ Organize yourself into groups
- ❖ Make sure members of each group sit together
- ❖ I hope lab details are already shared with you all



AGENDA

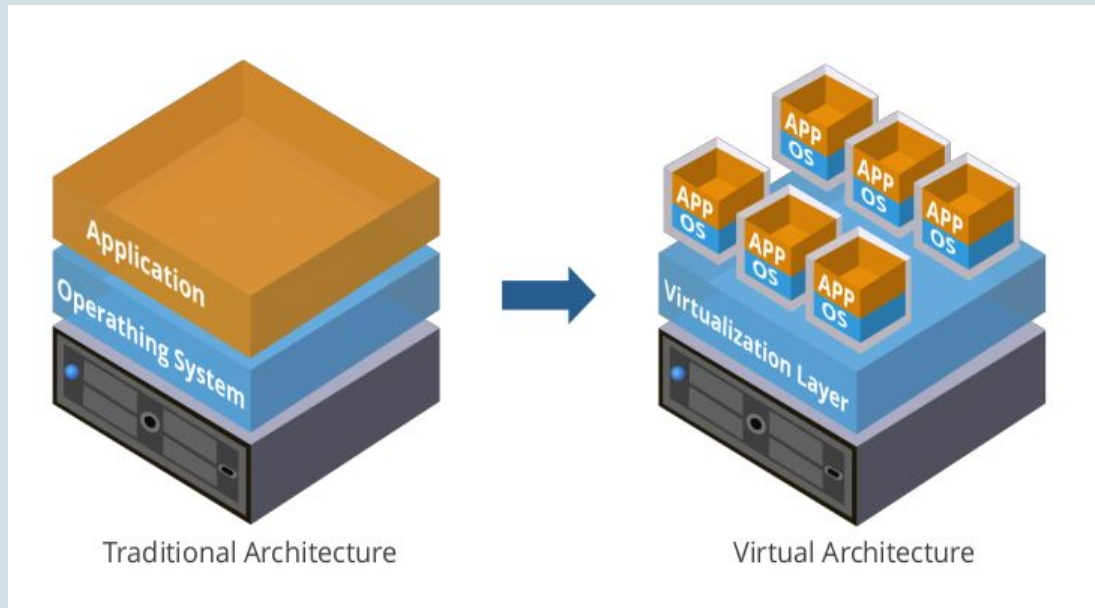
Agenda for Day 1

Module 1: Warm up!

- ❖ Benefits
- ❖ Introduction to containers
- ❖ Containers vs Virtual Machines
- ❖ Overall process flow
- ❖ Docker engine
- ❖ Registry

Introduction

- ❖ Virtualization - Need for virtualization and its implementations
 - important technique to address multiple issues



Virtual Machines



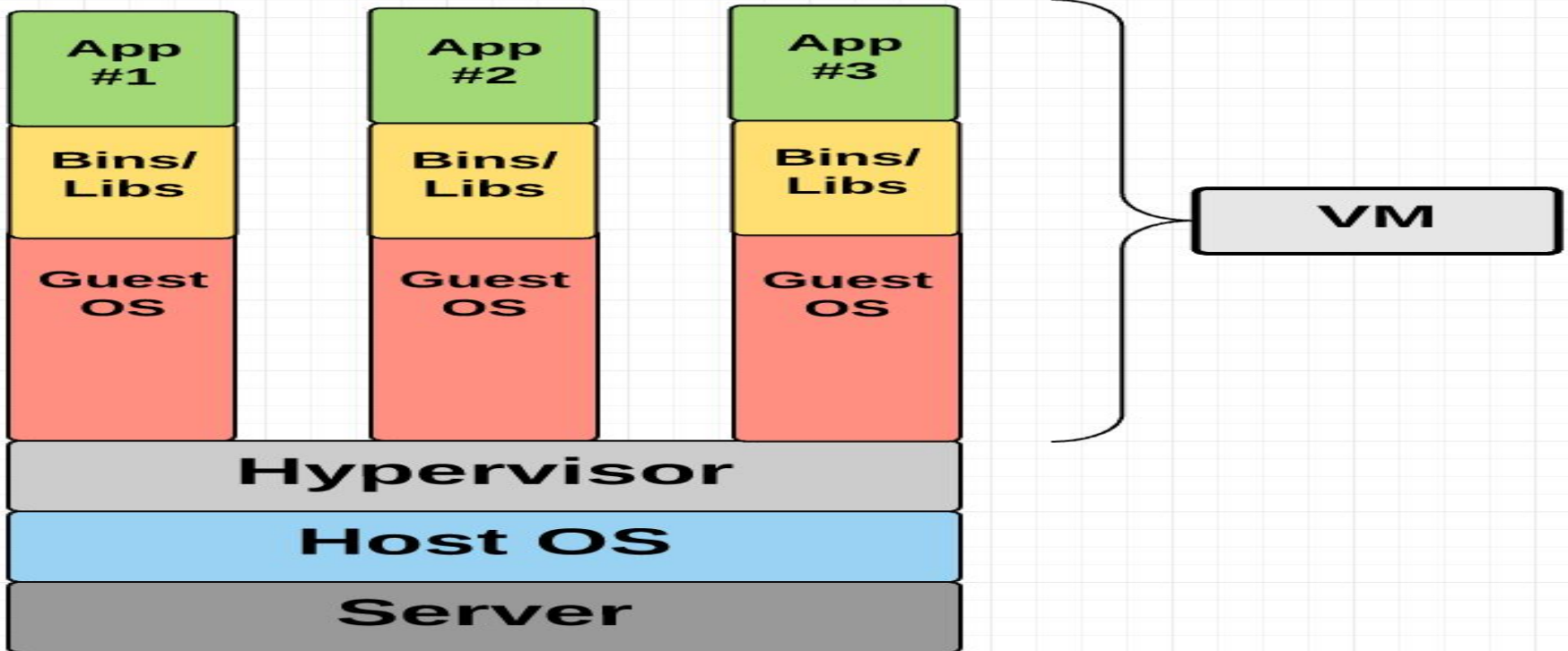
- What are virtual machines
- Hypervisor
- Host OS
- Guest OS

Virtual Machines



- Virtual machines are not new concepts
- 1960s computers used virtualization to allow multitasking between programs
- Each program appeared to have full access to the machine
- VMware has been a significant player in virtualizing the x86 architecture since 1998

Virtual Machines



Virtual Machines Advantages



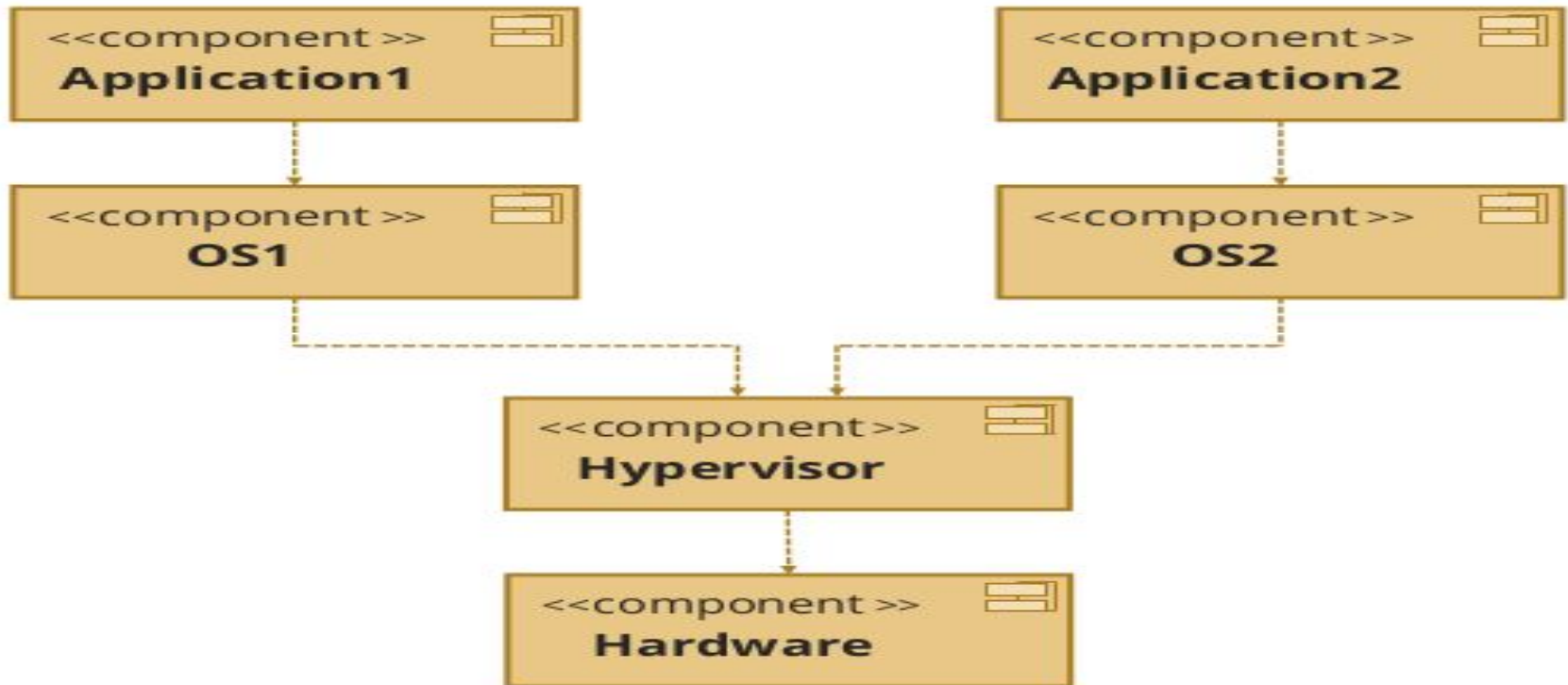
- a good way of setting up well defined environments
- simply copy the virtual machine; there is no need to install software on machines
- Several operating systems can be run on the same machine at the same time
- Virtual machines have largely replaced the need for multiple boot machines

Hypervisors



- Virtualization software is called a Hypervisor
- provides guest OS with configurable access to real and emulated hardware
- Type-1 or native or bare metal hypervisors
 - ◆ Run directly on the host hardware
 - ◆ Usually based on a modified Linux kernel
 - ◆ Guest operating systems run as processes on the host
 - ◆ Usually used on high end servers
 - ◆ Ex: Oracle VM Server, Citrix XenServer, VMware ESX/ESXi, Microsoft Hyper-V

Hypervisors

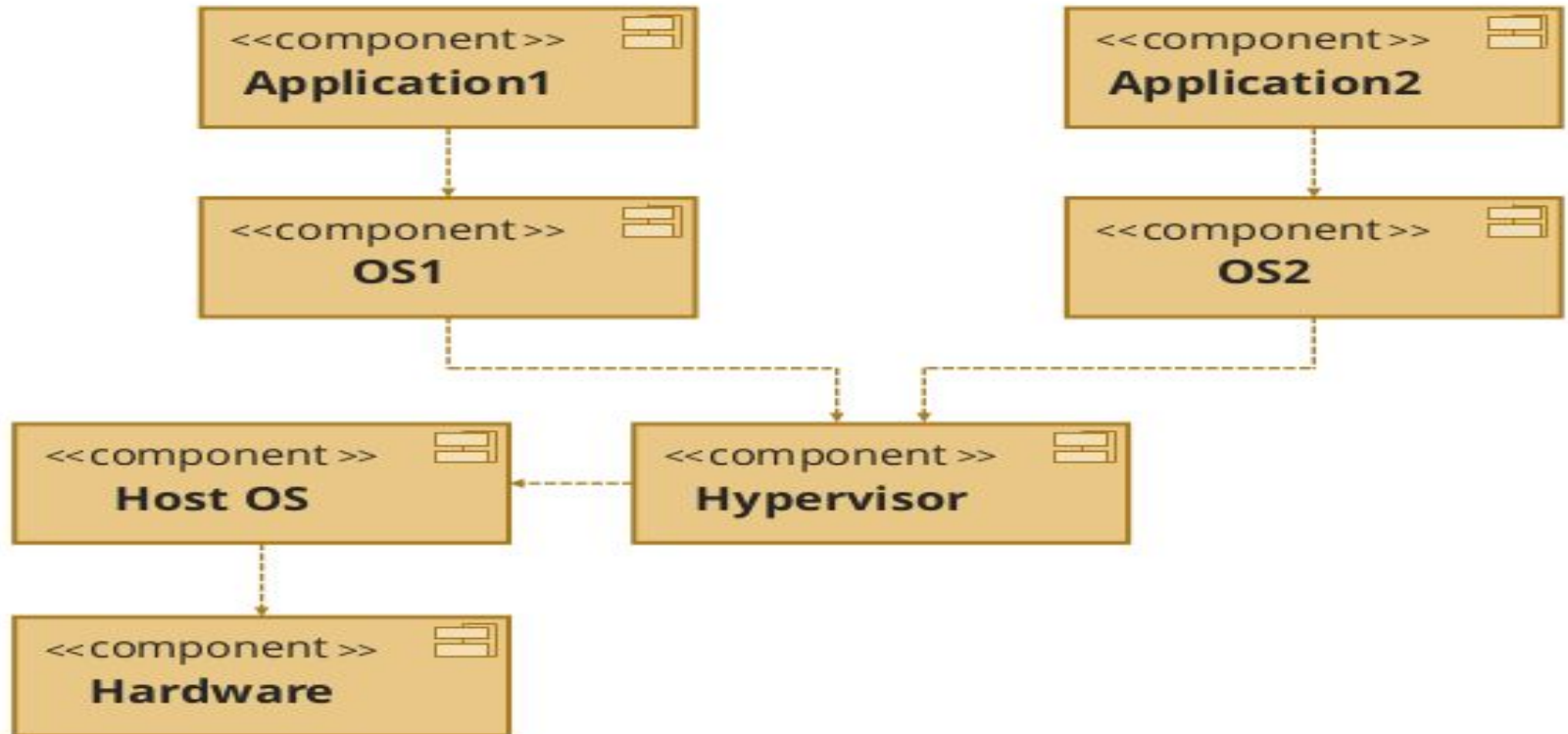


Hypervisors



- Type-2 or hosted hypervisors
- Run as a process on a conventional OS
- Often need to map the guest OS hardware calls onto host OS system calls
- Used on desktop and laptop computers
- Products include VMware Player, VMware Workstation, and Oracle VirtualBox

Hypervisors



Issues with Virtual Machines



- ❖ Virtual Machines are full Operating Systems with often performance issues
- ❖ typically created by installing an OS from, there may not be enough free resources
- ❖ typically very large
- ❖ Containers to the rescue....

Introduction to containers



- Analogy to real world container
- Definition of container:
 - ◆ Containerization is an approach to software development in which an application or service, its dependencies, and its configuration (abstracted as deployment manifest files) are packaged together as a container image
 - ◆ containerized application can be tested as a unit and deployed as a container image instance to the host operating system (OS)

Introduction to containers

Docker Host



Containers vs Virtual Machines



- Similar in goals: isolate an application and its dependencies into a self-contained unit
- These units can run anywhere
- containers and VMs remove the need for physical hardware
- efficient use of computing resources(energy consumption and cost effectiveness)
- main difference between containers and VMs is in their architectural approach

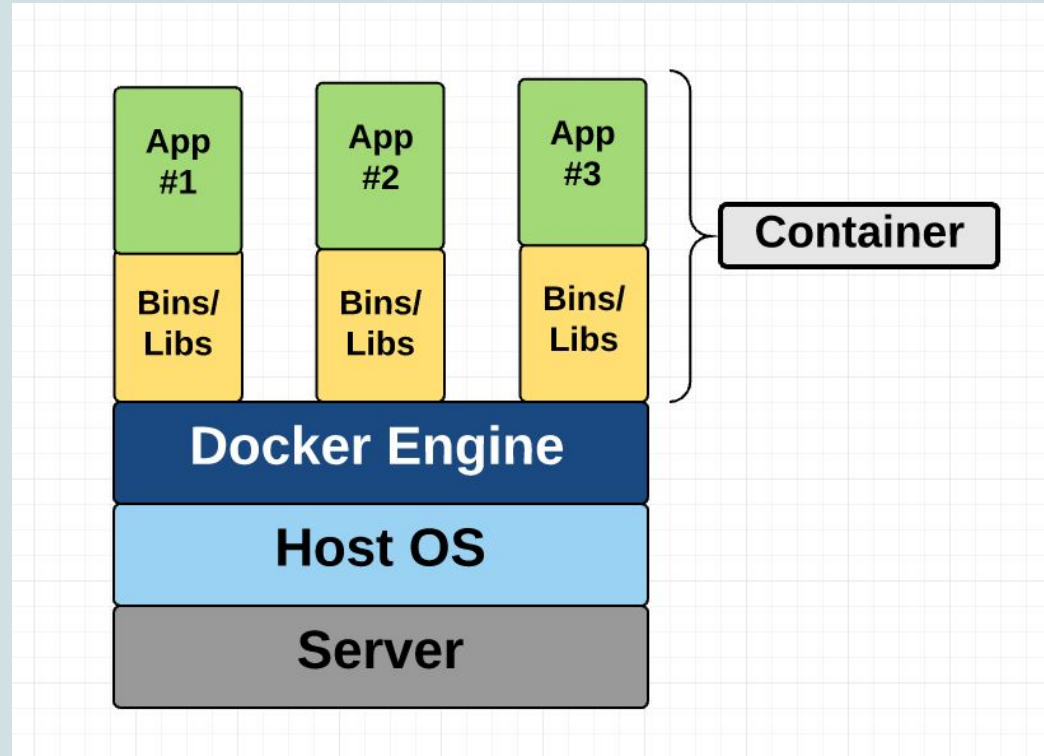
Container



- VM - hardware virtualization
- Container - operating-system-level virtualization by abstracting the “user space”
- containers have private space for processing
 - ◆ can execute commands as root
 - ◆ have a private network interface and IP address
 - ◆ allow custom routes and iptable rules
 - ◆ can mount file systems, and etc.
- big difference between containers and VMs is that containers *share* the host system's kernel with other containers

Container

- ❖ containers package up just the user space, and not the kernel or virtual hardware
- ❖ Each container gets its own isolated user space
- ❖ operating system level architecture is being shared across containers
- ❖ bins and libs are created
- ❖ containers are lightweight



Overall process flow



- Creating container image with Dockerfile
- Run service by creating a container, which will be running on the Docker host
- Store images in a registry:
 - ◆ Public registry
 - ◆ Private registry
- Putting images in a registry allows to store static and immutable application bits
- images can then be versioned and deployed in multiple environments and therefore provide a consistent deployment unit

Overall process flow



❖ Dockerfile:

- the instructions to build a Docker image

RUN apt-get y install some-package: to install a software package

EXPOSE 8000: to expose a port

ENV ANT_HOME /usr/local/apache-ant to pass an environment variable

and so forth

Sample Dockerfile



FROM golang:1.9.2-alpine3.6 AS build

RUN apk add --no-cache git

RUN go get github.com/golang/dep/cmd/dep

COPY Gopkg.lock Gopkg.toml /go/src/project/

WORKDIR /go/src/project/

RUN dep ensure -vendor-only

COPY . /go/src/project/

RUN go build -o /bin/project

FROM scratch

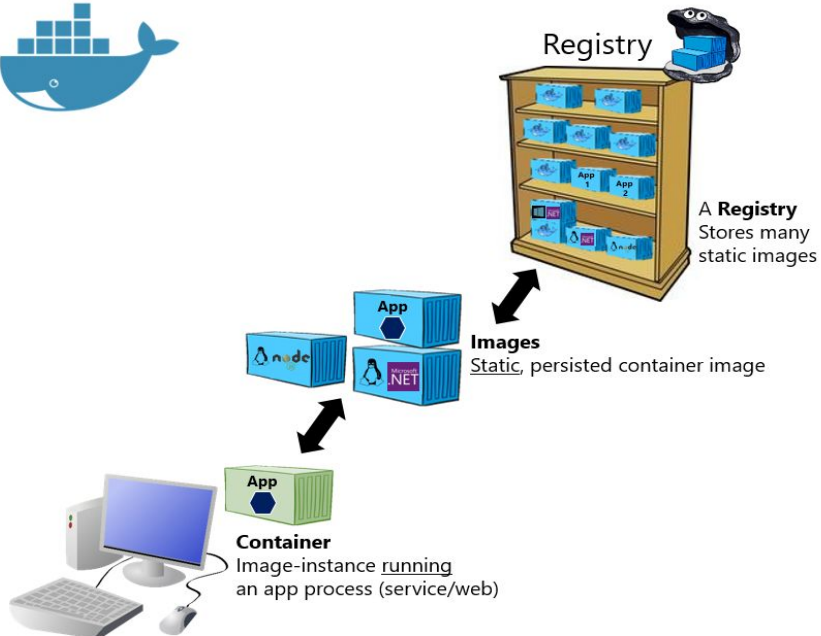
COPY --from=build /bin/project /bin/project

ENTRYPOINT ["/bin/project"]

CMD ["--help"]

Overall Process flow

Basic taxonomy in Docker



Hosted Docker Registry

Docker Trusted Registry on-prem.

On-premises
(n' private organizations)

Docker Hub Registry

Docker Trusted Registry on-cloud

Azure Container Registry

AWS Container Registry

Public Cloud
(specific vendors)

Google Container Registry

Quay Registry

Other Cloud

Docker Engine



→ Docker Engine

- ◆ layer on which Docker runs
- ◆ lightweight runtime and tooling that manages containers, images, builds, and more
- ◆ runs natively on Linux systems and is made up of:
 - A Docker Daemon that runs in the host computer
 - A Docker Client which communicates with the Docker Daemon
 - A REST API for interacting with the Docker Daemon remotely

Docker Engine



- Docker Client is what we, as the end-user of Docker, communicate with
- Think of it as the UI for Docker
- Example, when we do...
 - ◆ `docker build iampeekay/someImage`.
- we are communicating to the Docker Client, which then communicates our instructions to the Docker Daemon
- It is the primary user interface which communicates using a REST API
 - ◆ Over HTTP
 - ◆ Over local Unix socket

Docker Engine



- The Docker daemon is what actually executes commands sent to the Docker Client
 - ◆ Eg: building, running, and distributing our containers
- Docker Daemon runs on the host machine, but as a user, we never communicate directly with the Daemon
- Docker Client can run on the host machine as well, but it's not required to
- It can run on a different machine and communicate with the Docker Daemon that's running on the host machine

Registry



- Responsible for the storage, management, and delivery of Docker Images
 - ◆ Docker Hub
 - ◆ Private
 - ◆ Other vendors
- Those images can then be versioned and deployed in multiple environments and therefore provide a consistent deployment unit

Registry



- Private image registries, either hosted on-premises or in the cloud, are recommended when:
 - ◆ Our images must not be shared publicly due to confidentiality.
 - ◆ We want to have minimum network latency between our images and our chosen deployment environment

Creating Container



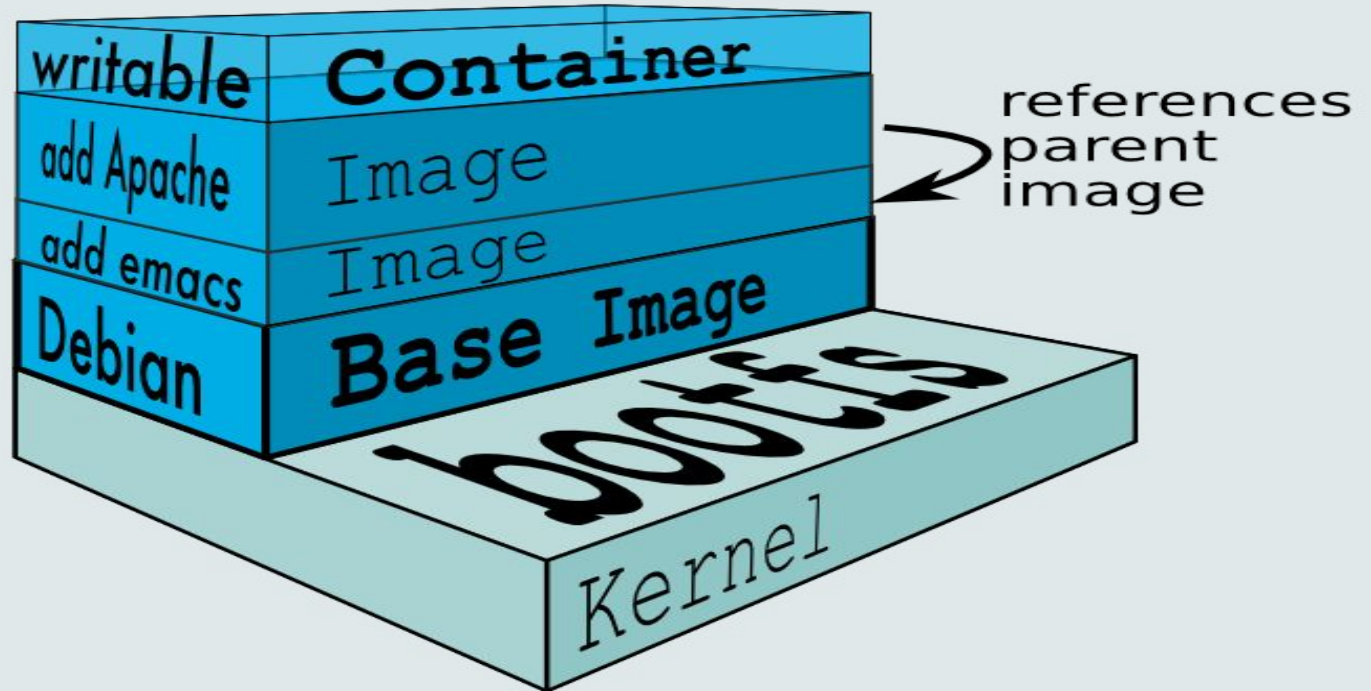
- Layers in container
- What is network interface?
- A Docker container wraps an application's software into an invisible box with everything the application needs to run
- includes the operating system, application code, runtime, system tools, system libraries, and etc.
- Docker containers are built off Docker images

Creating Container



- Since images are read-only, Docker adds a read-write file system over the read-only file system of the image to create a container
- Docker creates a network interface so that the container can talk to the local host, attaches an available IP address to the container, and executes the process that we specify to run our application when defining the image

Creating Container



Creating Container



- The term “container” is really just an abstract concept to describe how a few different features work together to visualize a “container”
- Namespaces provide containers with their own view of the underlying Linux system
- limit what the container can see and access
- run a container - Docker creates namespaces that the specific container will use
- several different types of namespaces in a kernel that Docker makes use of

Creating Container

→ Let's run through the different features real quick:

➤ Namespaces

- NET
- PID
- MNT
- UTS
- IPC
- USER

➤ Control groups

➤ Isolated Union File system

Creating Container



→ Namespaces

◆ NET:

- Provides a container with its own view of the network stack of the system
- Ex: network devices, IP addresses, IP routing tables, /proc/net directory, port numbers, etc.)

◆ PID: Process ID.

- gives containers their own scoped view of processes they can view and interact
- includes an independent init (PID 1), which is the “ancestor of all processes”

Creating Container



→ MNT:

- ◆ Gives a container its own view of the [“mounts” on the system](#), processes in different mount namespaces have different views of the filesystem hierarchy

→ UTS(UNIX Timesharing System.):

- ◆ allows a process to identify system identifiers(i.e. hostname, domainname, etc.)
- ◆ UTS allows containers to have their own hostname and NIS domain name which are independent of other containers and the host system

Creating Container



→ IPC(InterProcess Communication):

- ◆ isolates IPC resources between processes running inside each container

→ USER:

- ◆ isolates users within each container
- ◆ functions by allowing containers to have a different view of the uid (user ID) and gid (group ID) ranges, as compared with the host system

→ Docker uses these namespaces together in order to isolate and begin the creation of a container

Creating Container



→ Control groups(also called cgroups):

- ◆ a Linux kernel feature that isolates, prioritizes, and accounts for the resource usage (CPU, memory, disk I/O, network, etc.) of a set of processes
- ◆ a cgroup ensures that Docker containers only use the resources they need – and, if needed, set up limits to what resources a container **can** use
- ◆ Cgroups also ensure that a single container doesn't exhaust one of those resources and bring the entire system down

Orchestration overview - machine, swarm, compose



❖ Docker Machine:

- Docker Machine takes us from zero-to-Docker in seconds with a single command
- With Docker Machine, whether provisioning the Docker daemon on a new laptop, on virtual machines in the data center, or on a public cloud instance, the same, single command ...
- `% machine create -d [infrastructure provider] [provider options] [machine name]`

... gets the target host ready to run Docker containers

Orchestration overview - machine, swarm, compose

- ❖ A swarm is a group of machines that are running Docker and joined into a cluster
- ❖ Set up our swarm, create a couple of VMs using docker-machine, using VirtualBox:
 - `docker-machine create --driver virtualbox myvm1`
 - `docker-machine create --driver virtualbox myvm2`
- ❖ LIST THE VMS AND GET THEIR IP ADDRESSES:
 - `docker-machine ls`
- ❖ INITIALIZE THE SWARM AND ADD NODES
 - `docker-machine ssh myvm1 "docker swarm init --advertise-addr <myvm1 ip>"`

Orchestration overview - machine, swarm, compose



- ❖ To add a worker to this swarm, run the following command:
 - `docker swarm join --token <token> <myvm ip>:<port>`
- ❖ Join swarm:
 - `docker-machine ssh myvm2 "docker swarm join --token <token> <ip>:2377"`
- ❖ Run `docker node ls` on the manager to view the nodes in this swarm:
 - `docker-machine ssh myvm1 "docker node ls"`


Orchestration overview - machine, swarm, compose



→ Docker Compose

- After provisioning Docker daemons on any host in any location with Docker Machine and clustering them with Docker Swarm, users can employ Docker Compose to assemble multi-container distributed apps that run on top of these clusters
- In the following simple two-container app declaration, the first container is a Python app built each time from the Dockerfile in the current directory
- second container is built from the redis Official Repo on the Docker Hub Registry

Orchestration overview - machine, swarm, compose



→ Docker Compose

containers:

web:

build: .

command: python app.py

ports:

- "5000:5000"

volumes:

- ./code

links:

- redis

environment:

- PYTHONUNBUFFERED=1

redis:

image: redis:latest

command: redis-server --appendonly yes



AGENDA



Module 2: Setup and Installation

- ❖ Install docker from the ground up
- ❖ Docker images
- ❖ Docker Hub & searching for images
- ❖ Create multiple containers and explore the process of container creation
- ❖ Commands to work with containers
- ❖ Listing, starting, stopping and deleting containers
- ❖ Creating and removing images
- ❖ Looking inside a container / shell into a container.
- ❖ Exploring the ephemeral nature of containers.
- ❖ Discuss the best practice of saving data.
- ❖ Docker container and OS process relationship.

Install docker from the ground up



❖ Prerequisites:


➤ OS requirements

- To install Docker CE, we need a maintained version of CentOS 7
- centos-extras repository must be enabled
- overlay2 storage driver is recommended

➤ Uninstall old versions

- Older versions of Docker were called docker or docker-engine
- If these are installed, uninstall them, along with associated dependencies

Install docker from the ground up - CentOS 7



❖ *\$ sudo yum remove docker \
docker-client \
docker-client-latest \
docker-common \
docker-latest \
docker-latest-logrotate \
docker-logrotate \
docker-selinux \
docker-engine-selinux \
Docker-engine*

❖ It's OK if yum reports that none of these packages are installed

Install docker from the ground up

❖ Install Docker CE

- we can install Docker CE in different ways, depending on our needs:
 - set up Docker's repositories and install from them - recommended approach
 - download the RPM package and install it manually and manage upgrades
 - useful in situations such as installing Docker on air-gapped systems with no access to the internet

Install docker from the ground up



❖ Install using the repository

- Before we install Docker CE for the first time on a new host machine, we need to set up the Docker repository
- Afterward, we can install and update Docker from the repository
 - SET UP THE REPOSITORY
 - INSTALL DOCKER CE
 - UPGRADE DOCKER CE

Install docker from the ground up



- ❖ Install required packages

- *sudo yum install -y yum-utils device-mapper-persistent-data Lvm2*

- ❖ Use the following command to set up the stable repository

- *sudo yum-config-manager --add-repo*

- <https://download.docker.com/linux/centos/docker-ce.repo>

- ❖ Optional: Enable the edge and test repositories

- *sudo yum-config-manager --enable docker-ce-edge*

- *sudo yum-config-manager --enable docker-ce-test*

Install docker from the ground up



- ❖ Install the latest version of Docker CE, or go to the next step to install a specific version:
 - *sudo yum install docker-ce*
- ❖ To install a specific version of Docker CE:
 - *yum list docker-ce --showduplicates | sort -r*
 - *sudo yum install docker-ce-<VERSION STRING>*
- ❖ Start Docker
 - *sudo systemctl start docker*

Install docker from the ground up



- ❖ Verify that docker is installed correctly by running the hello-world image
 - *sudo docker run hello-world*
- ❖ This command downloads a test image and runs it in a container
- ❖ When the container runs, it prints an informational message and exits

Install docker from the ground up - Ubuntu

- ❖ `sudo apt-get remove docker docker-engine docker.io -y && sudo apt-get update -y`
- ❖ `sudo apt-get install apt-transport-https ca-certificates curl`
`software-properties-common -y`
- ❖ `curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -`
- ❖ `sudo apt-key fingerprint 0EBFCD88`
- ❖ `sudo add-apt-repository "deb [arch=amd64]`
`https://download.docker.com/linux/ubuntu \$(lsb_release -cs) \stable"`
- ❖ `sudo apt-get update -y && sudo apt-get install docker-ce -y`

Starting and Stopping Docker



- ❖ `/etc/init.d/docker start`
- ❖ `service docker start`
- ❖ `systemctl start docker`

Docker Images



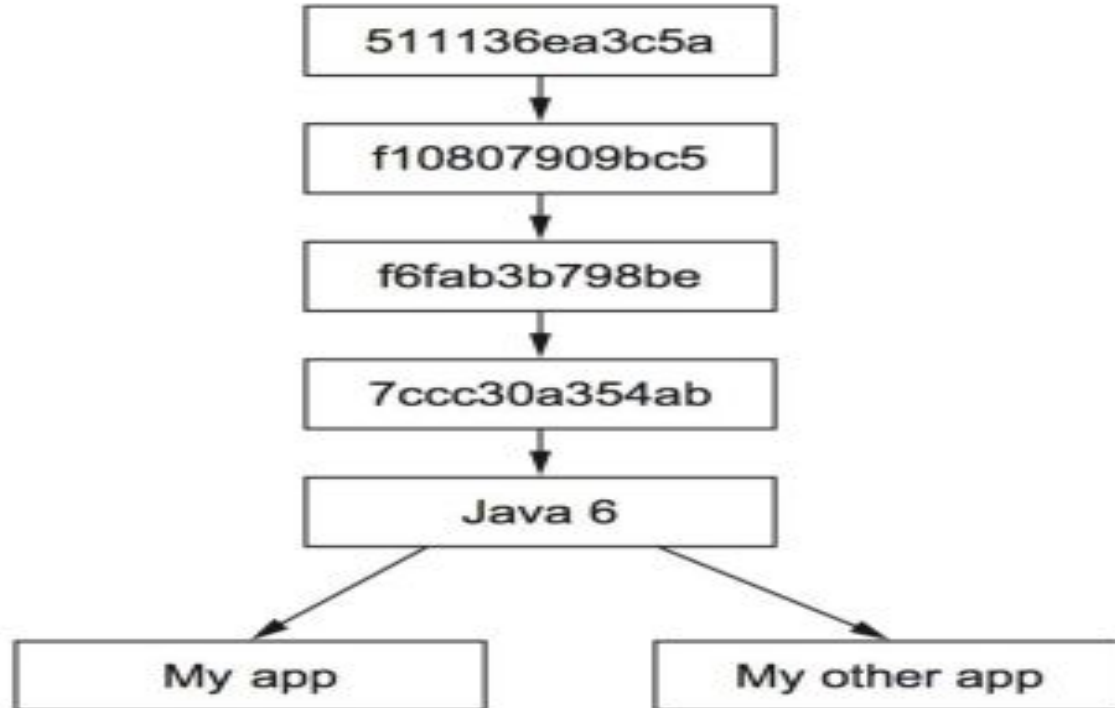
- Docker images are read-only templates
 - ◆ Foundation is a simplified version of the Linux operating system
 - ◆ Changes to foundation, such as application installations added to the Image
 - ◆ Images are the templates or build commands for Docker

Docker Images



- Docker images are built in layers
- Each layer is a file system & combined in a union file system to make a single image
- Images are the build component of Docker and start from a base image
- Foundation is usually a specifically prepared Linux operating system
- Custom base images can also be created
- Docker Image are then built by adding layers

Docker Images



Docker Images



- Docker has a number of base images on Docker Hub
 - ◆ Including many versions of Linux distributions
 - ◆ The image `centos:latest` will be used as a base
- Application images are created by adding layers to a base image
 - ◆ It can be done manually from a container running a shell
 - ◆ It can be automated using Docker's build process
- The Docker search facility can be used to search for images on Docker Hub

`docker search centos`

Docker Images



- Images can be listed using the images command
- Images can be deleted using the rmi command
 - ◆ *docker images*
 - ◆ *docker rmi centos-git*

Docker Images



→ Adding Packages:

◆ The CentOS base image is cut down

- It doesn't have the `which` or `ifconfig` commands
- We can install these from the command line using `yum`
 - *`docker run -it --name centos centos:latest`*
 - *`yum install -y which`*
 - *`yum install -y net-tools`*
 - *`Exit`*

Docker Images

→ Inspecting Images

- ◆ The images can be inspected to see the layers
 - The inspect command returns a JSON array by default
 - It can be used on images and containers
 - The `-s` or `--size` switch gives the size of a container
 - The `-f` or `--format` switch can be used to extract the JSON fields
 - *`docker inspect centos-net`*
 - *`docker inspect -f {{.Architecture}} centos`*

Docker Hub and searching for images



- Docker Hub is the default registry
 - ◆ It has a root namespace for official images
 - ◆ Root images include versions of supported Linux distributions
 - ◆ For example the nginx images can be addressed in different ways
 - *hub.docker.com/_/nginx:1.9*
 - *nginx:1.9*

Docker Hub and searching for images



→ Labels: used to uniquely identify images

- ◆ Labels look like URIs
- ◆ Components separated by /

→ Label components:

- ◆ Registry FQDN
- ◆ Namespace _ is for Docker Hub, r is for user
- ◆ User or organization name
- ◆ Repository name: tag

Docker Hub and searching for images



→ A tag is either a version number or a descriptive label

<https://hub.docker.com/r/databliss/netkernel-se/>

→ URI = Uniform Resource Identifier

→ FQDN = Fully Qualified Domain Name

Docker Hub and searching for images

→ `docker search [OPTIONS] TERM`

Options

| Name, shorthand | Default | Description |
|---|---------|--|
| <code>--automated</code> | | deprecated Only show automated builds |
| <code>--filter</code> , <code>-f</code> | | Filter output based on conditions provided |
| <code>--format</code> | | Pretty-print search using a Go template |
| <code>--limit</code> | 25 | Max number of search results |
| <code>--no-trunc</code> | | Don't truncate output |
| <code>--stars</code> , <code>-s</code> | | deprecated Only displays with at least x stars |

Create Multiple Containers and explore the process of container creation

- define each of the containers we want to deploy plus certain characteristics for each container deployment. `docker search [OPTIONS] TERM`
- Once we have a multi-container deployment description file, we can deploy the whole solution in a single action orchestrated by the `docker-compose up` CLI command

Create Multiple Containers and explore the process of container creation

version:vers '2'

services:

webmvc:

image: eshop/webmvc

environment:

- CatalogUrl=http://catalog.api

ports:

- "5100:80"

depends_on:

- catalog.api

catalog.api:

image: eshop/catalog.api

expose:

- "80"

ports:

- "5101:80"

extra_hosts:

- "CESARDLSURFBOOK:10.0.75.1"

Create Multiple Containers and explore the process of container creation

- The root key in this file is services. Under that key we define the services we want to deploy and run when we execute the docker-compose up command
- or when we deploy from Visual Studio by using this docker-compose.yml file

Create Multiple Containers and explore the process of container creation

- The docker `run` command starts a container based on a named Docker Image
 - ◆ Docker first looks for a local copy of the image
 - ◆ If does not exist it is pulled from a Docker Registry(Docker Hub-default Registry)
 - ◆ A new container is created using the file system from the image
 - ◆ A read-write layer is added to the top of the file system
 - ◆ A network interface is created and an IP address is assigned from a pool
 - ◆ Standard input, output, and error streams are connected
 - ◆ A specified application is executed

Create Multiple Containers and explore the process of container creation

- A Docker Image must be located on the local computer
 - ◆ It may have been created locally
 - ◆ It may have been pulled from a Registry
 - ◆ It may be missing
- The `pull` command insures that the specified image is on the local computer
 - ◆ It will transfer all constituent layers of the image as separate transfers

Create Multiple Containers and explore the process of container creation

- The `run` command creates and initiates a container based on the image
 - ◆ The example runs the latest CentOS image
 - ◆ It runs the command `command`
 - `docker pull centos:latest`
 - `docker run centos:latest whoami`
 - `docker run centos:latest pwd`
 - `docker run centos:latest date`

Create Multiple Containers and explore the process of container creation

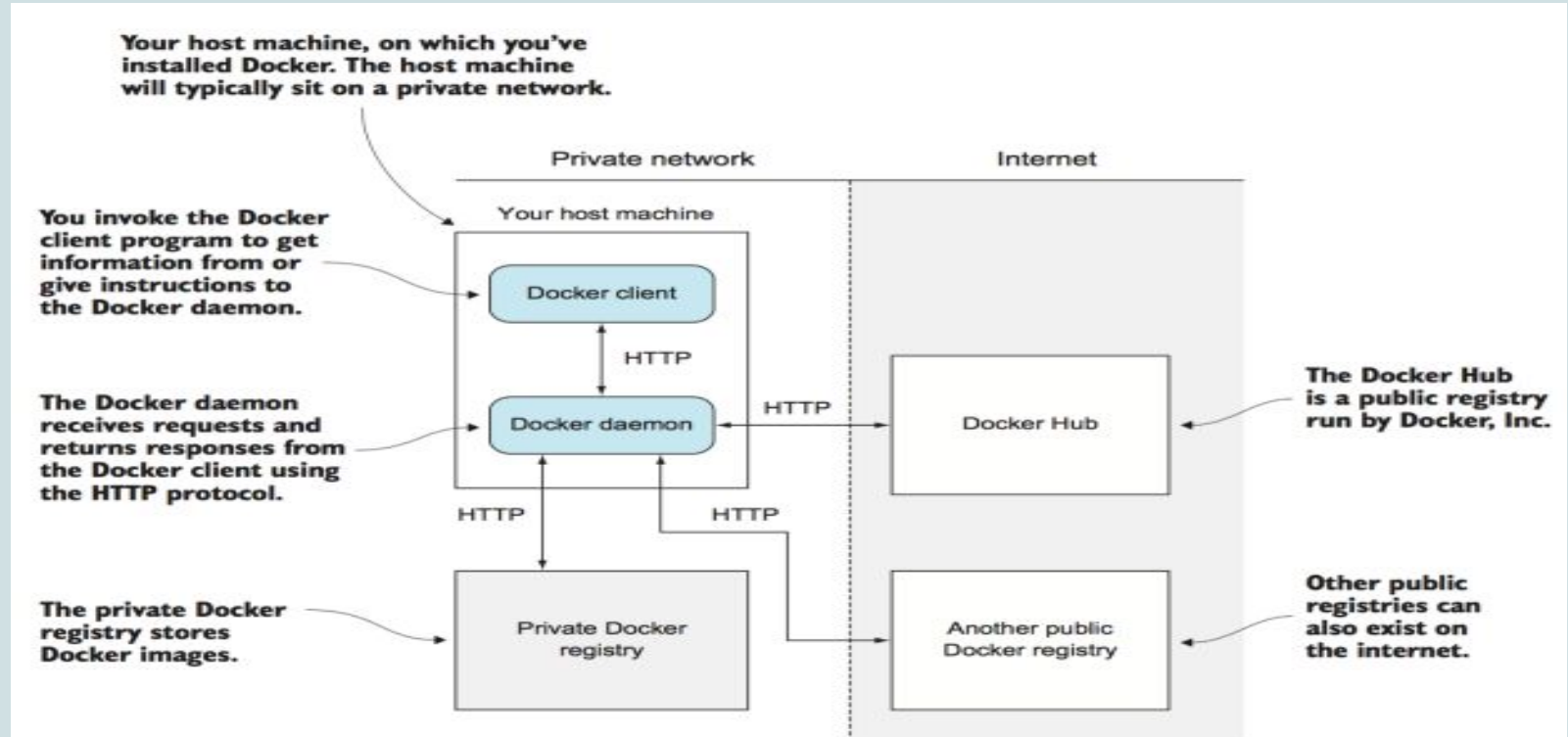
→ Running a Container Interactively

- ◆ To use a container interactively requires switches to the run command
 - -i or --interactive keeps STDIN open
 - • -t or --tty allocates a pseudo TTY

`docker run -i -t centos:latest /bin/bash`

Create Multiple Containers and explore the process of container creation

Overview of docker container



Commands to work with containers



- The docker run command first creates a writeable container layer over the specified image, and then starts it using the specified command. That is, docker run is equivalent to the API `/containers/create` then `/containers/(id)/start`
- A stopped container can be restarted with all its previous changes intact using docker start
- The docker run command can be used in combination with docker commit to change the command that a container runs

Commands to work with containers



- Attaching to a container attaches to the contained process's STDIN, STDOUT, and STDERR
 - ◆ we can attach with either the container ID or its name
 - ◆ Several command prompts can attach to the same container process
 - ◆ All tty sessions see the same input and outputs
 - ◆ The container ID is obtained using `ps`

Commands to work with containers



→ We can detach from a container and leave it running using `^p ^q`

→ On first terminal:

◆ *`docker run -it --name centos centos:latest /bin/bash`*

> date

→ On second terminal:

◆ *`docker attach centos`*

> date

Commands to work with containers

- Containers can be explicitly named using the `--name` switch
 - ◆ By default, Docker makes up a comical name such as `hungry_lumiere`
 - ◆ Most commands will accept either the name or the ID of the container
 - ◆ Note: The `ps` command accepts a switch `-a` for all containers

`docker run -it --name centos centos:latest /bin/bash`

Listing, starting, stopping and deleting containers

- Running containers can be listed using the `ps` command
 - ◆ Note that the container has been given a name
- The `stats` command shows running container resource usage, use `^C` to exit
 - ◆ `docker ps`
 - ◆ `docker stats`

Listing, starting, stopping and deleting containers

- Containers can be stopped using the stop command
 - ◆ It has an optional `-t` or `--time` parameter which defaults to 10 seconds
 - ◆ The main process, `PID=1`, is sent a `SIGTERM`
 - ◆ After the timeout interval, it is sent a `SIGKILL`
 - *`docker stop centos`*

Listing, starting, stopping and deleting containers

- Once a container is stopped, it is still available
 - ◆ The `ps -a` command shows all containers
- A stopped container can be started with the `start` command
 - ◆ The `-i` or `--interactive` switch connects STDIN
 - ◆ The `-a` or `attach` switch attaches STDIN and STDERR
 - *`docker ps -a`*
 - *`docker start -ai centos`*

Listing, starting, stopping and deleting containers

→ Docker containers can be removed with the `rm` command

- ◆ The `-f` or `--force` switch causes running containers to be force stopped for removal
- ◆ The `ps -aq` command shows all containers IDs to enable all to be removed

`docker rm centos`

`docker rm $(docker ps -aq)`

Listing, starting, stopping and deleting containers

→ Containers can be run in the background

◆ Use the run command with the `-d` or `--detach` switch

→ Signals can be sent using the kill command

◆ The default signal is SIGTERM

```
docker run -d --name webserver nginx
```

```
docker kill webserver
```

Creating and removing images



→ Dockerfile contains build directives

- ◆ FROM defines the starting image
- ◆ MAINTAINER defines the email address of the builder

FROM centos:latest

MAINTAINER phill@totaleclipse.eu

Creating and removing images



- Images can be listed using the images command
 - ◆ Images can be deleted using the rmi command

docker images

docker rmi centos-git

Looking inside a container / shell into a container


→ The Dockerfile SHELL command defines the default shell to use

- ◆ It must be specified in JSON form
- ◆ The default for Linux and windows are shown
- ◆ It takes the form `SHELL ["executable", "parameters"]`


`SHELL ["/bin/sh", "-c"]`

`SHELL ["cmd", "/S", "/C"]`

Exploring the ephemeral nature of containers

- 
- To understand the impact of containers, it's important to consider a concept that's coupled with containers—microservices, a method of developing applications into small distributed processes
 - A classic example of a microservice is a web server. As demand for an application rises, an application can spawn additional Apache web servers, each running in an individual container
 - Once demand subsides, the system resources used to spawn instances of the microservice can be released

Exploring the ephemeral nature of containers

- 
- Containers are ephemeral
 - Data and system settings exist only during the life of each container's run cycle
 - The ephemeral nature of containers lends itself to the microservices architecture
 - Each time an application needs to spawn a new microservice, a request is made to the container management system



- ❖ Setting up the environment
- ❖ Basics of Docker
- ❖ Installation of Docker
- ❖ Playing around with images

We will call this a day!!

That's all for Day 1

Day 2





AGENDA

Day 2

Module 3: Docker Engine

- ❖ Dockerise a web application
- ❖ See logs of a container
- ❖ Mapping ports from container to the host
- ❖ Creating a mount point
- ❖ Creating a custom image using commit
- ❖ Introduction and explore dockerfile and build process
- ❖ Explore the key dockerfile commands (from, run, cmd, env, expose)
- ❖ Introduction to layers & optimizing layers in a Docker image



AGENDA

Day 2

Module 3: Docker Engine

- ❖ Dockerize MySQL
- ❖ Create a custom schema and create an image
- ❖ Access the MySQL from any MySQL client and create data
- ❖ Docker network
- ❖ Container linking
- ❖ Building a multi-container application

Recapitulation



❖ Vagrant commands

- executed from the same directory where you have placed the Vagrantfile
- `vagrant up dockerM`
- `vagrant ssh dockerM`
- `vagrant halt dockerM`
- `vagrant destroy dockerM` # DO NOT EXECUTE THIS COMMAND
- `vagrant global-status --prune`

Recapitulation



❖ Docker Search Commands:

- `docker search centos`
- `docker search redis`
- `docker search ubuntu`
- `docker search mysql`
- `docker search jenkins`
- `docker search elasticsearch`
- `docker search java`

Recapitulation



❖ Managing Images:

- docker images
- docker rmi <image name>
- docker inspect centos_image
- docker pull centos:latest\
- docker run centos:latest whoami
- docker run centos:latest date
- docker run centos:latest ls -ltr

Recapitulation



❖ Adding package to an image:

- `docker run -it --name <container name> <image name>`
- Then install your dependent packages etc.
- `yum install -y which`
- `yum install -y net-tools`
- Exit

Recapitulation



❖ Manage Containers:

- `docker ps`
- `docker ps -a`
- `docker run -it --name centos_image centos:latest`
- `docker run -i -t --name centos_container_attach centos:latest /bin/bash`
- `docker attach centos_container_attach`

Recapitulation



❖ Manage Containers:

- `docker stats`
- `docker stop centos_container_attach`
- `docker start -ai centos_container_attach`
- `docker rm centos_container_attach`
- `docker rm $(docker ps -aq)`
- `docker run -d --name webserver nginx`
- `docker kill webserver`

Dockerise a web application



❖ Install epel repository

- `yum install -y epel-release`
- `yum update -y`
- `yum install -y npm`
- `cd /home/vagrant`
- `mkdir node_app`
- `cd node_app`

Dockerise a web application



- ❖ Create a docker container for nodejs application
 - Create package.json
 - Run npm install
 - Create nodejs application
 - Create Dockerfile
 - Build docker image
 - Run docker container

Dockerise a web application

❖ Package.json

```
{  
  "name": "docker_web_app",  
  "version": "1.0.0",  
  "description": "Node.js on Docker",  
  "author": "First Last <hello.dk@outlook.com>",  
  "main": "server.js",  
  "scripts": {  
    "start": "node server.js"  
  },  
  "dependencies": {  
    "express": "^4.16.1"  
  }  
}
```


Dockerise a web application

- ❖ Run npm-install: If we are using npm version 5 or later, this will generate a package-lock.json file which will be copied to our Docker image

- ❖ Create web application file server.js

```
'use strict';
const express = require('express');
// Constants
const PORT = 8080;
const HOST = '0.0.0.0';
// App
const app = express();
app.get('/', (req, res) => {
  res.send('Hello world\n'); });
app.listen(PORT, HOST);
console.log(`Running on http://${HOST}:${PORT}`);
```

Dockerise a web application

❖ Create Dockerfile

```
FROM node:carbon  
# Create app directory  
WORKDIR /usr/src/app  
# Install app dependencies  
# A wildcard is used to ensure both package.json AND package-lock.json are copied  
# where available (npm@5+)  
COPY package*.json ./  
RUN npm install  
# Bundle app source  
COPY . .  
EXPOSE 8080  
CMD [ "npm", "start" ]
```

❖ Create .dockerignore file with the following content

```
node_modules  
Npm-debug.log
```

Dockerise a web application



- ❖ Verify the server by running it locally
 - `node server.js`
- ❖ Now go to your browser
 - `Localhost:8080`
- ❖ Once it's verified that this service is up and running properly, lets get this packaged!!

Dockerise a web application



- ❖ Build docker image: Go to the directory that has our Dockerfile and run the following command to build the Docker image
 - *docker build -t <our username>/node-web-app .*
 - *docker build -t hellodk/node-web-app .*
- ❖ List images
 - *docker images*
- ❖ Run the image
 - *docker run -p 49160:8080 -d <our username>/node-web-app*

See Logs of the container



❖ Print the output of the containerised app:

➤ *Get container ID*

- `docker ps`

➤ *Print app output*

- `docker logs <container id>`

❖ *Enter the container*

➤ `docker exec -it <container id> /bin/bash`

Mapping ports

- ❖ test our app, get the port of our app that Docker mapped:

➤ *\$ docker ps*

| <i>ID</i> | <i>IMAGE</i> | <i>COMMAND</i> | <i>...</i> | <i>PORTS</i> |
|---------------------|---|------------------|------------|-----------------------|
| <i>ecce33b30ebf</i> | <i><our username>/node-web-app:latest</i> | <i>npm start</i> | <i>...</i> | <i>49160->8080</i> |

- ❖ Docker mapped the 8080 port inside of the container to the port 49160 on our machine
 - ❖ call our app using curl
- *curl -i localhost:49160*

Create a mountpoint

- ❖ Data volumes can be created explicitly using the create command
- ❖ Data volumes can be created implicitly using the run command
- ❖ Any number of data volumes can be created and mounted in a container
- ❖ The --volume or -v option can be used with the Docker run command
- ❖ The mount point directory gets created automatically
 - `docker volume create --name mydata #explicit`
 - `docker run -it --name centos --volume mydata:/opt/data centos /bin/bash #implicit`

Create a mountpoint

- ❖ Data volumes can be located using Docker inspect on a container
- ❖ Look for the Mounts section in the output
- ❖ By default they have rather long names and have been shortened using:

➤ `docker inspect centos`

```
"Mounts": [  
  {  
    "Name": "6f9f9518...d25561f58bdc3e",  
    "Source": "/var/lib/docker/volumes/6f9f9518...d25561f58bdc3e/_data",  
    "Destination": "/opt/data",  
    "Driver": "local",  
    "Mode": "",  
    "RW": true,  
    "Propagation": ""  
  } ],
```


Create a mountpoint

- ❖ Data volumes can be listed & removed, below command just prints volume name
 - `docker volume ls -q`
- ❖ Volumes can't be removed if in use by containers & can be created and named
- ❖ can be mounted into a container explicitly
 - `docker volume ls`
 - `docker volume rm $(docker volume ls -q)`
 - `docker volume create --name jabber`
 - `docker volume ls`
 - `docker run -it --name centos --volume jabber:/opt/data centos /bin/bash`

Create a mountpoint

❖ Options

- The `-v` or `--volume` parameter takes the form
 - `[host-path:]container-path[:options]`
- Options are comma separated
 - The option `ro` mounts the volume read-only
 - There is also `rw` for read-write which is the default
- If there are files under the container mount point they get copied into the volume

Creating custom image using commit

- ❖ Steps involved in creating a custom image:
 - Pull down an image
 - Run the container
 - Modify the container to meet the basic needs of our developers,
 - Commit those changes to a new image

Creating custom image using commit

- ❖ Pulling the image and running the container
 - first step is to pull the latest NGINX image
 - `sudo docker pull nginx`
 - Once the image has downloaded, run it such that we can use the terminal window like so:
 - `sudo docker run --name nginx-template-base -p 8080:80 -d nginx`

Creating custom image using commit

- ❖ Accessing and modifying the container

- `docker ps`

- `sudo docker exec -it CONTAINER_ID bash`

Where `CONTAINER_ID` is the ID presented to us when we ran the run command

- ❖ After running this command we will find our self in the terminal of the running container Now, let's

add the necessary software for the template

- ❖ To do this, issue the following commands:

- `apt-get install nano build-essential tree telnet -y`

Creating custom image using commit

- ❖ Exit the container and commit the changes
 - Now that we've modified the container we have to commit the changes
 - First exit the container with the command `exit`
 - `exit`
 - commit the changes and create a new image based on said changes, issue the command:
 - `sudo docker commit CONTAINER_ID nginx-template`

Where `CONTAINER_ID` is the ID given to us when we initially ran the container
- ❖ If we issue the command *`docker images`*, we should now see the new image

Introduction and explore dockerfile and build process

❖ Dockerfile Syntax

- Dockerfiles use simple, clean, and clear syntax
- strikingly easy to create and use
- Dockerfile syntax consists of two kind of main line blocks: comments and commands + arguments

Line blocks used for commenting

❖ A Simple Example:













Print "Hello docker!"

RUN echo "Hello docker!"

Introduction and explore dockerfile and build process

- ❖ Dockerfile starts with
 - FROM ubuntu:18.04
- ❖ That says pull the base image of ubuntu, above which other commands runs and image is created based on the commands
- ❖ When we build from the dockerfile each build will have layers of the final image
- ❖ Link for official docker images:
 - <https://hub.docker.com/explore/>

Docker Images

| | |
|--|--|
|  nginx official |  ubuntu official |
|  alpine official |  postgres official |
|  busybox official |  node official |
|  redis official |  mysql official |
|  httpd official |  memcached official |
|  mongo official |  registry official |

Introduction and explore dockerfile and build process

- ❖ Docker image creation can be automated
- ❖ Create a directory containing all the files required for the build
- ❖ Add a file called Dockerfile which defines the build process
- ❖ The directory becomes the build context
- ❖ Each command in the build file creates a layer of the image
- ❖ A new container is created at each stage

Explore the key dockerfile commands

❖ ADD

- Usage: ADD [source directory or URL] [destination directory]
- ADD /my_app_folder /my_app_folder
- Source can also be a link

❖ CMD

- Usage 1: CMD application "argument", "argument", ..
- CMD "echo" "Hello docker!"
- Not executed during build, but only when a container is instantiated

Explore the key dockerfile commands

❖ ENTRYPOINT

- Usage example with CMD:
 - Arguments set with CMD can be overridden during **run**
 - CMD "Hello docker!"
 - ENTRYPOINT echo

❖ ENV

- Usage: ENV key value
 - ENV SERVER_WORKS 4
- Sets environment variables in key:value structure

Explore the key dockerfile commands

❖ EXPOSE

➤ Usage: EXPOSE [port]

■ EXPOSE 8080

❖ FROM

➤ Usage: FROM [image name]

■ FROM ubuntu

Explore the key dockerfile commands

❖ MAINTAINER

➤ Usage: MAINTAINER [name]

■ MAINTAINER authors_name

❖ USER

➤ Usage: USER [UID]

■ USER 751

➤ used to set the UID (or username) which is to run the container based on the image being built

Explore the key dockerfile commands

❖ RUN

- Usage: RUN [command]
 - RUN aptitude install -y riak
- Central executing directive for Dockerfile
- takes a command as its argument and runs it to form the image
- used to build the image (forming another layer on top of the previous one which is committed)

Explore the key dockerfile commands

❖ VOLUME

➤ Usage: VOLUME ["/dir_1", "/dir_2" ..]

■ VOLUME ["/my_files"]

❖ WORKDIR

➤ Usage: WORKDIR /path

■ WORKDIR ~/

➤ set where the command defined with CMD is to be executed.

Dockerizing Python Web Application

- ❖ `yum install -y python-pip`
- ❖ `pip install requests flask`
- ❖ `mkdir webapp_python`
- ❖ `cd webapp_python/`
- ❖ `vim app.py`

```
➤ from flask import Flask
    app = Flask(__name__)
    @app.route('/')
    def hello_world():
        return 'Hello World!'
    if __name__ == '__main__':
        app.run(debug=True,host='0.0.0.0')
```

Dockerizing Python Web Application

❖ vim requirements.txt

➤ Flask==0.12

❖ vim Dockerfile

```
# Format: FROM repository[:version]
```

```
FROM ubuntu:latest
```

```
COPY ./app
```

```
WORKDIR /app
```

```
RUN apt-get update -y
```

```
RUN apt-get install -y python-pip python-dev build-essential
```

```
RUN pip install -r requirements.txt
```

```
ENTRYPOINT ["python"]
```

```
CMD ["app.py"]
```

Dockerizing Python Web Application

- ❖ `docker build -t helloworldapp:latest .`
- ❖ `docker run -d -p 5000:5000 helloworldapp`
- ❖ access the app using hostname:5000 port
 - `curl -i localhost:5000`

Introduction to layers and optimizing layers in a Docker image

- ❖ Create dockerfile:

```
FROM debian:wheezy
RUN mkdir /tmp/foo
RUN fallocate -l 1G /tmp/foo/bar
```

- ❖ Let's build this image:

➤ `docker build -t sample`

Step 0: FROM debian:wheezy ---> e8d37d9e3476

Step 1: RUN mkdir /tmp/foo ---> Running in 3d5d8b288cc2 ---> 9876aa270471 Removing intermediate container 3d5d8b288cc2

Step 2: RUN fallocate -l 1G /tmp/foo/bar ---> Running in 6c797329ee43 ---> 3ebe08b36733 Removing intermediate container 6c797329ee43 Successfully built 3ebe08b36733

Introduction to layers and optimizing layers in a Docker image

- ❖ read the output of the docker build command we can see exactly what Docker is doing to construct our sample image:
 - Using the value specified in our FROM instruction, Docker will docker run a new container from the debian:wheezy image (the container's ID is 3d5d8b288cc2)
 - Within that running container Docker executes the mkdir /tmp/foo instruction
 - The container is stopped, committed (resulting in a new image with ID 9876aa270471) and then removed

Introduction to layers and optimizing layers in a Docker image

- Docker spins-up another container, this time from the image that was saved in the previous step (this container's ID is 6c797329ee43)
- Within that running container Docker executes the `fallocate -l 1G /tmp/foo/bar` instruction
- The container is stopped, committed (resulting in a new image with ID 3ebe08b36733) and then removed

Introduction to layers and optimizing layers in a Docker image

- ❖ Each layer is itself an image - layers are nothing more than a collection of other images
- ❖ In the same way that we could say: `docker run -it sample:latest /bin/bash`, we could just as easily execute one of the untagged layers:
 - `docker run -it 9876aa270471 /bin/bash`
- ❖ Both are images that can be turned into running containers -- the only difference is that the former has a tag associated with it ("sample:latest") while the later does not
- ❖ ability to create a container from any image layer can be really helpful when trying to debug problems with our Dockerfile

Introduction to layers and optimizing layers in a Docker image

- ❖ Image Size can be reduced by:
 - Choosing our Base wisely
 - Reusing our Base
 - Chaining our commands
 - Flatten our image

Introduction to layers and optimizing layers in a Docker image

- ❖ Sample image that we created just now can be converted into tar file

- *docker save sample > sample.tar*

- *ls -lh sample.tar*

- *-rw-r--r-- 1 core core 1.1G Jul 26 02:35 sample.tar*

- ❖ Creating new image with the following file gives the same size image

- FROM debian:wheezy*

- RUN mkdir /tmp/foo*

- RUN fallocate -l 1G /tmp/foo/bar*

- RUN rm /tmp/foo/bar*

Introduction to layers and optimizing layers in a Docker image

❖ Choosing our base

- If using Ubuntu when BusyBox would actually meet our needs we're consuming a lot of extra space unnecessarily

❖ Reuse our Base

- Each of the images adds something on top of the debian:wheezy image
- but there aren't three copies of Debian
- Instead each image simply maintains a reference to the single instance of the Debian layer

Introduction to layers and optimizing layers in a Docker image

❖ Chain our commands

```
FROM debian:wheezy
WORKDIR /tmp
RUN wget -nv http://centurylinklabs.com/someutility-v1.0.0.tar.gz && \
    tar -xvf someutility-v1.0.0.tar.gz && \
    mv /tmp/someutility-v1.0.0/someutil /usr/bin/someutil && \
    rm -rf /tmp/someutility-v1.0.0 && \
    rm /tmp/someutility-v1.0.0.tar.gz
```

Introduction to layers and optimizing layers in a Docker image

❖ Chain our commands

➤ <https://github.com/nginxinc/docker-nginx/blob/f4d30145c60c433966df96f618d78513fee9d322/mainline/stretch/Dockerfile>

Dockerize mysql

- ❖ Pull the image

- `docker pull mysql/mysql-server:latest`

- ❖ Run the image:

- `docker run --name=mysql1 -p 3306:3306 --network host -d mysql/mysql-server:latest`

- ❖ Wait for a while and :

- Get password

- `docker logs mysql1 2>&1 | grep GENERATED`

Create a custom schema and create an image

- ❖

Get inside the container:
 - `docker exec -it mysql1 mysql -uroot -p`
- ❖ `mysql> ALTER USER 'root'@'localhost' IDENTIFIED BY 'root';`
- ❖ `mysql> create database mydb;`
- ❖ `mysql> use mydb;`
- ❖ `mysql> CREATE TABLE pet (name VARCHAR(20), owner VARCHAR(20), species VARCHAR(20), sex CHAR(1));`

Create a custom schema and create an image

- ❖ `mysql> Insert into pet values('cat', 'Deepak', 'xyz' , 'M');`
- ❖ `mysql> exit`
- ❖ Commit the changes in the container:
 - `sudo docker commit CONTAINER_ID mysql-template`

Docker Network

- ❖ When we install Docker, it creates three networks automatically

➤ `docker network ls`

| NETWORK ID | NAME | DRIVER |
|--------------|--------|--------|
| 7fca4eb8c647 | bridge | bridge |
| 9f904ee27bf5 | none | null |
| cf03ee007fb4 | host | host |

- ❖ When we run a container, we can use the `--network` flag to specify which networks our container should connect to
- ❖ The bridge network represents the `docker0` network present in all Docker installations
- ❖ The Docker daemon connects containers to this network by default

Docker Network

- ❖ The none network adds a container to a container-specific network stack
- ❖ The container lacks network interface. Attaching to such a container & looking at its stack:
 - `docker run --network none --name nonenetcontainer -d mongo`
 - `docker exec -it nonenetcontainer /bin/bash`
 - `root@0cb243cd1293:/# cat /etc/hosts`

```
127.0.0.1    localhost
::1         localhost ip6-localhost ip6-loopback
fe00::0     ip6-localnet
ff00::0     ip6-mcastprefix
ff02::1     ip6-allnodes
ff02::2     ip6-allrouters
```

Docker Network

- ❖ The host network adds a container on the host's network stack
- ❖ As far as the network is concerned, there is no isolation between the host machine and the container
- ❖ If we run a container that runs a web server on port 80 using host networking, the web server is available on port 80 of the host machine

Docker Network

- ❖ default bridge network is present on all Docker hosts
- ❖ The docker network inspect command returns information about a network:
 - docker network inspect bridge

```
[
  {.....
    "Driver": "bridge",
    "IPAM": {
      "Driver": "default",
      "Config": [
        {
          "Subnet": "172.17.0.1/16", "Gateway": "172.17.0.1"
        }
      ]
    },
    "Containers": {},
    "Options": {
      "com.docker.network.bridge.name": "docker0", "com.docker.network.driver.mtu": "9001"
    },
    "Labels": {}
  }
]
```

Docker Network

- ❖ Run the following two commands to start two busybox containers, which are each connected to the default bridge network

- `docker run -itd --name=container1 busybox`

3386a527aa08b37ea9232cbcace2d2458d49f44bb05a6b775fba7ddd40d8f92c

- `docker run -itd --name=container2 busybox`

94447ca479852d29aeddca75c28f7104df3c3196d7b6d83061879e339946805c

Docker Network

-
- ❖ Inspect the bridge network again after starting two containers
 - ❖ Make note of their IP addresses, which will be different on our host machine than in the example below

Docker Network

```
$ docker network inspect bridge
```

```
{[{
  .....,
  "Containers": {
    "3386a527aa08b37ea9232cbcace2d2458d49f44bb05a6b775fba7ddd40d8f92c": {
      "EndpointID": "647c12443e91faf0fd508b6edfe59c30b642abb60dfab890b4bdccee38750bc1",
      "MacAddress": "02:42:ac:11:00:02",
      "IPv4Address": "172.17.0.2/16",
      "IPv6Address": ""
    },
    "94447ca479852d29aeddca75c28f7104df3c3196d7b6d83061879e339946805c": {
      "EndpointID": "b047d090f446ac49747d3c37d63e4307be745876db7f0ceef7b311cbba615f48",
      "MacAddress": "02:42:ac:11:00:03",
      "IPv4Address": "172.17.0.3/16",
      "IPv6Address": ""
    }
  },
  "Options": {
    ...
  }, "Labels": {}
}]}
```

Docker Network

- ❖ Containers connected to the default bridge network can communicate with each other by IP address
 - *docker attach container1*
 - *root@3386a527aa08:/# ip -4 addr*

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1
inet 127.0.0.1/8 scope host lo
valid_lft forever preferred_lft forever
633: eth0@if634: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc
noqueue
inet 172.17.0.2/16 scope global eth0
valid_lft forever preferred_lft forever

Docker Network

- ❖ From inside the container, use the ping command to test the network connection to the IP address of the other container.

➤ `root@3386a527aa08:/# ping -w3 172.17.0.3`

```
PING 172.17.0.3 (172.17.0.3): 56 data bytes
64 bytes from 172.17.0.3: seq=0 ttl=64 time=0.096 ms
64 bytes from 172.17.0.3: seq=1 ttl=64 time=0.080 ms
64 bytes from 172.17.0.3: seq=2 ttl=64 time=0.074 ms
```

```
--- 172.17.0.3 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.074/0.083/0.096 ms
```


Docker Network

- ❖ Use the cat command to view the /etc/hosts file on the container
- ❖ This shows the hostnames and IP addresses the container recognizes

➤ root@3386a527aa08:/# cat /etc/hosts

```
172.17.0.2    3386a527aa08
127.0.0.1    localhost
::1          localhost ip6-localhost ip6-loopback
fe00::0      ip6-localnet
ff00::0      ip6-mcastprefix
ff02::1      ip6-allnodes
ff02::2      ip6-allrouters
```

Docker Network

- ❖ User-defined networks
 - `docker network create --driver bridge isolated_nw`
 - `docker network inspect isolated_nw`

```
[ {  
  "Name": "...",  
  ...  
  "Driver": "bridge",  
  "IPAM": {  
    "Driver": "default",  
    "Config": [  
      {  
        "Subnet": "172.21.0.0/16",  
        "Gateway": "172.21.0.1/16"  
      } ] },  
  "Containers": {},  
  "Options": {},  
  "Labels": {}  
}]
```

Docker Network

❖ docker network ls

| NETWORK ID | NAME | DRIVER |
|--------------|-------------|--------|
| 9f904ee27bf5 | none | null |
| cf03ee007fb4 | host | host |
| 7fca4eb8c647 | bridge | bridge |
| c5ee82f76de3 | isolated_nw | bridge |

❖ After we create the network, we can launch containers on it using the docker run

--network=<NETWORK> option

➤ `docker run --network=isolated_nw -itd --name=container3 busybox`

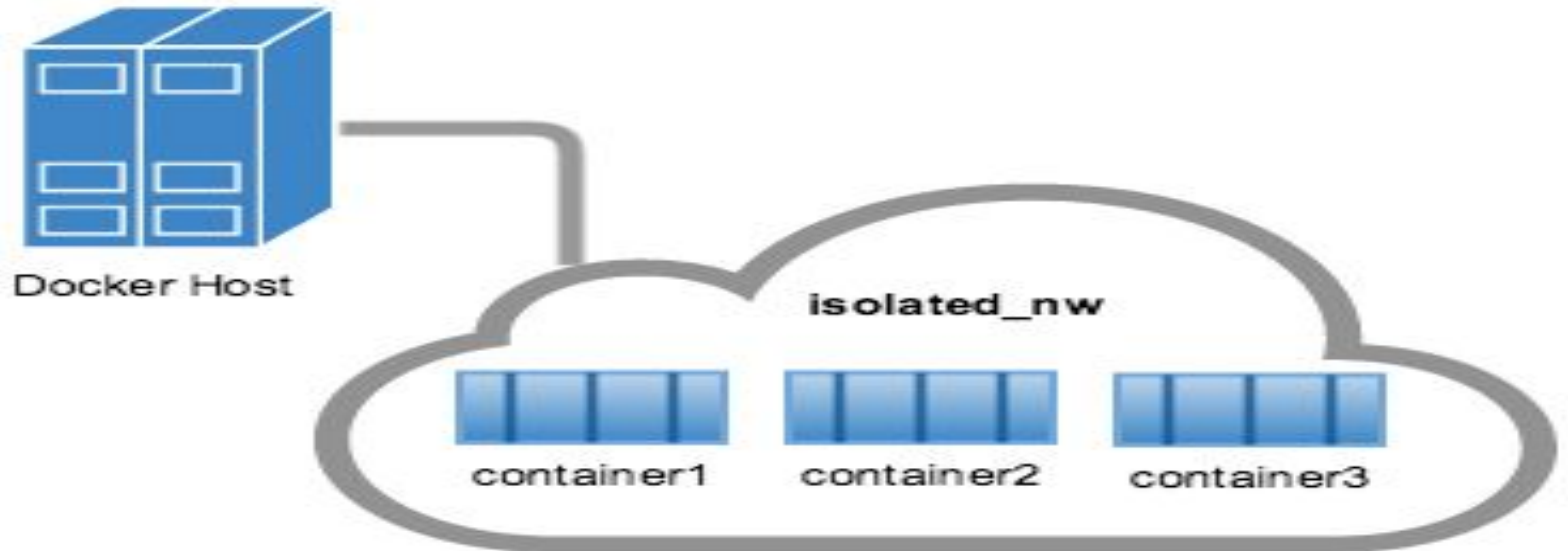
Docker Network

❖ `docker network inspect isolated_nw`

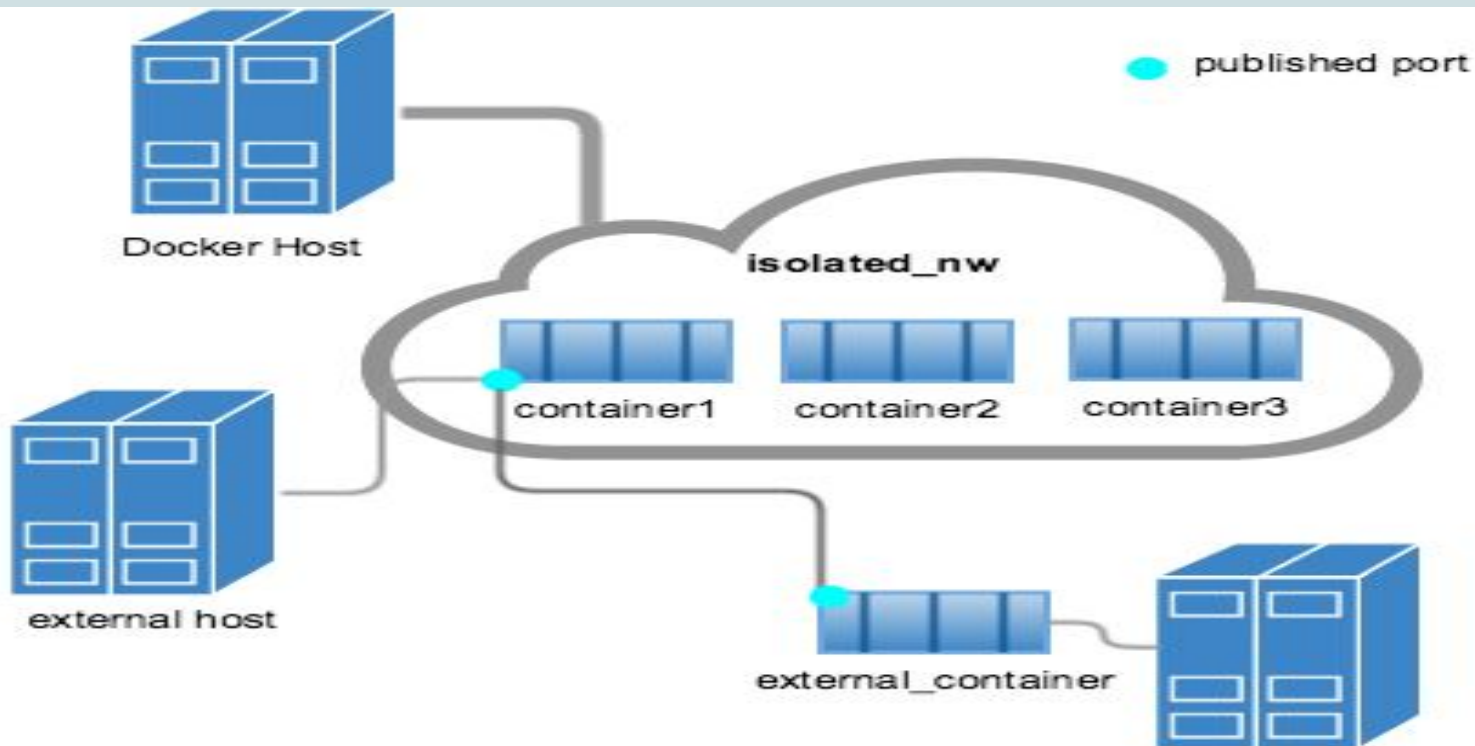
```
[
  {
    ...
    "Driver": "bridge",
    "IPAM": {
      "Driver": "default",
      "Config": [
        {}
      ]
    },
    "Containers": {
      "8c1a0a5be480921d669a073393ade66a3fc49933f08bcc5515b37b8144f6d47c": {
        "EndpointID": "93b2db4a9b9a997beb912d28bcfc117f7b0eb924ff91d48cfa251d473e6a9b08",
        "MacAddress": "02:42:ac:15:00:02",
        "IPv4Address": "172.21.0.2/16",
        "IPv6Address": ""
      }
    },
    "Options": {},
    "Labels": {}
  }
]
```

Docker Network

- ❖ containers in the network can communicate with other containers in the network
- ❖ the network itself isolates the containers from external networks



Docker Network



We will call this a day!!

That's all for Day 2



AGENDA

Day 3

Module 4: Cloud and Container Providers

- ❖ Creating a private repository in Docker Hub
- ❖ Publish custom images to the private repository on Docker Hub
- ❖ Explore the models of Docker Hub and CI
- ❖ Managing containers and Instances that run containers
- ❖ Docker Swarm

Recapitulation

- ❖ Image vs Containers?
 - Image creation
 - Running a container
- ❖ Dockerizing web applications
- ❖ Seeing container logs
- ❖ Entering the containers
- ❖ Hitting the web service using curl command
- ❖ Creating volumes

Recapitulation

- ❖ Creating mount points(explicit and implicit)
- ❖ Listing volumes
- ❖ Creating custom image by modifying a container and making commit to that
- ❖ Dockerfile syntax fundamentals
- ❖ Docker layers and it's optimization techniques
- ❖ Docker networks and its types
- ❖ Differentiate between a container and shell prompt

Pending

❖ Python Web Application

- `yum install -y python-pip && pip install requests flask`
- `mkdir /home/vagrant/webapp_python`
- `cd webapp_python/`
- `vim app.py, vim requirements.txt, vim Dockerfile`
- `docker build -t helloworldapp:latest .`
- `docker run -d -p 5000:5000 helloworldapp`

❖ access the app using hostname:5000 port

Building a multi-container application

- ❖ Define services in `docker-compose.yml` when building a multi-container Docker application
- ❖ explicitly describe how we want to deploy our multi-container application in the `docker-compose.yml` file
- ❖ define each of the containers we want to deploy plus certain characteristics for each container deployment
- ❖ deploy the whole solution in a single action orchestrated by the `docker-compose up` CLI command

Building a multi-container application

- ❖ Step 1: Setup
- ❖ Define the application dependencies
- ❖ Create a directory for the project:
 - `mkdir composetest`
 - `cd composetest`
- ❖ Create a file called `app.py` in our project directory and paste this in:

Building a multi-container application

```
import time
import redis
from flask import Flask
app = Flask(__name__)
cache = redis.Redis(host='redis', port=6379)
def get_hit_count():
    retries = 5
    while True:
        try:
            return cache.incr('hits')
        except redis.exceptions.ConnectionError as exc:
            if retries == 0:
                raise exc
            retries -= 1
            time.sleep(0.5)
```

```
@app.route('/')
def hello():
    count = get_hit_count()
    return 'Hello World! I have been seen
    {} times.\n'.format(count)

if __name__ == "__main__":
    app.run(host="0.0.0.0", debug=True)
```

Building a multi-container application

- ❖ In this example, redis is the hostname of the redis container on the application's network - default port for Redis, 6379
- ❖ Handling transient errors
- ❖ basic retry loop lets us attempt request multiple times if the redis service is unavailable
- ❖ useful at startup while the application comes online, but also makes our application more resilient if the Redis service needs to be restarted
- ❖ In a cluster, this also helps handling momentary connection drops between nodes

Building a multi-container application

- ❖ Create another file called requirements.txt in our project directory and paste this in:
 - flask
 - Redis
- ❖ Create a Dockerfile
 - FROM python:3.4-alpine
 - ADD ./code
 - WORKDIR /code
 - RUN pip install -r requirements.txt
 - CMD ["python", "app.py"]

Building a multi-container application

- ❖ This tells Docker to:
 - Build an image starting with the Python 3.4 image.
 - Add the current directory `.` into the path `/code` in the image.
 - Set the working directory to `/code`.
 - Install the Python dependencies.
 - Set the default command for the container to `python app.py`.

Building a multi-container application

- ❖ Step 3: Define services in a Compose file
- ❖ Create a file called docker-compose.yml in our project directory and paste the following:

```
version: '3'
services:
  web:
    build: .
    ports:
      - "5000:5000"
  redis:
    image: "redis:alpine"
```

- ❖ This Compose file defines two services, web and redis

Building a multi-container application

- ❖ The web service:
 - Uses an image that's built from the Dockerfile in the current directory
 - Forwards the exposed port 5000 on the container to port 5000 on docker host
 - We use the default port for the Flask web server, 5000
- ❖ The redis service uses a public Redis image pulled from the Docker Hub registry

Building a multi-container application

- ❖ Step 4: Build and run our app with Compose
- ❖ From our project directory, start up our application by running docker-compose up
 - `docker-compose up`
- ❖ Enter `http://0.0.0.0:5000/` in a browser to see the application running
- ❖ Refresh the page and see the change

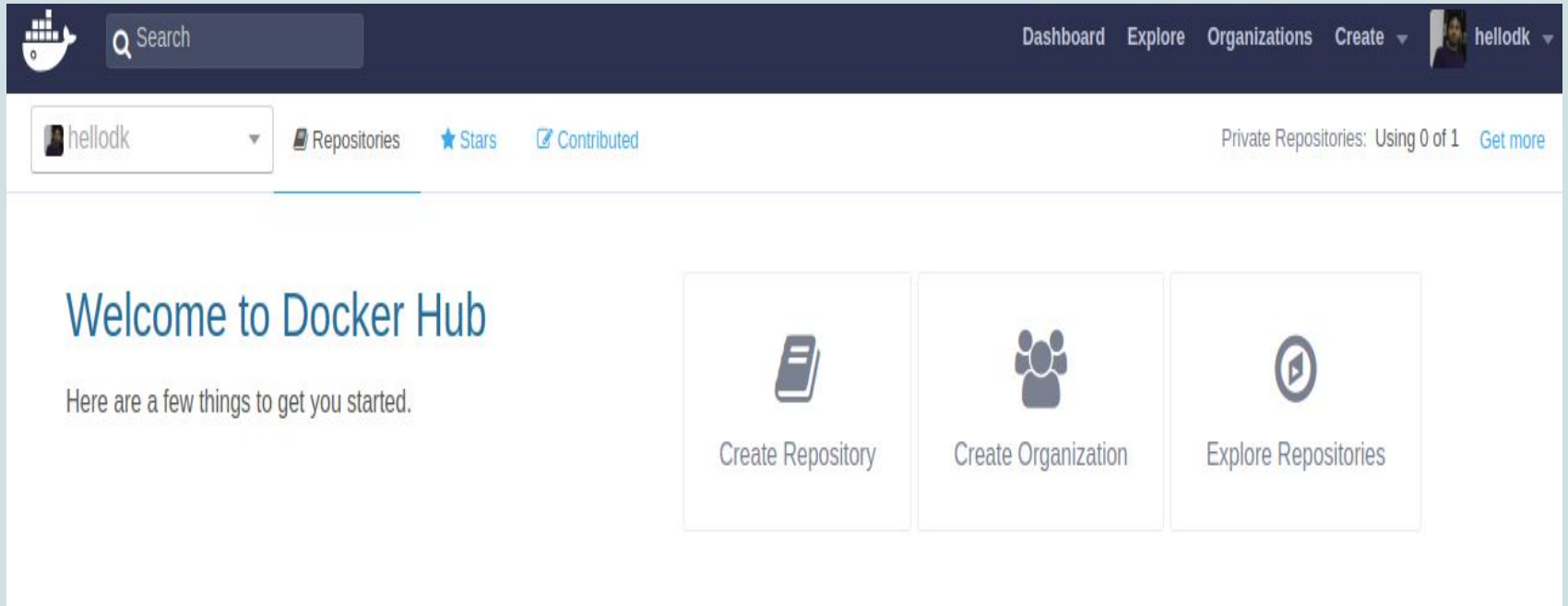
Creating a private repository in Docker Hub

- ❖ first create a Docker Hub user, we see a “Get started with Docker Hub.” screen, from which we can click directly into “Create Repository”
 - `hub.docker.com`
- ❖ When creating a new repository, choose to put it in our Docker ID namespace, or that of any organization that we are in the “Owners” team
- ❖ The Repository Name needs to be unique in that namespace
- ❖ can be two to 255 characters, can only contain lowercase letters, numbers or - and _

Creating a private repository in Docker Hub

- ❖ Select private repository from dropdown, hit the “Create” button
- ❖ we then need to docker push images to that Hub based repository

Creating a private repository in Docker Hub



The screenshot shows the Docker Hub interface for a user named 'hellodk'. The top navigation bar includes the Docker logo, a search bar, and links to Dashboard, Explore, Organizations, Create, and the user's profile. Below the navigation bar, the user's name 'hellodk' is displayed in a dropdown menu, followed by tabs for Repositories, Stars, and Contributed. On the right, it shows 'Private Repositories: Using 0 of 1' with a 'Get more' link. The main content area features a 'Welcome to Docker Hub' message and a list of three action cards: 'Create Repository' (with a document icon), 'Create Organization' (with a group of people icon), and 'Explore Repositories' (with a magnifying glass icon).

hellodk

Repositories Stars Contributed

Private Repositories: Using 0 of 1 [Get more](#)

Welcome to Docker Hub

Here are a few things to get you started.

- Create Repository
- Create Organization
- Explore Repositories

Creating a private repository in Docker Hub

Create Repository

1. Choose a namespace (*Required*)
2. Add a repository name (*Required*)
3. Add a short description
4. Add markdown to the full description field
5. Set it to be a private or public repository

hellodk

test-repo

hello test repo

This a test repo for demo purpose

Visibility

private

Create

Publish custom images to the private repository on Docker Hub

- ❖ In a terminal window, set the environment variable `DOCKER_ID_USER` as our username in Docker Cloud
 - `export DOCKER_ID_USER="username"`
- ❖ This allows we to copy and paste the commands directly from this tutorial
 - `docker login`
- ❖ This logs us in using our Docker ID, shared between Docker Hub and Docker Cloud

Publish custom images to the private repository on Docker Hub

- ❖ Tag our image using docker tag
 - ❖ In the example below replace my_image with our image's name, and DOCKER_ID_USER with our Docker Cloud username if needed
 - `docker tag my_image $DOCKER_ID_USER/my_image`
 - `docker images`
- ❖ Push our image to Docker Hub using docker push (making the same replacements as in the previous step)
 - `docker push $DOCKER_ID_USER/my_image`

Publish custom images to the private repository on Docker Hub

- ❖ Login to dockerhub.com
- ❖ Go to the repository we just created
- ❖ Click on settings
- ❖ Look for the option “make private” - makes this repository as a private repository

Private Registry -setup, push an image

- ❖ Private registries are a good solution for the following cases:
 - Provide a local image cache to speed up image loading
 - Allow teams to share images locally
 - Store images specific to a project lifecycle stage, development, and UAT
 - Guarantee that the registry will be available for as long as required

Private Registry -setup, push an image

- ❖ easiest way to create a private registry is to use a pre-built Docker container
 - Docker Hub has a number of registry images including the official one
 - The registry images can be pulled from Docker Hub
 - `mkdir registry`
 - `cd registry`
 - `docker pull registry`

Private Registry -setup, push an image

- ❖ registry image can now be run as a container
 - It needs to be run as a daemon container
 - The `-p` option exposes the container ports as local ports
 - It usually uses port 5000
 - `docker run -p 5000:5000 -d registry`

Private Registry -setup, push an image

- ❖ The registry is identified by hostname:port
 - For example localhost:5000
 - Images need to be named with the registry prefix
 - For example, localhost:5000/alpine
 - The image can then be pushed into the registry or pulled from it
 - `docker pull alpine`
 - `docker tag alpine:latest localhost:5000/alpine:latest`
 - `docker push localhost:5000/alpine`

Explore the models of dockerhub and CI

| Challenges with CI/CD |
|--|
| Difficulty supporting diverse language stacks and tooling |
| Slow provisioning and setup of build and test environments |
| Low throughput of jobs and software shipped to stage or production |
| Inconsistencies between environments |

Explore the models of dockerhub and CI

Docker Accelerates CI/CD

Eliminate system and language conflicts by isolating in containers

Run more jobs faster

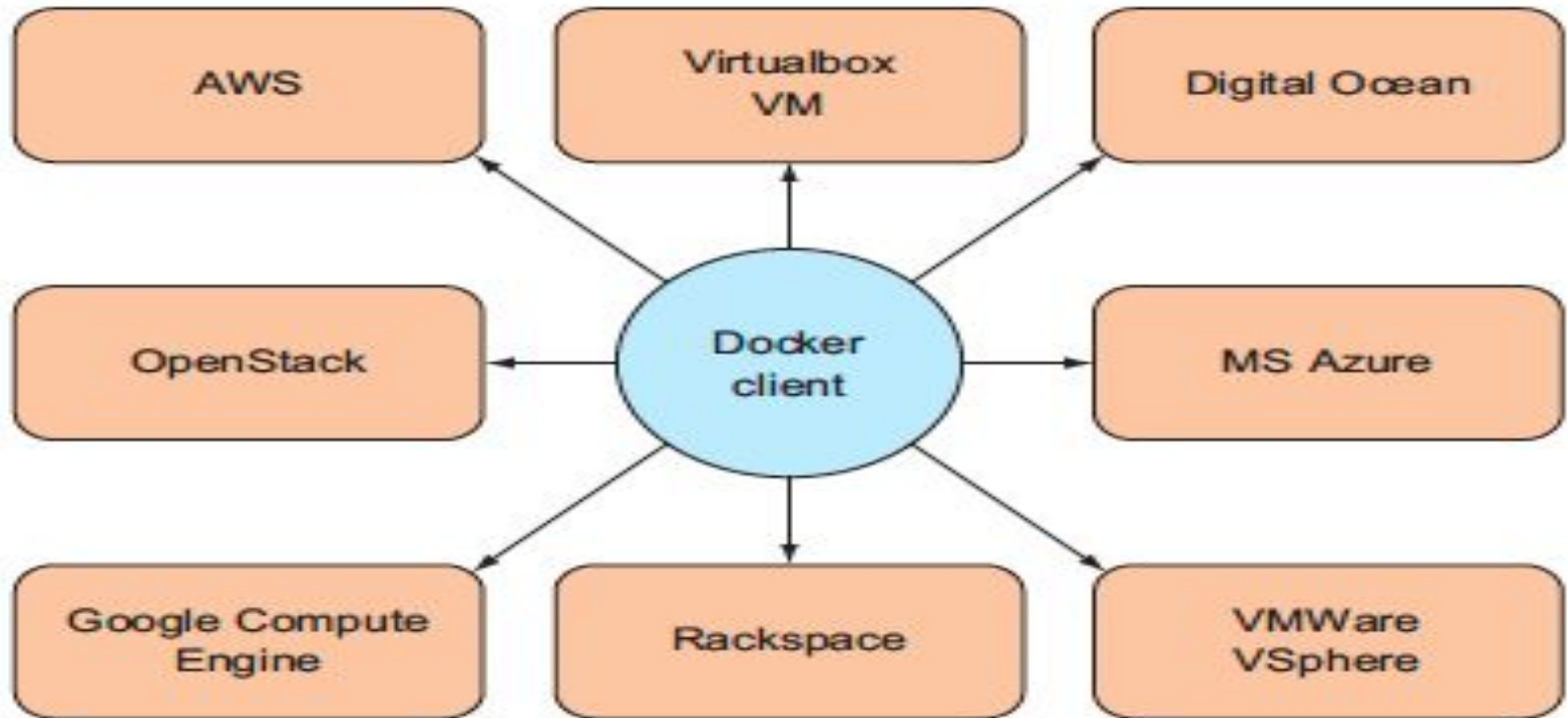
Ship more software

Standardized yet flexible environment

Understanding Docker Machine

- ❖ Docker Machine allows the management of external Docker hosts
 - client for external Docker hosts which can be standalone, virtual machines, or cloud engines
 - It allows the Docker client and daemon to be configured, controlled, or updated
- ❖ Docker was available only as a native application for Linux prior to v1.12
 - It was available only for Windows and Mac as a virtual machine
 - Docker Machine was the only way to run Docker in a virtual machine

Understanding Docker Machine

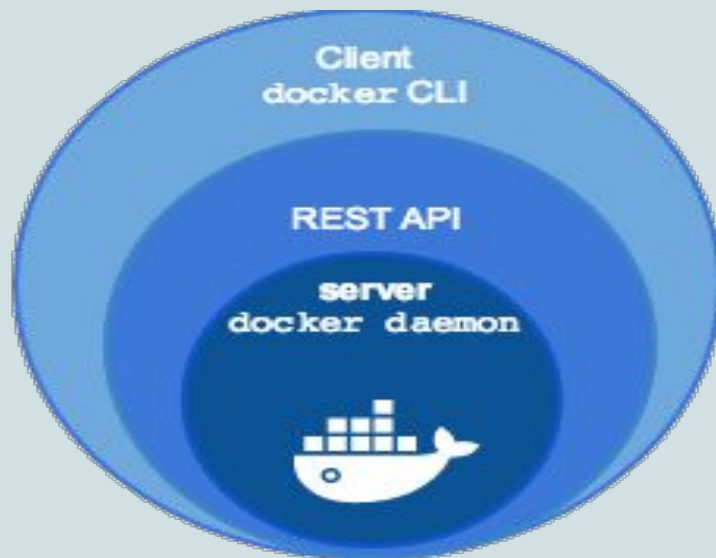


Understanding Docker Machine

- ❖ There are two use cases for using Docker Machine:
 - provision multiple remote Docker hosts
 - run Docker on older Windows and Mac machines which are not supported by the native versions of Docker Engine
- ❖ Docker Toolbox is the virtual machine environment for older Windows and Mac machines
 - It requires Docker Machine to issue docker commands

Understanding Docker Machine

- ❖ Docker Engine is usually referred to as, Docker
 - It consists of the Docker daemon, a RESTful API, and the command line client



Managing containers and instances that run containers

- ❖ we need an orchestration tool to schedule and manage our containers
- ❖ container provides the platform for a software package to run without a full operating system to support it
- ❖ Containers, need just parts of an operating system to function
- ❖ VMs can also run applications in isolation, containers suffer the loss of functioning independently
- ❖ Since containers lack an operating system or a hypervisor, they need their own management system

Managing containers and instances that run containers

-
- ❖ Docker developed Swarm and now builds it into the Docker Toolbox
 - ❖ The more mature container manager is Google's Kubernetes

Managing containers and instances that run containers

The Swarm Factor

- ❖ Docker Swarm - clustering tool for Docker
- ❖ It forms a swarm cluster using one or more Docker hosts
- ❖ Swarm is designed to pack containers onto a host, so it can save other host resources for bigger containers
- ❖ This clustering produces an advantageous economy of scale compared to randomly scheduling a container to a host in the cluster

Managing containers and instances that run containers

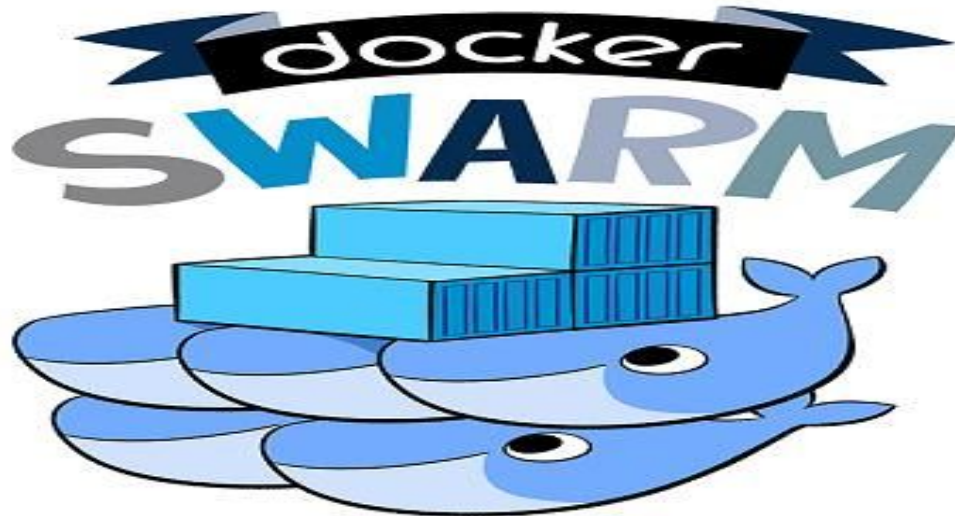
- ❖ Swarm's key factors that differentiate it from Kubernetes:
 - native clustering capabilities to turn a group of Docker engines into a single, virtual Docker Engine
 - swarm contains only two components:
 - Agents
 - managers
 - One cluster has a host that runs a Swarm agent
 - Another host runs a Swarm manager

Managing containers and instances that run containers

- ❖ essential to the operation because the manager segment orchestrates and schedules containers on the hosts
- ❖ Swarm uses a discovery service to find and add new hosts to the cluster
- ❖ Swarm provides the standard Docker API

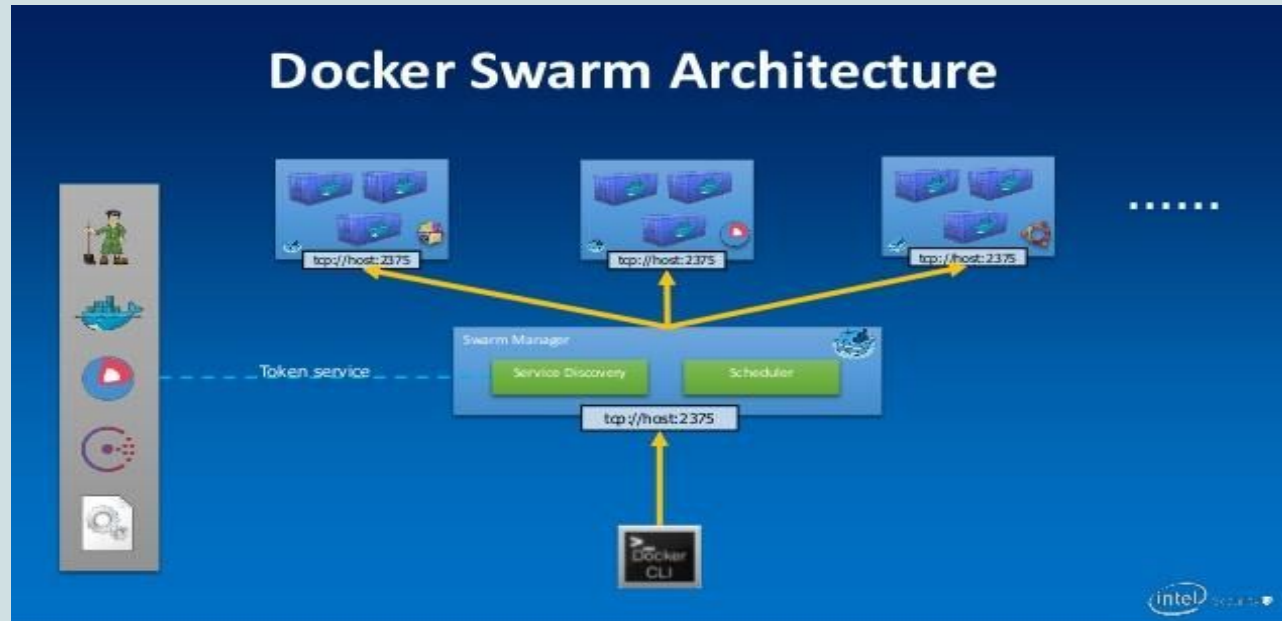
Understanding Docker Swarm

- ❖ Docker Swarm is clustering for Docker
- ❖ It turns several Docker hosts into a single virtual Docker host
- ❖ The regular Docker client works with Swarm



Understanding Docker Swarm

- ❖ The Swarm is controlled by a Swarm Manager
- ❖ Each Docker node communicates with the manager



Understanding Docker Swarm

- ❖ A machine needs to be designated as a manager
- ❖ There can be several managers
- ❖ The manager is created by initializing a Swarm
- ❖ Swarm must be performed on the manager machine
 - `docker swarm init --advertise-addr 192.168.0.38`

Swarm initialized: current node (5lo6zmzvashepfm8ipnlni37) is now a manager

Understanding Docker Swarm

❖ To add a worker to this swarm, run the following command:

➤ `docker swarm join --token`

`SWMTKN-1-51icto7l3hfkym98e2cq0rflh6a6hkyqzpxyb8jaid3qzx5kmm-864t0x9ebw4a2yms`
`ms1kvq9s9 192.168.0.38:2377`

➤ To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions

Understanding Docker Swarm

- ❖ The manager node is automatically added to the Swarm
- ❖ Information about the Swarm can be found using `docker info`

...

Swarm: active

NodeID: 5lo6zmzvashexpfm8ipnlni37

Is Manager: true

ClusterID: e9ff1sv78oxyb1989xa0hvy77

Managers: 1

Nodes: 1

...

Node Address: 192.168.0.38

...

Understanding Docker Swarm

- ❖ Nodes can be added to the Swarm
- ❖ Ensure that firewall rules aren't blocking port 2377 on the manager
- ❖ The nodes can be listed

➤ `docker swarm join --token`

`SWMTKN-1-51icto7l3hfkym98e2cq0rflh6a6hkyqzpxpyb8jaid3qzx5kmm-864t0x9e`

`bw4a2ymsms1kvq9s9 192.168.0.38:2377`

➤ `docker node ls`

Understanding Docker Swarm

Docker Swarm services

- ❖ A Docker Service is a container
- ❖ Services run in Docker Swarm
- ❖ They can only be started on a Swarm Manager node
- ❖ Multiple copies of services can be run
- ❖ The Swarm Manager replicates the service on other nodes in the Swarm

Understanding Docker Swarm

- ❖ A service can be added to the manager node
- ❖ A service is a container
- ❖ It also appears as a running container
 - `docker service create --replicas 1 --name helloworld alpine ping docker.com`
 - `docker service ls`
 - `docker service inspect --pretty helloworld`
 - `docker ps`

Understanding Docker Swarm

- ❖ Docker Swarm Services - Scaling
- ❖ A service can be scaled
 - The service will be duplicated and run on different nodes
 - The service can be removed from all nodes
 - `docker service ps helloworld`
 - `docker service scale helloworld=2`
 - `docker service ps helloworld`
 - `docker service rm helloworld`

More Hands on Swarm

- ❖ Assumption: A swarm is present
- ❖ Create a Service
 - `docker service create --replicas 5 -p 80:80 --name web nginx`
 - `docker service ls`
 - `docker service ps web`
 - `docker ps`

More Hands on Swarm

- ❖ Try out a curl to any of the Docker Machine IPs or hit the URL (<http://<machine-ip>>) in the browser
 - `curl 192.168.10.40:80`
- ❖ You should be able to get the standard NGINX Home page
- ❖ put the Docker Swarm service behind a Load Balancer - Scaling up and Scaling down
 - `docker service scale web=5`
 - `docker service ls`
 - `docker service ps web`

More Hands on Swarm

❖ Inspecting nodes

- `docker node inspect self`
- `docker node inspect worker1`

❖ Draining a node

- `docker node ls`
- `docker service ps web`
- `docker node ps worker1`
- `docker node inspect worker1`

More Hands on Swarm

❖ Draining a node

- `docker node inspect - pretty worker1`
- `docker node update --availability drain worker1`
- `docker service ps web`
- `docker node update --help`
- `docker node update --availability active worker1`

More Hands on Swarm

- ❖ Rebalance the swarm
 - `docker service update web --force`
- ❖ Remove the Service
 - `docker service rm web`
 - `docker service ls`
 - `docker service inspect web`

Consolidated Commands

- `docker login`
- `docker --version`
- `docker info`
- `docker build -t friendlyhello .`
- `docker image ls`
- `docker run helloworld`
- `docker container ls --all`
- `docker ps`
- `docker ps -a`
- `docker start <container>`

- ❖ Building the app:
 - `docker build -t friendlyhello .`
 - `docker image ls`
 - `docker run -p 4000:80 friendlyhello`

Questions?

References



- ❖ <https://docs.docker.com/engine/reference/commandline/cli/#configuration-files>
- ❖ <https://hub.docker.com>
- ❖ <http://www.floydhilton.com/docker/2017/03/31/Docker-ContainerHost-vs-ContainerOS-Linux-Windows.html>
- ❖ <https://rominirani.com/docker-swarm-tutorial-b67470cf8872>
- ❖ <https://realguess.net/2014/12/31/mount-multiple-data-volumes-and-multiple-data-volume-containers-in-docker/>
- ❖ <https://containerjournal.com/2017/02/24/monolithic-apps-can-run-docker/>

Suggestion & comments?



Thank you for being an
awesome Audience!

