Docker Containerization

Agenda

- Overview on Docker
- Working with Containers - In depth
- Docker Volumes
- Networking & Port Forwarding
- Creating your own images
- Pushing Images to Docker Registry
- Configuring Web Application with Docker
- CI/CD with Docker & Bamboo

Agenda

- CI/CD with Docker & Bamboo
- Docker Compose
- Docker Machine
- Troubleshooting docker containers
- Docker Swarm
- Overview on Kubernetes

What is Docker

Docker is a open-source platform for Building, Shipping and Running applications using container virtualization technology.

The docker platform consists of multiple products/tools:

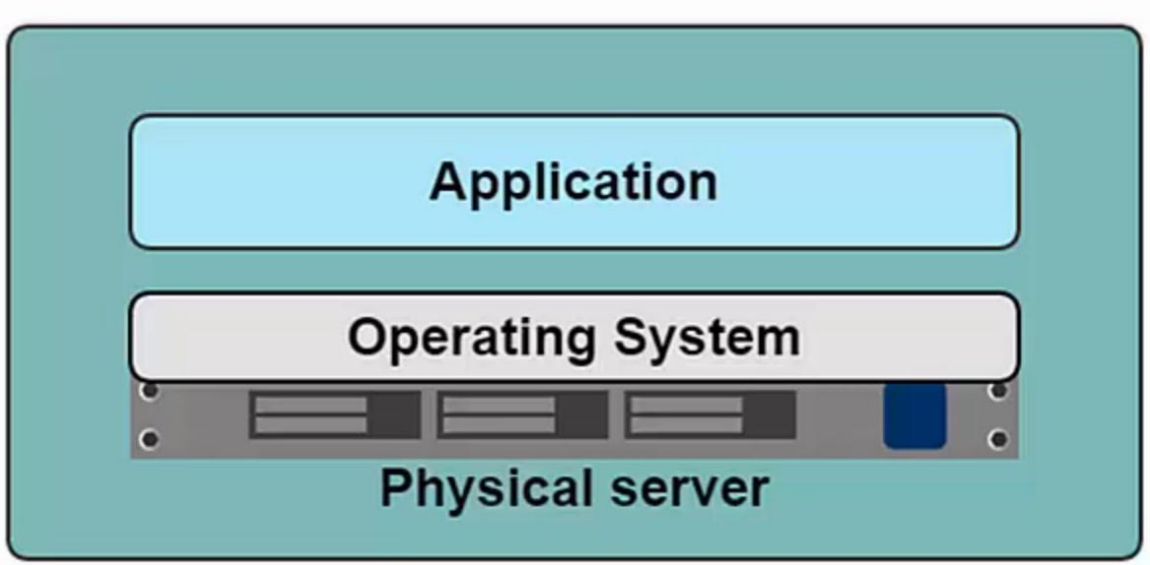
- Docker Engine
- Docker Hub
- Docker Machine
- Docker Swarm
- Docker Compose

Why would you use docker

- Conventional Deployment takes longer time
- Infrastructure development takes time
- Application portability is a challenge (it works on my machine)
- Manual deployment scripts are difficult to manage and version control.

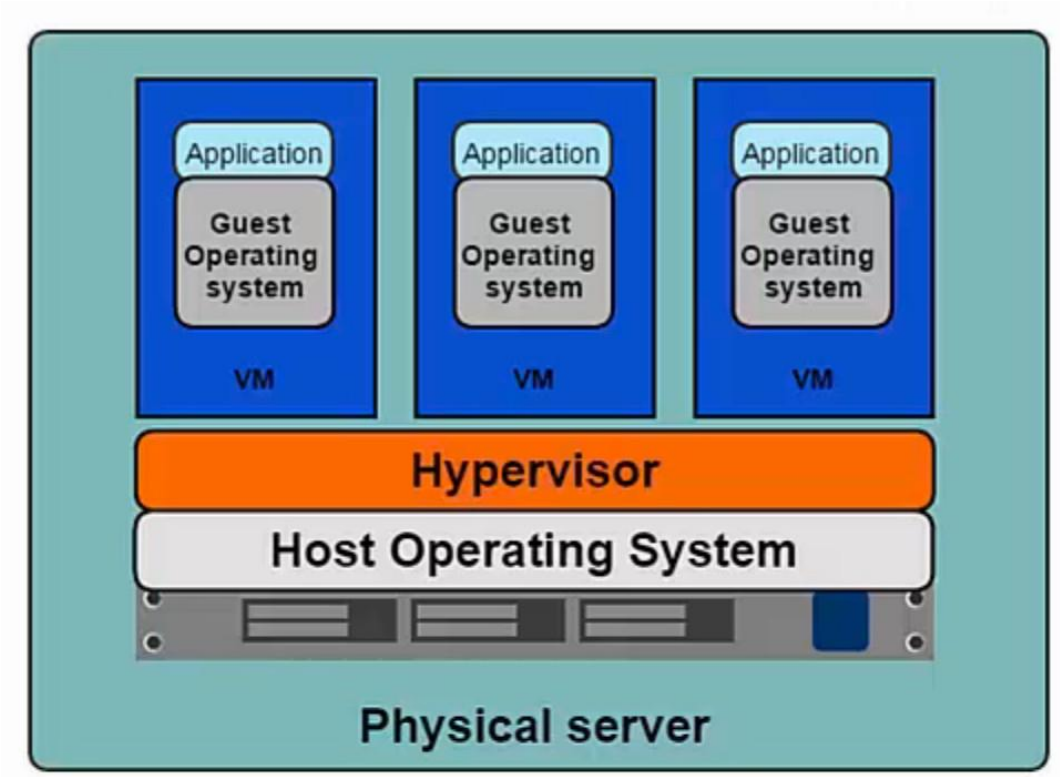
History

- Slow deployment times
- Huge costs
- Wasted resources
- Difficult to scale
- Vendor lock in



Hypervisor based Virtualization

- One physical server can contain multiple applications
- Each application runs in a virtual machine



Benefits of VM

- Better resource pooling
  - One physical machine divided into number of VMs
- Easier to scale
- VM's in the cloud
  - Rapid elasticity
  - Pay as you go model

Limitations of VM

- Each VM still requires
  - CPU Allocation
  - Storage
  - RAM
  - An entire guest operation system

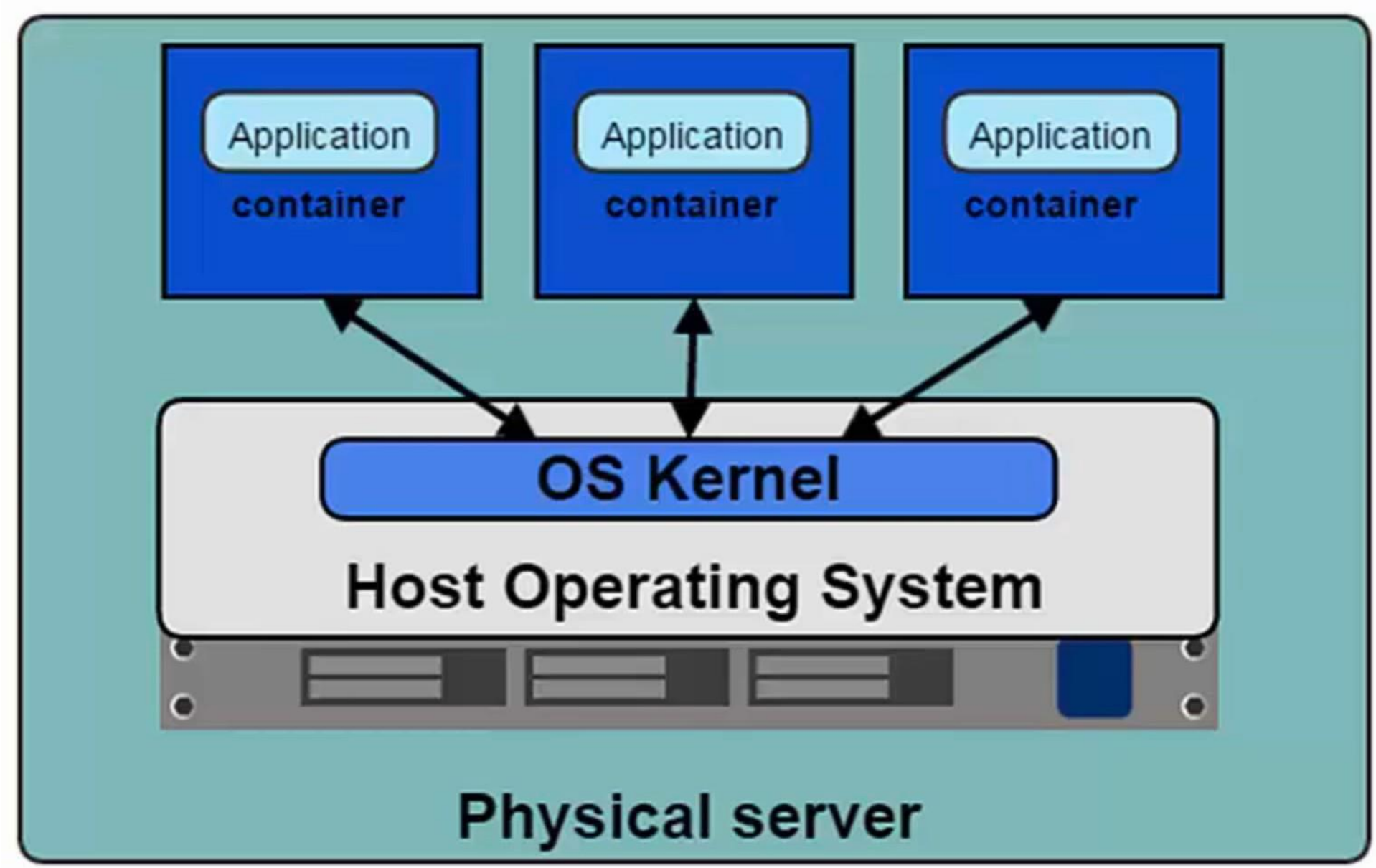
- The more VMs you run, the more resources you need
- Guest OS means wasted resources
- Application portability not guaranteed

Introduction to Containers

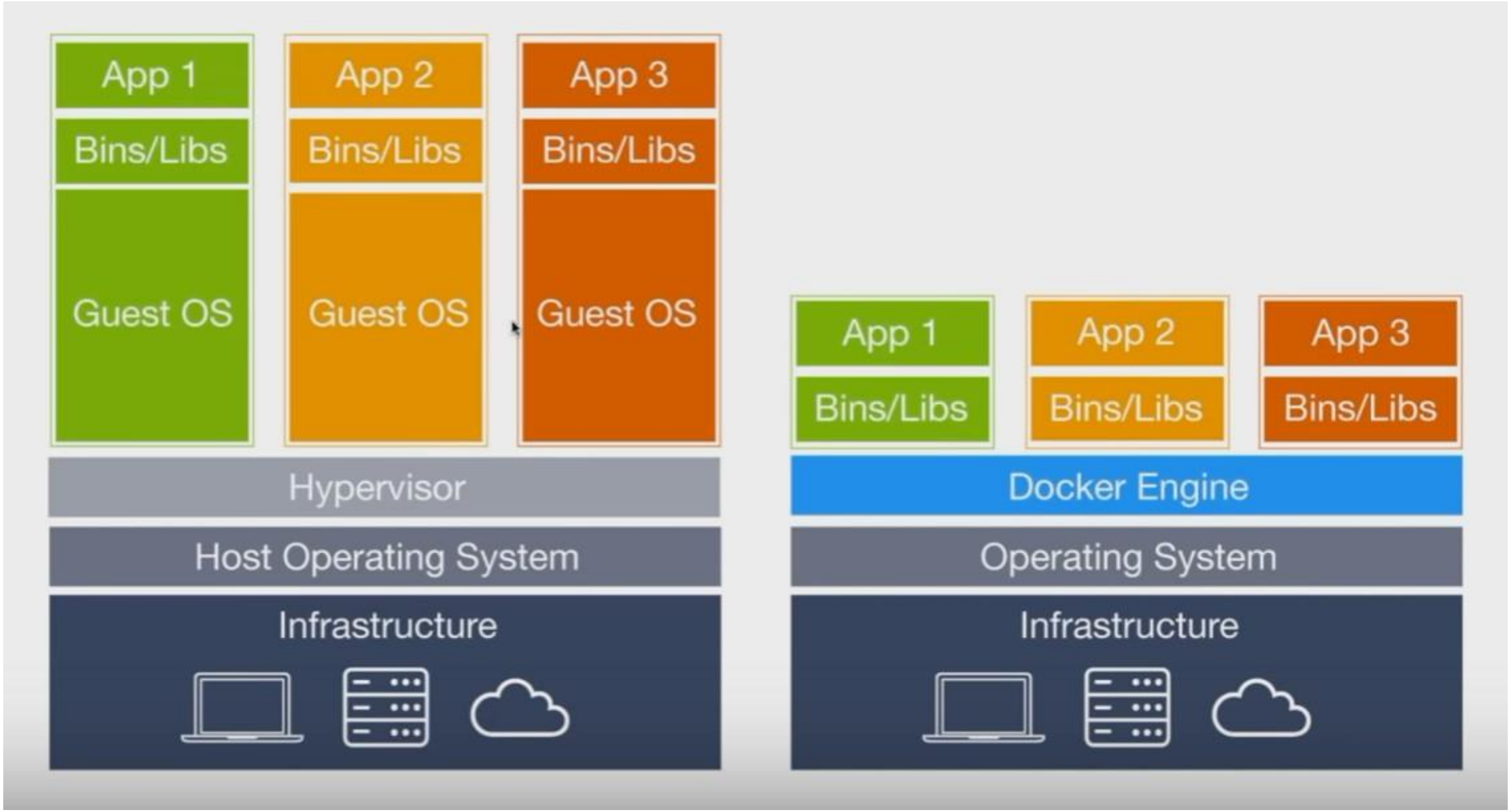
Container based virtualization uses the kernel on the host's operating system to run multiple guest instances

- Each guest instance is called a container
- Each container has its own
  - Root file system
  - Processes
  - Memory
  - Devices
  - Network ports

Introduction to Containers



VM Vs Containers



Underlying Technology

- Namespace to provide isolation
- Control groups to share/limit hardware resources
- Union file system makes it light and fast
- libcontainer defines container format

Advantages of Docker

- Faster Deployments
- Isolation
- Portability - ‘it works on my machine’
- Limit resource usage
- Sharing

Install Docker

Follow the instructions at https://docs.docker.com/installation to install the latest Docker maintained Docker package on your preferred operating system

The installation provides

- Docker Engine
- Docker CLI client
- Docker Compose
- Docker Machine.

Install Docker on CentOS

- Create a fresh VM with CentOS7
- Initial Setup
  - ssh root@IPAddress
  - adduser demo - To add a new user
  - passwd demo - To create password for demo user
  - gpasswd -a demo wheel - Add user to wheel group to allow sudo privileges
- On local (host) machine
  - create ssh key pari - ssh-keygen
  - ssh-copy-id demo@IP address - To copy public key to remote centos machine
  - check with ssh demo@IPaddress

Install Docker on CentOS

- Install Docker
  - On CentOS machine - sudo demo
  - sudo yum check-update
  - curl -fsSL https://get.docker.com/ | sh
  - After installation has completed, start docker daemon: sudo systemctl start docker
  - Verify that its running: sudo systemctl status docker
  - To make sure it starts on every reboot: sudo systemctl enable docker
  - To execute commands without sudo, add demo user to docker group: sudo usermod -aG docker demo

Install Docker-Machine on CentOS

```
curl  -L https://github.com/docker/machine/releases/download/v0.12.2/docker-machine-`uname  -s`-`uname -m` >/
tmp/docker-machine &&
chmod +x  /tmp/docker-machine &&
sudo cp  /tmp/docker-machine /usr/local/bin/docker-machine
```

Install Docker-Compose on CentOS

1. sudo yum install epel-release
2. sudo yum install -y python-pip
3. sudo pip install docker-compose
4. sudo yum upgrade python*

Commands to Verify the Installation

- docker version
- docker-machine version
- docker-compose version

Check the Installation

Run the below commands to verify the installation

- **docker version**

```
Client:
 Version:      17.03.1-ce-rc1
 API version: 1.27
 Go version:  go1.7.5
 Git commit:   3476dbf
 Built:        Fri Mar 17 00:27:41 2017
 OS/Arch:      darwin/amd64

Server:
 Version:      17.03.1-ce-rc1
 API version:  1.27 (minimum version
1.12)
 Go version:   go1.7.5
 Git commit:   3476dbf
 Built:        Wed Mar 15 20:28:18 2017
 OS/Arch:      linux/amd64
 Experimental: true
```

- **docker-machine version**
  - docker-machine version 0.10.0, build 76ed2a6
- **docker-compose version**
  - docker-compose version 1.11.2, build dfed245
  - docker-py version: 2.1.0
  - CPython version: 2.7.12
  - OpenSSL version: OpenSSL 1.0.2j 26 Sep 2016

Docker Components

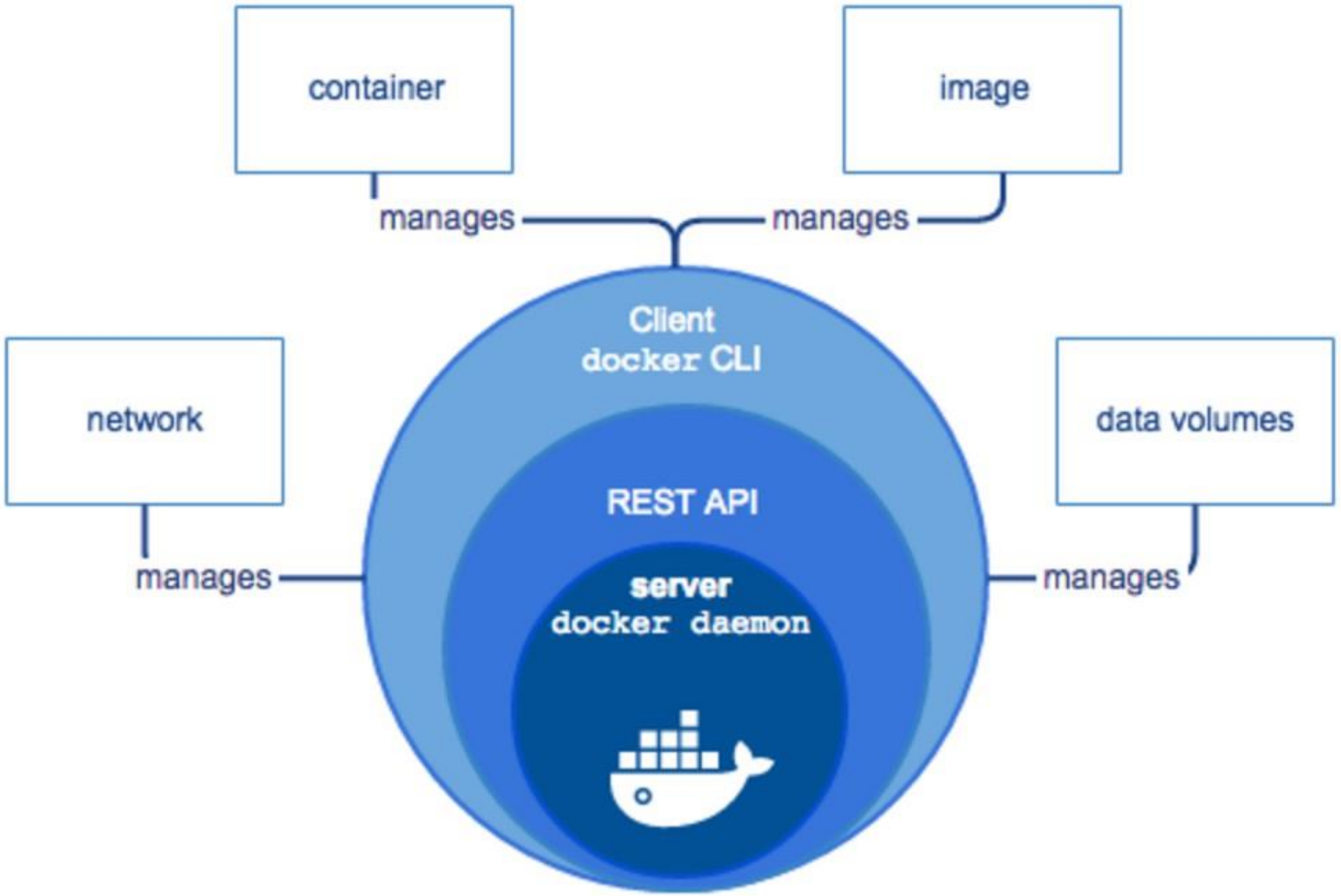
- Docker Engine
- Images
- Containers
- Registry
- Repository
- Docker Hub
- Docker Orchestration tools

Docker Engine

Docker Engine is a client-server application with these major components:

- A server which is a type of long-running program called a daemon process.
- A REST API which specifies interfaces that programs can use to talk to the daemon and instruct it what to do.
- A command line interface (CLI) client.

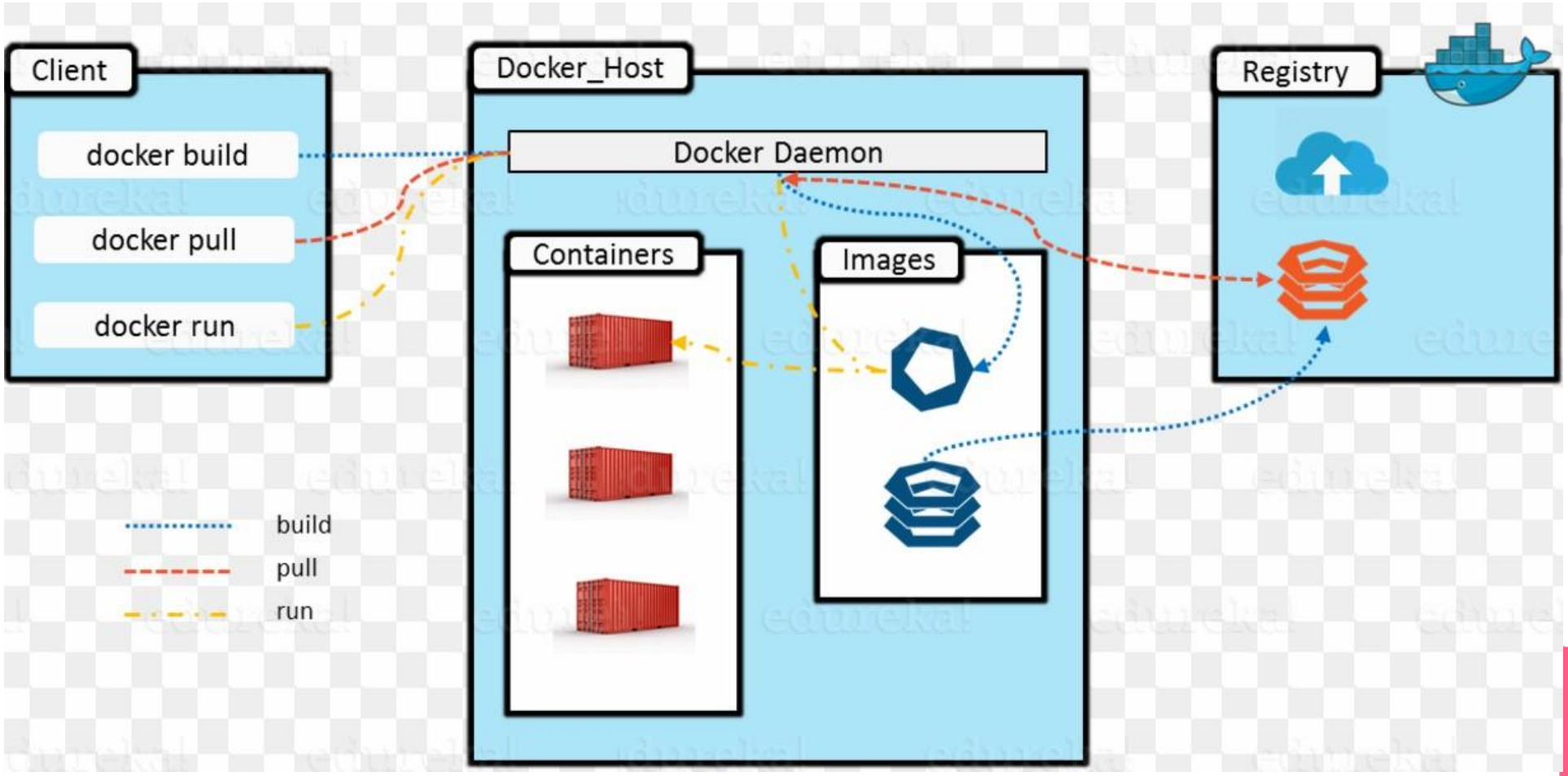
Docker Engine



Client and Server (Daemon)

- Client / Server architecture
- Client takes user inputs and send them to the daemon
- Daemon builds, runs and distributes containers
- Client and daemon can run on the same host or on different hosts
- CLI client and GUI (Kitematic)

Docker Architecture



Container & Images

Images

- Read only template used to create containers
- Built by you or other docker users
- Stored in the Docker Hub or your local registry

Containers

- Runnable instance of a docker image
- Isolated application platform
- Contains everything needed to run your application
- Based on one or more images

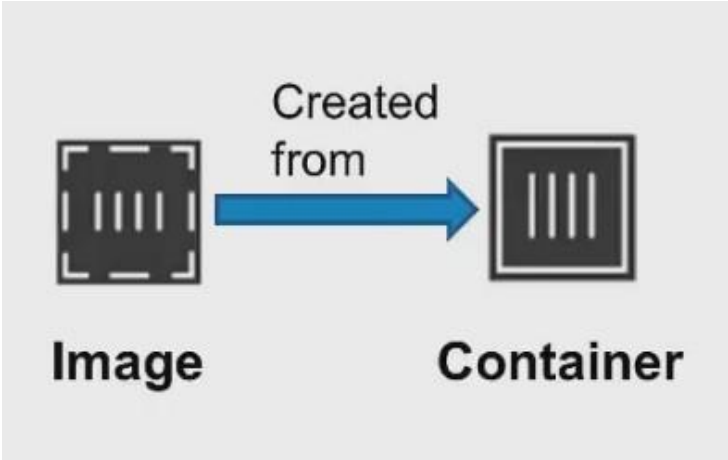
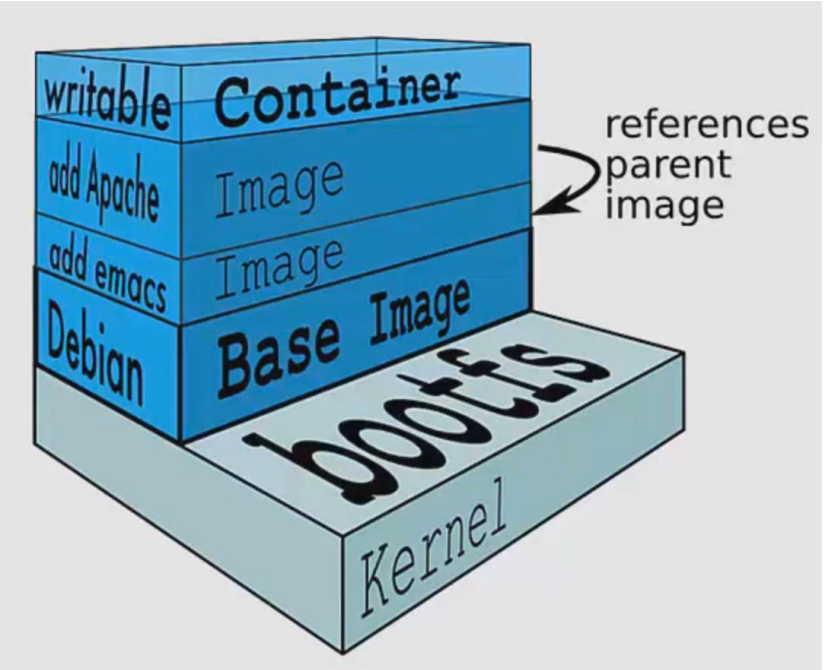


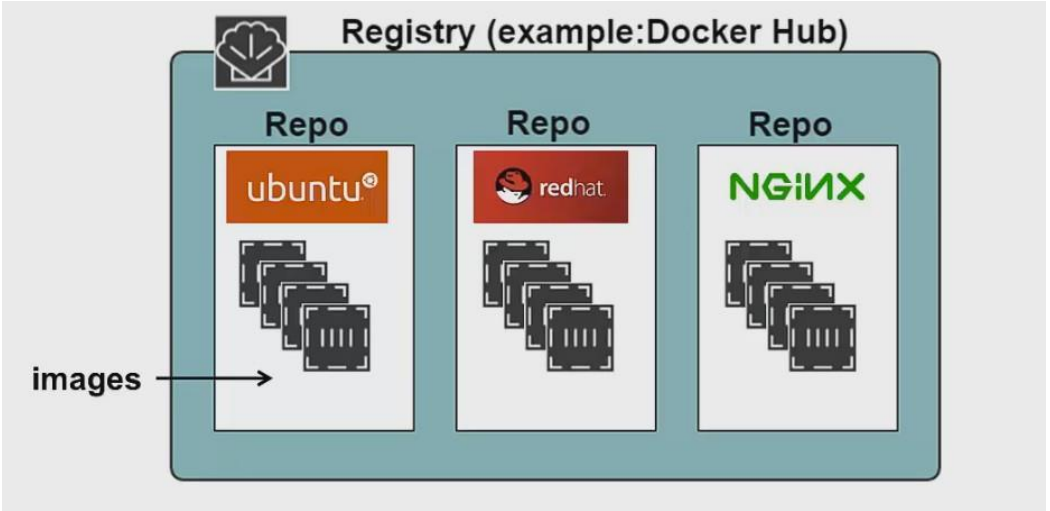
Image Layers

- Images are comprised of multiple layers
- Every image contains a base layer
- A layer is also just another image
- Docker uses a copy on write system
- Layers are read-only



Docker Registry

- A docker registry is a library of images. A registry can be public or private, and can be on the same server as the Docker daemon or Docker client, or on a totally separate server.
- Docker Hub is a public registry that contains a large number of images available for your use.



Docker Orchestration

Three tools for orchestrating distributed applications with Docker

- **Docker Machine**
  - Tool that provisions Docker hosts and installs the Docker Engine on them
- **Docker Swarm**
  - Tool that clusters many Engines and schedules containers
- **Docker Compose**
  - Tools to create and manage multi-container applications

Hands-On

1. Create your first container.
2. Deploying web server in a container
3. Deploying customized java application in a container
4. Interactive mode and opening bash terminal
5. Detached Mode
6. Creating containers with Volumes
7. Exposing Containers with Port Redirect
8. Build your own images - Dockerfile
9. Store images on docker hub & private registry

Create your first container

- Run the ‘hello-world’ container to test the installation
  - **sudo docker run hello-world**

Exercise

- Run command: docker run hello-world
  - check the output
  - docker images
  - docker ps
  - docker ps -a

Interactive Mode & Getting Terminal Access

- Use *docker run* command
  - Syntax: *docker run [options] [images] [command] [args]*
- Image is specified with repository:tag

```
docker run centos:7
```

Interactive Mode & Getting Terminal Access

- Use -i and -t flags with docker run
- The -i flag tells docker to connect to STDIN on the container
- The -t flag specifies to get a pseudo-terminal
- Note: You need to run a terminal process as your command (e.g. /bin/bash)

docker run -i -t centos /bin/bash

What happens when you run the Container?

When you run this command, Docker Engine does the following:

- **Pulls the centos image with tag 7**
- **Creates a new container:** Docker uses the image to create a container.
- **Allocates a filesystem and mounts a read-write layer:** The container is created in the file system and a read-write layer is added to the image.
- **Allocates a network / bridge interface:** Creates a network interface that allows the Docker container to talk to the local host.
- **Sets up an IP address:** Finds and attaches an available IP address from a pool.
- Opens up a Bash terminal

Exercise

- Run command: docker run centos:7
  - docker images
  - docker ps
  - docker ps -a
- Run in Interactive mode: docker run -it cents:7 /bin/bash
  - You will be placed inside the container.
  - Open another terminal and run following
    - docker ps - You will see a running container

Start and Stop Container

- docker run -it centos:7
- sudo yum update
- yum install git
- check git version - You will see git installed
- exit container
- Again: docker run -it centos:7
- You will not see GIT installed, because a new container has been pinned up
- docker ps -a - And check the container ID
- docker start <containerID>
  - docker exec -it <containerID> /bin/bash
- docker stop <containerID>

Inside Docker

- docker inspect <containerID>
- docker logs <containerID>
- docker logs —follow

Exercise

- docker run -it centos:7
  - sudo yum update
  - yum install git
  - check git version - You will see git installed
- exit container
- Again: docker run -it centos:7
  - You will not see GIT installed, because a new container has been spinned up
- docker ps -a - And check the container ID
- docker start <containerID>
  - docker exec -it <containerID> /bin/bash
- docker stop <containerID>

Starting a Web Server in a Container

- docker run nginx

Running in Detached Mode

- Also known as running in background or as a daemon
- Use -d flag
- To observer output use docker logs [container ID]
- docker logs -f [containerID]

docker run -d nginx

Container Networking

- Typically, a Docker host comprises multiple Docker containers and hence the networking has become a crucial component for realizing composite containerized applications. Docker containers also need to interact and collaborate with local as well as remote ones to come out with distributed applications. The
- `bridge` network is the default network interface that Docker Engine assigns to a container if the network is not configured using the `--net` option of the `docker run` subcommand.

docker network ls

docker network inspect

Networking - Exposing Containers with Port Redirect

- -P flag to map container ports to host ports, and assigns port automatically
- -p - To assign specific port, use -p

docker run -P nginx
docker run-p <myport>:<containerport> nginx

Exec Command

- Used to run a command in already existing container
- docker exec -i -t <contianerID> /bin/bash

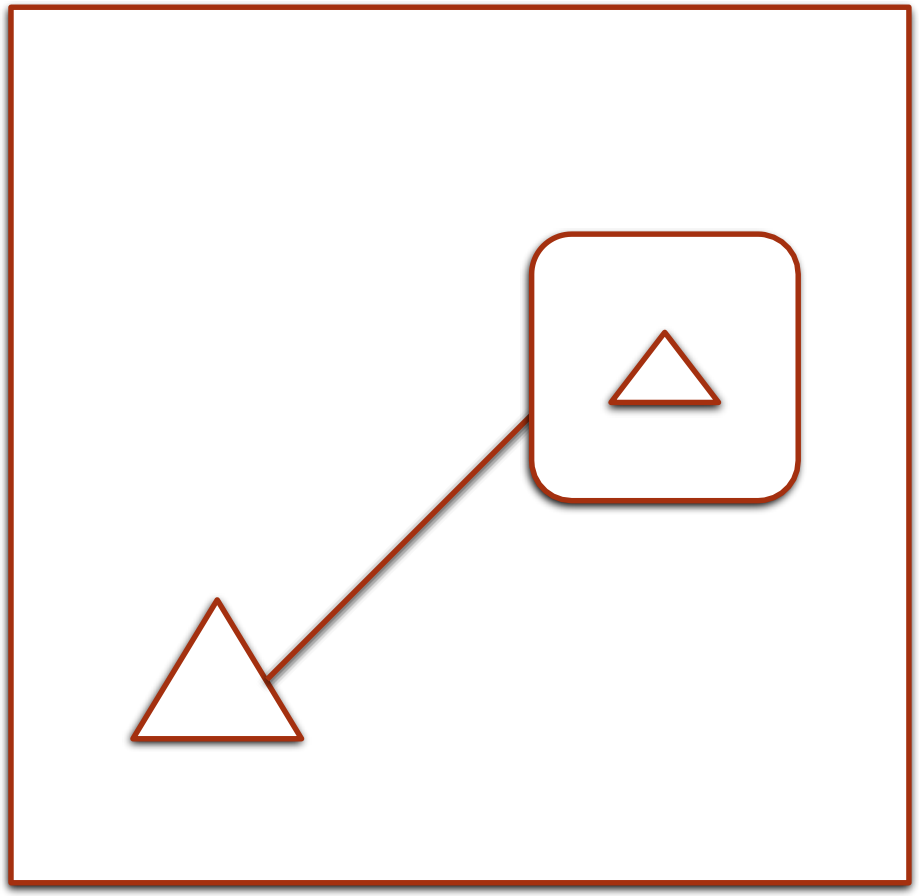
Exercise

- docker run nginx
- docker run -d nginx
- docker run -d -P nginx
- docker run -d -p 8081:80 nginx
- docker run -d -p 8080:8080 tomcat:7
  - Now open the browser on your machine and type localhost:8080. It should display nginx page

Docker Volumes

- Docker manages data within the docker container using Docker Volumes.
- For e.g. let's say that you are running an application that is generating data and it creates files or writes to a database and so on. Now, even if the container is removed and in the future you launch another container, you would like that data to still be there
- Until now, all the files that we created in an image or a container are part and parcel of the Union filesystem. However, the data volume is part of the Docker host filesystem, and it simply gets mounted inside the container.

Docker Volumes



Docker Volumes

- It is initialized when the container is created. By default, it is not deleted when the container is stopped.
- Data volumes are designed to persist data, independent of the container's lifecycle. Docker therefore *never* automatically deletes volumes when you remove a container, nor will it "garbage collect" volumes that are no longer referenced by a container.
- Data volumes can be shared across containers too, and can be mounted in read-only mode also.

Docker Volumes

- docker run -it -v data:/myvol tomcat:7
- docker run -d -v ~/practice/shared:/myvol tomcat:7
- docker inspect - Checkout mounts

- docker run -it —name master -v backup:/backup -v logs:/logs ubuntu bash
  - exec in this container, and create test files in logs and backup
- docker run -it name slave1 —volumes-from master ubuntu bash docker
- inspect <volumename>

Docker Volumes

docker run -it -v /home/demo/myvol:/myvol —centos1 cents

docker run -it -v /myvol2 —centos2 busybox (Volume Container)

(cd /var/lib/docker/<containerID>/_data
docker run -it -v —volumes-from centos2 —name centos3 centos

Exercise

Volumes

docker run -it -v data:/myvol tomcat:7

docker run -d -v ~/practice/shared:/myvol tomcat:7

docker volume ls

Volumes-from

- docker run -it —name master -v backup:/backup -v logs:/logs ubuntu bash
    - exec in this container, and create test files in logs and backup
- docker run -it name slave1 —volumes-from master ubuntu bash docker
- inspect <volumename>

Create Your Own Image

Images can be created using two methods

- Docker Commit:
    - docker commit command saves changes in a container as a new image
        - docker commit [ContainerID] [repository:tag]
- Dockerfile

Create Your Own Image

- Docker Commit:
  - docker run -it centos:7 /bin/bash
    - yum update
    - yum install git
    - yum install curl
  - exit container
  - docker commit <containerID> <username><yourreponame>:<tag>
  - docker images
  - Run the image, and verify if git is installed
  - exit
  - docker push <username><yourreponame>:<tag>

Exercise

- Docker Commit:
  - docker run -it centos:7 /bin/bash
    - yum update
    - yum install git
    - yum install curl
  - exit container
  - docker commit <containerID> <repository:tag>
  - docker images
  - Run the image, and verify if git is installed

Dockerfile

- A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image.

- Using “docker build” users can create an automated build that executes several command-line instructions in succession.

- The docker build command builds an image from a Dockerfile and a *context*.

Dockerfile

FROM

- The FROM instruction sets the Base image for subsequent instructions
- A valid dockerfile must have a FROM instruction
- FROM can occur multiple times in the dockerfile

E.x. FROM java:7 or FROM cantos:7

CMD

- CMD defines a default command to execute when a container is created
- CMD performs no action during the build image
- Shell and EXEC form

  Can only be specifed once in a Dockerfile
- Can be overridden at run time

FROM ubuntu
CMD echo "This is a test." | wc -

Dockerfile

RUN

- Executes a command in a new layer on top of the current image and commit the results

COPY

- The COPY instruction copies new files or directories from <src> and adds them to the filesystem of the container at the path <dest>
- COPY is preferred over ADD

Dockerfile

- ENTRYPOINT - This helps us configure the container as an executable. Similar to `CMD`, there can be at max one instruction for `ENTRYPOINT`; if more than one is specified, then only the last one will be honored.

- MAINTAINER - This sets the author for the generated image, `MAINTAINER <name>`

- ADD <src> <dst> - This copies files from the source to the destination:

- WORKDIR <path> - This sets the working directory for the `RUN`, `CMD`, and `ENTRYPOINT` instructions that follow it

Dockerfile

- EXPOSE - This exposes the network ports on the container on which it will listen at runtime

- ENV - This will set the environment variable `<key>` to `<value>`. It will be passed all the future instructions and will persist when a container is run from the resulting image

- VOLUME [“/data”] OR /data - This instruction will create a mount point with the given name and flag it as mounting the external volume

- USER <username>/<UID> - This sets the username for any of the following run instructions

Dockerfile Examples

Example1

FROM busybox:latest

CMD echo Hello World!!

Example2

- FROM centos:7
- RUN yum install -y git
- VOLUME /myvol
- CMD [“git”, “—version”]

Example3

- FROM java:7
- COPY First.java .
- RUN javac First.java
- CMD ["java", "First”]

Exercise

Example1

- FROM ubuntu:14.04
- RUN apt-get update
- RUN apt-get -y install git
- CMD [“git”, “—version”]

Example2

- FROM java:7
- COPY First.java .
- RUN javac First.java
- CMD ["java", "First”]

Pushing image to DockerHUB

- Push our images to docker hub
  - hub.docker.com
  - Create a repository
  - Run command: ‘docker push repo:tag’.
    - Repo name must be same as name of repo created on docker hub

Docker Private Repository

- docker run -d -p 5000:5000 --restart=always --name registry registry:2

- docker pull centos:7 && docker tag centos:7 localhost:5000/centos:7
- Change the docker push https connection to http
    - Edit the file "/usr/lib/systemd/system/docker.service" and change the parameter
        - ExecStart=/usr/bin/dockerd … to ExecStart=/usr/bin/dockerd –insecure-registry docker-repo.example.com:5000
        - systemctl daemon-reload
        - systemctl restart docker

- docker push localhost:5000/centos:7

- docker images - To list down the images you uploaded to private registry

- docker pull localhost:5000/centos:7

Exercise

Repeat steps of the previous slide

Automated Builds with Docker

CI & CD Automation using Docker can be achieved in Two ways

1. Through Dockerhub Automated Builds

2. Through Jenkins/Bamboo CI server

Automation through dockerHub

Steps

1. Create an application
2. Create Dockerfile for this application to be built. Dockerfile will compile, build, test and package as required.
3. Create an automated build on Dockerhub (Assuming account already created)
4. Push the code on GITHUB.
5. This will run the build on dockerhub automatically and create an image
6. This image can be pulled to QA or any other server.

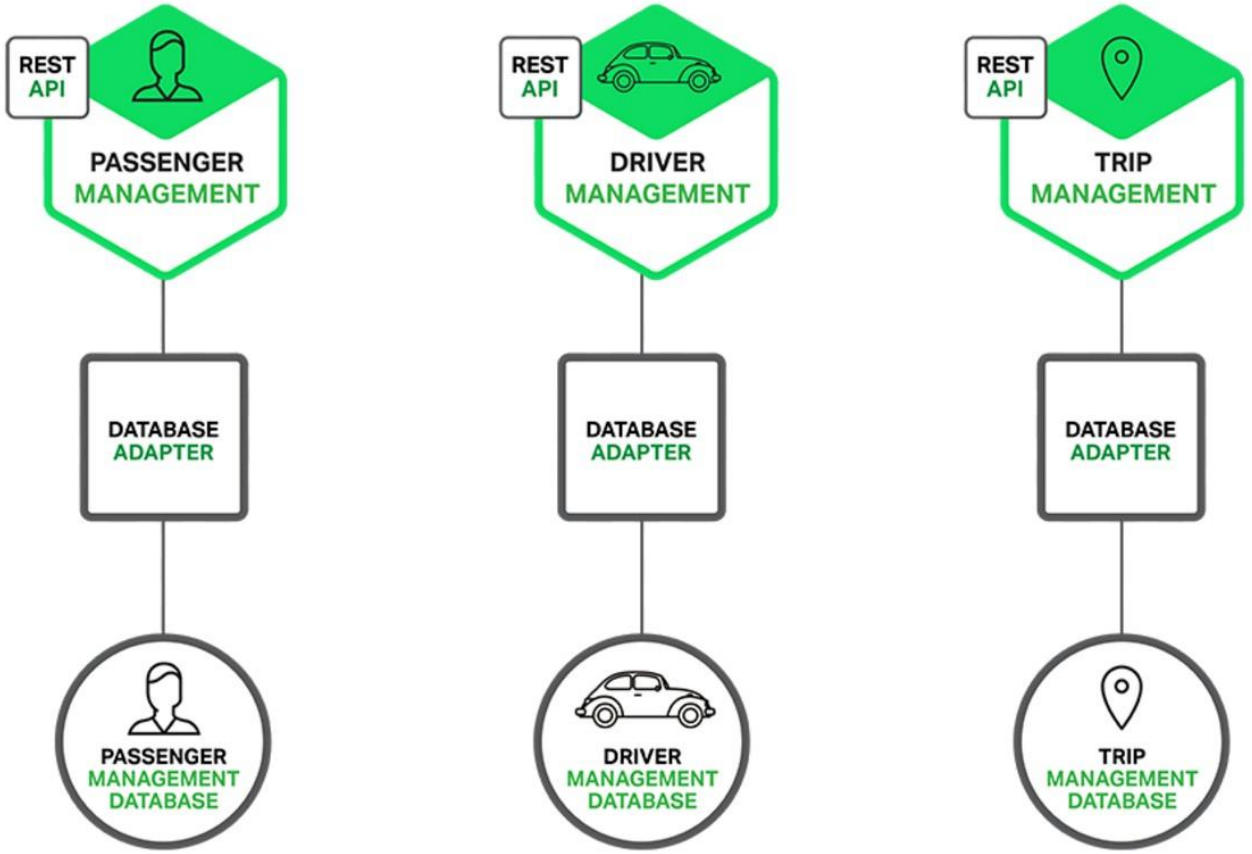
Docker Orchestration

Three tools for orchestrating distributed applications with Docker

- **Docker Compose**
  - Tools to create and manage multi-container applications
- **Docker Machine**
  - Tool that provisions Docker hosts and installs the Docker Engine on them
- **Docker Swarm**
  - Tool that clusters many Engines and schedules containers

Microservices Architecture Vs Monolithic Architecture

Microservices is an architectural style that structures an application as a collection of loosely coupled services, which implement business capabilities.



Docker Compose

- Docker Compose is a tool for defining and running multi-container Docker applications.
- With Compose, you use a Compose file to configure your application's services
- Then, using a single command, you create and start all the services from your configuration.

Docker Compose

Using Compose is basically a **three-step process.**

1. Define your app's environment with a Dockerfile so it can be reproduced anywhere.
2. Define the services that make up your app in docker-compose.yml so they can be run together in an isolated environment.
3. Lastly, run docker-compose up and Compose will start and run your entire app.

docker-compose.yml

```
version: '2'
services:
  db:
    image: postgres
  web:
    build: .
    command: bundle exec rails s -p 3000 -b '0.0.0.0'
    volumes:
      - .:/myapp
    ports:
      - "3000:3000"
    depends_on:
      - db
```

HandsOn - Sample Python Example

- create a new directory - compose(anyname)
- create new file docker-compose.yml
- create a directory myvol and file1 inside it
- vi docker-compose.yml
- (write the steps as mentioned)

- docker-compose up

HandsOn - Sample Python Example

```
version: '2'
services:
  web:
    build: .
    ports:
     - "5000:5000"
    volumes:
     - .:/code
  redis:
    image: "redis:alpine"
```

HandsOn - Sample Python Example

```
version: "3"
services:
  web:
    image: tomcat:7
    ports:
     - "8080:8080"
    volumes:
     - ./myvol:/myvol
```

Troubleshooting - Debug Commands

- Docker details:
  - docker info
  - docker version

- Service and Container Debugging
    - docker logs <containername/id>
    - docker inspect <containername/id>
    - docker service logs <servicename/id>
    - docker service inspect <servicename/id>

- Network debugging
    - docker network inspect <networkname/id>
- Basic Swarm Debugging
    - docker node ls

Docker Daemon Logs

- Ubuntu — /var/log/upstart/docker.log
- Boot2Docker — /var/log/docker.log
- Debian GNU/Linux — /var/log/daemon.log
- CentOS — /var/log/messages | grep docker
- Fedora — journalctl -u docker.service
- Red Hat Enterprise Linux Server —/var/log/messages | grep docker

Troubleshooting Containers

- Troubleshooting Basics
- Command Issues
    - Volumes
    - Networking
    - TLS
- Advanced Troubleshooting techniques

Troubleshooting Basics

- Submitting diagnostics, feedback, and GitHub issues
- Checking the Logs
- Make sure certificates are set up correctly

Submitting Diagnostics, feedback and GITHUB issues

- Diagnose and Feedback on Windows
  - If any issue is encountered, follow the documentation on Docker for Windows issues on GitHub, or the Docker for Windows forum. If the solution is not found, than you can send the diagnosis report to docker. This can be done through Diagnose and Feedback section. You can also create issues on GITHUB for docker team.
- Diagnose and Feedback on Mac
  - Diagnose and upload - This will upload logs to docker team
  - Diagnose only - This will display the logs, which you can copy while creating issue on github
  - To create issue on mac: https://github.com/docker/for-mac/issues, and for windows: https://github.com/docker/for-win/issues

Checking Logs

- The **docker logs** command shows information logged by a running container.

  The **docker service logs** command shows information logged by all containers participating in a service. The information that is logged and the format of the log depends almost entirely on the container's endpoint command.

- In Linux: /var/lib/docker/containers/……
- In Windows; Use the systray menu to view logs:
  - To view Docker for Windows latest log, click on the Diagnose & Feedback menu entry in the systray and then on the Log file link. You see the full can history of logs in your AppData\Local folder.

Troubleshooting Volume Errors

- Error: Unable to remove filesystem
- Permissions errors on data directories for shared volumes
- Volume mounting requires shared drives for Linux containers
- Verify domain user has permissions for shared drives (volumes)

Error: Unable to remove file system

- Some container-based utilities, such as Google cAdvisor, mount Docker system directories, such as /var/lib/docker/, into a container. For instance, the documentation for cadvisor instructs you to run the cadvisor container.

- When you bind-mount /var/lib/docker/, this effectively mounts all resources of all other running containers as filesystems within the container which mounts/var/lib/docker/. When you attempt to remove any of these containers, the removal attempt may fail with an error like the following:

  Error: Unable to remove filesystem for 74bef250361c7817bee19349c93139621b272bc8f654ae112dd4eb9652af9515: remove /var/lib/docker/containers/74bef250361c7817bee19349c93139621b272bc8f654ae112dd4eb9652af9515/shm: Device or resource busy

- To work around this problem, stop the container which bind-mounts/var/lib/docker and try again to remove the other container.

Permission Errors on data directories for shared volumes

- Docker for Windows sets permissions on shared volumes to a default value of 0755 (read, write, executepermissions for user, read and execute for group). If you are working with applications that require permissions different than this default, you will likely get errors similar to the following.
- Data directory (/var/www/html/data) is readable by other users. Please change the permissions to 0755 so that the directory cannot be listed by other users.

Volume Mounting requires shared drives for Linux Conatiners

If you are using mounted volumes and get runtime errors indicating an application file is not found, a volume mount is denied, or a service cannot start (e.g., with Docker Compose), you might need to enable shared drives.Volume mounting requires shared drives for Linux containers (not for Windows containers). Go to -->Settings --> Shared Drives and share the drive that contains the Dockerfile and volume.

Verify Domain User has permission for shared drives (volumes)

Permissions to access shared drives are tied to the username and password you use to set up shared drives. If you run docker commands and tasks under a different username than the one used to set up shared drives, your containers will not have permissions to access the mounted volumes. The volumes will show as empty.

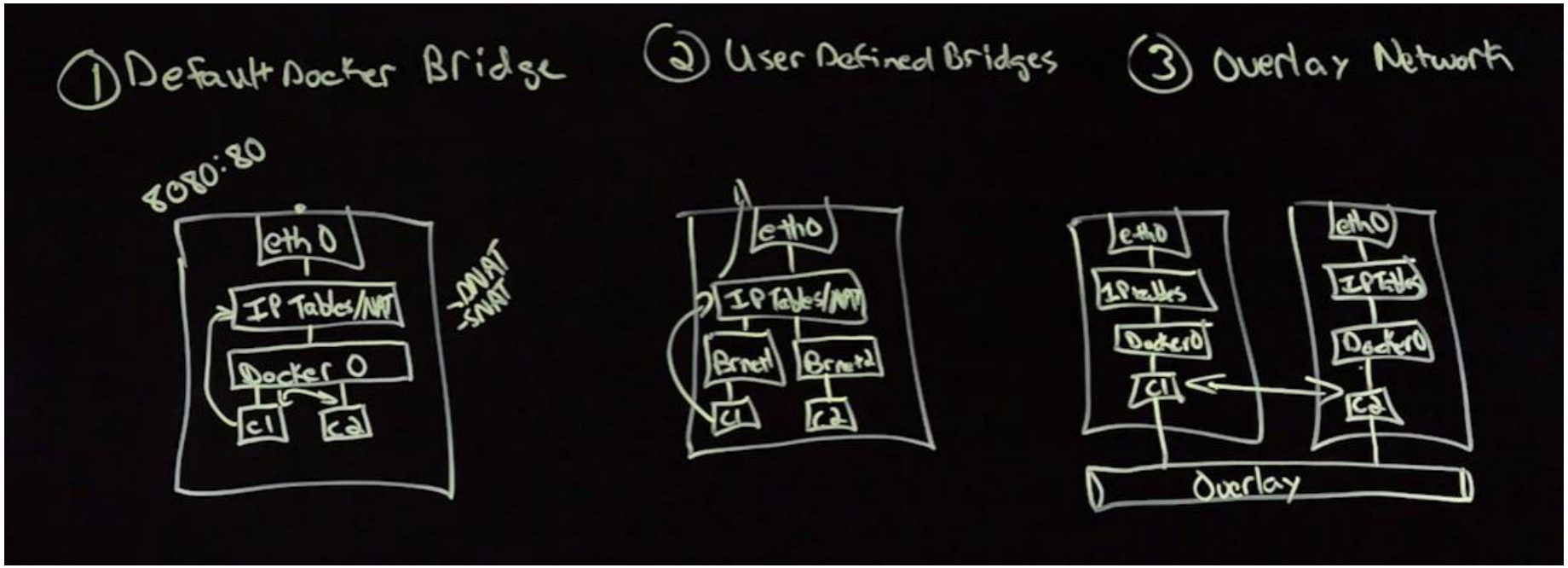
The solution to this is to switch to the domain user account and reset credentials on shared drives.

Docker Networking

Default Docker Bridge

User defined bridges

Overlay network



Troubleshooting - Using Sysdig to Debug

• Sysdig provides application monitoring for containers

From a top level, what sysdig brings to our container management is this:

- Ability to access and review processes (inclusive of internal and external PIDs) in each container
- Ability to drill-down into specific containers
- Ability to easily filter sets of containers for process review and analysis

Sysdig provides data on CPU usage, I/O, logs, networking, performance, security, and system state.

Docker Orchestration

- **Docker Machine**
  - Tool that provisions Docker hosts and installs the Docker Engine on them
- **Docker Swarm**
  - Tool that clusters many Engines and schedules containers
- **Kubernetes**
  - Docker cluster management tool by Google

Docker Machine

Docker Machine is a tool that lets you create a virtual host and install Docker Engine on virtual hosts, and manage the hosts with docker-machine commands.

You can use Machine to create Docker hosts on your local Mac or Windows box, on your company network, in your data center, or on cloud providers like AWS or Digital Ocean.

Docker Machine

- docker-machine create —driver virtualbox dockerhost1
- docker-machine create —driver digitalocean —digitalocean-access-token <your access token> —digitalocean-size 2gb testhost2
- docker-machine create —driver amazonec2 —amazonec2-access-key AKI***** —amazonec2-secret-key 8T9******aws-sandbox
- docker-machine create -d generic --generic-ip-address {ip-address} {docker-vm-name}

Docker Machine

- docker-machine create —driver virtualbox dockerhost1

Env Setup

  - docker-machine env machinename
  - docker-machine env -u
  - eval $(docker-machine env machinename) - This configures our docker CLI utility to use this particular machine

- docker-machine ls
- docker-machine stop <name>
- docker-machine start <name>
- docker-machine ip <name>

Docker Machine

- docker-machine ls
- docker-machine stop <name>
- docker-machine start <name>
- docker-machine ip <name>

Clustering & Load Balancing with Docker Swarm

- Docker Swarm is a tool that clusters Docker hosts and schedules containers
  - Turns a pool of host machines into a single virtual host
  - Ships with simple scheduling backend
  - Supports many discovery backends like Hosted discovery, etcd, consul, Zookeeper, Static files

Swarm Mode

- Natively managing a cluster of Docker Engines called the Swarm
- Docker CLI to create swarm, deploy apps and manage swarm
- No single point of failure
- Declarative state mode
- Self organizing and self healing
- Service discovery, load balancing, scaling
- Rolling updates

Clustering & Load Balancing with Docker Swarm

1. Create nodes and setup docker on all machines willing to be part of network
2. Create Swarm cluster
3. Join Nodes to the swarm
4. Deploy Service to run on the nodes

Create Cluster

1. Create 1 manager and 3 worker nodes
   - docker-machine create —driver virtualbox manager1
   - docker-machine ip manager1 ls
   - docker-machine

2. SSH into manage VM
   - docker-machine ssh manager1

4. Create docker cluster
   - docker swarm init —advertise-addr MANAGER_IP
   - docker node ls
   - docker node inspect <hostname> —pretty

Deploy Service

- Join worker node to cluster
  - Get the token: docker swarm join-token worker
- SSH into each worker machine
  - RUN the token command on each worker machine

- Deploy Service:
  - docker service create --replicas 5 -p 80:80 --name myservice nginx
  - docker service ls
  - docker service ps myservice

Command Summary

- docker-machine ls (docker-machine regenerate-certs <machinename>
- docker-machine ssh manager1
- docker swarm init —advertise-addr <managerIP>
- docker swarm join-token worker
- docker-machine ssh worker1 (and run the token command)
- docker service create —replicas 5 —name myservice1 -p 8080:80 nginx: 1.12
- docker service ls
- docker service ps <servicename>

Command Summary

- docker service scale myservice1=10
- docker node inspect self
- docker service update —image <imagename>:<tag> myservice1 node
- docker update —availability drain <nodename>
- docker node update —availability active <nodename>
- docker swarm leave —force
- docker swarm rm <servicename>

Scale & Manage Service

- Scale a service: docker service scale nginx=10
- Inspecting Nodes: docker node inspect self, docker node inspect worker1, docker node inspect — pretty worker1
- Remove a service: docker service rm nginx
- Apply rolling updates: docker service update --image <imagename>:<version> web

Manage Cluster Nodes

- Managing nodes
  - docker node update --availability drain <NODE>
  - docker node update --availability active <NODE>
  - docker node inspect master --format "{{ .ManagerStatus.Reachability }}"
  - docker node inspect manager1 --format "{ .Status.State }}
    {"
- Command to leave the cluster
  - docker swarm leave —force

Deploying Services using Compose YAML file

```
docker stack deploy --compose-file=docker-compose.yml myservice
```

```
docker service scale myservice_db=2
```

```
docker stack rm <stack-name>
```

Kubernetes

- Kubernetes is a platform for hosting Docker Containers in a clustered environment with multiple Docker hosts
- Provides container grouping, load balancing, auto-healing, scaling features
- Project is started by Google
- Contributors == Google, CodeOS, Redhat, Mesosphere, Microsoft, HP, IBM, VMWare, Pivotal, Saltstack etc

Key Concepts of Kubernetes

- Master
- Node
- Pod - A group of Containers
- Kubelet - Container Agent
- Services
- Deployments
- Replica Sets
- Labels - Labels for identifying pods
- Selectors

Kubelet

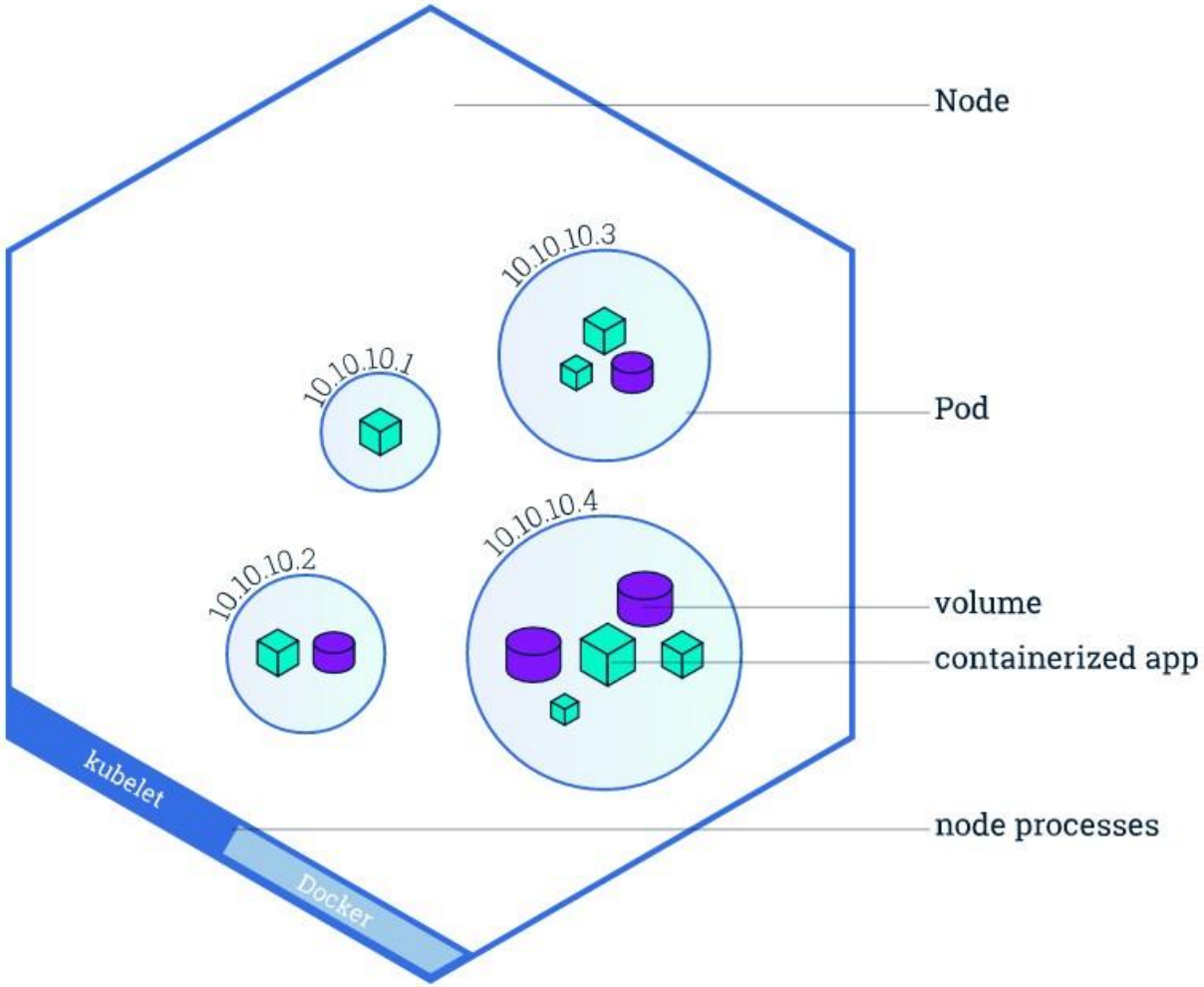
Kubelet, a process responsible for communication between the Kubernetes Master and the Nodes; it manages the Pods and the containers running on a machine.

The kubelet is the primary "node agent" that runs on each node. The kubelet works in terms of a PodSpec. A PodSpec is a YAML or JSON object that describes a pod. The kubelet takes a set of PodSpecs that are provided through various mechanisms (primarily through the apiserver) and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn't manage containers which were not created by Kubernetes.

Nodes

- A Pod always runs on a **Node**. A Node is a worker machine in Kubernetes and may be either a virtual or a physical machine, depending on the cluster.
- Each Node is managed by the Master.
- A Node can have multiple pods, and the Kubernetes master automatically handles scheduling the pods across the Nodes in the cluster. The Master's automatic scheduling takes into account the available resources on each Node.
- Node will run docker

Nodes


Node
10.10.10.3
10.10.10.1
Pod
10.10.10.4
10.10.10.2
volume
containerized app
kubelet
Docker
node processes

Pods

- A Pod is a Kubernetes abstraction that represents a group of one or more application containers and some shared resources for those containers.Those resources include:
  - Shared storage, as Volumes
  - Networking, as a unique cluster IP address
  - Information about how to run each container, such as the container image version or specific ports to use

Kubernetes Cluster

- Minikube - For testing & Learning purpose
- Custom Cluster from Scratch
- Hosted Solutions
  - Google Container Engine
  - Azure Container Service
  - IBM Bluemix Container Service
- Turn-key cloud Solutions
    - AWS Ec2
    - Asure
    - CenturyLink Cloud
    - IBM Bluemix

Introduction to Minikube & Kubectl

- Minikube is a lightweight Kubernetes implementation that creates a VM on your local machine and configures a simple cluster containing only one node. Minikube is available for Linux, Mac OS and Windows systems.

- The Minikube CLI provides basic bootstrapping operations for working with your cluster, including start, stop, status, and delete.

- Kubectl is a command line interface to interact with kubenetes.

- ***kubectl version*** **-** To check if kubectl is installed and running. The client version is the kubectl version; the server version is the Kubernetes version installed on the master.

HandsOn - Create a Kubernetes Cluster

- minikube version
- ***minikube start*** **-** This command will create a vm on Virtual Box and setup a kubernetes cluster
- ***kubectl config use-context minikube*** **-** This will set the context of your machine to minikube.

- **kubectl cluster-info** - This command give detail information about the cluster
- **kubectl get nodes** - This command shows all nodes that can be used to host our applications.

Create a Deployment

- **kubectl run hello-nginx —labels="run=load-balancer" —image=nginx:** This creates a deployment and we can investigate into the Pod that gets created, which will run the container:
- kubectl get deployments hello-nginx
- kubectl get replicasets
- kubectl get pods OR kubectl get pods --selector="run=load-balancer"
- kubectl describe deployments hello-nginx
- minikube dashboard —url=true

Scaling Up

- kubectl scale --replicas=3 deployment/<nameofdeployment>
- kubectl get deployment - To see the status

Rolling Out Changes

- kubectl set image deployment/<deploymentname> nginx=nginx:1.9.1
- kubectl rollout status deployment/<deploymentname>

Exercise

- Create a deployment with nginx of 5 replicas

- Scale up to 10 replicas

Deployment

- Deployment: A *Deployment* provides declarative updates for Pods and ReplicaSets (the next-generation ReplicationController). You only need to describe the desired state in a Deployment object, and the Deployment controller will change the actual state to the desired state at a controlled rate for you. You can define Deployments to create new resources, or replace existing ones by new ones.

Replica Sets

- ReplicaSet is the next-generation Replication Controller. ReplicaSet ensures that a specified number of pod "replicas" are running at any given time.
- However, a Deployment is a higher-level concept that manages ReplicaSets and provides declarative updates to pods along with a lot of other useful features.
- Therefore, it is recommended to use Deployments instead of directly using ReplicaSets, unless you require custom update orchestration or don't require updates at all.

Creating a Service

We have pods running nginx in a flat, cluster wide, address space. In theory, you could talk to these pods directly, but what happens when a node dies? The pods die with it, and the Deployment will create new ones, with different IPs. This is the problem a Service solves.

A Kubernetes Service is an abstraction which defines a logical set of Pods running somewhere in your cluster, that all provide the same functionality. When created, each Service is assigned a unique IP address (also called clusterIP). This address is tied to the lifespan of the Service, and will not change while the Service is alive. Pods can be configured to talk to the Service, and know that communication to the Service will be automatically load-balanced out to some pod that is a member of the Service.

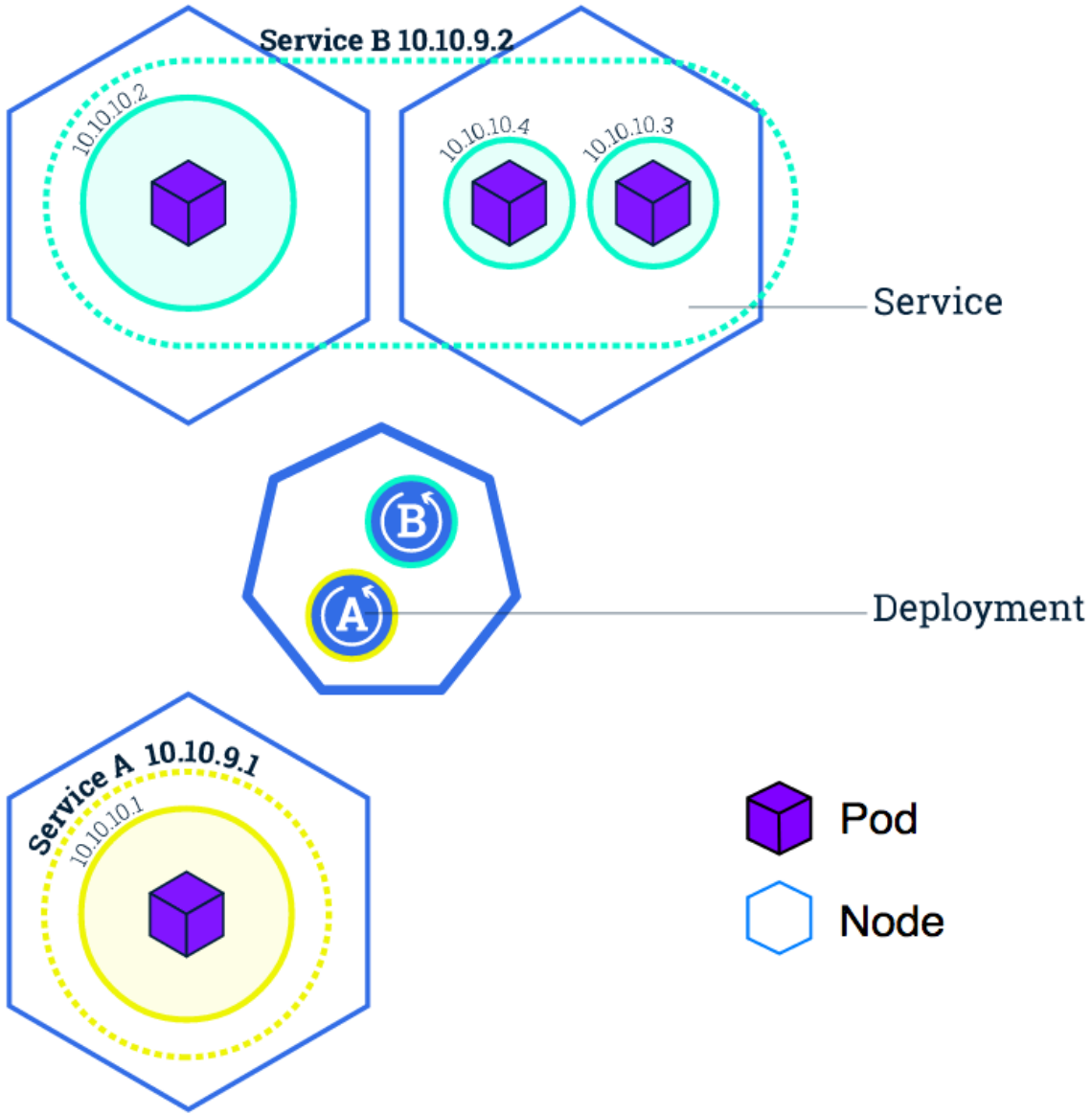
You can create a Service for your 2 nginx replicas with **`kubectl expose`**:

```
kubectl expose deployment/my-nginx
```

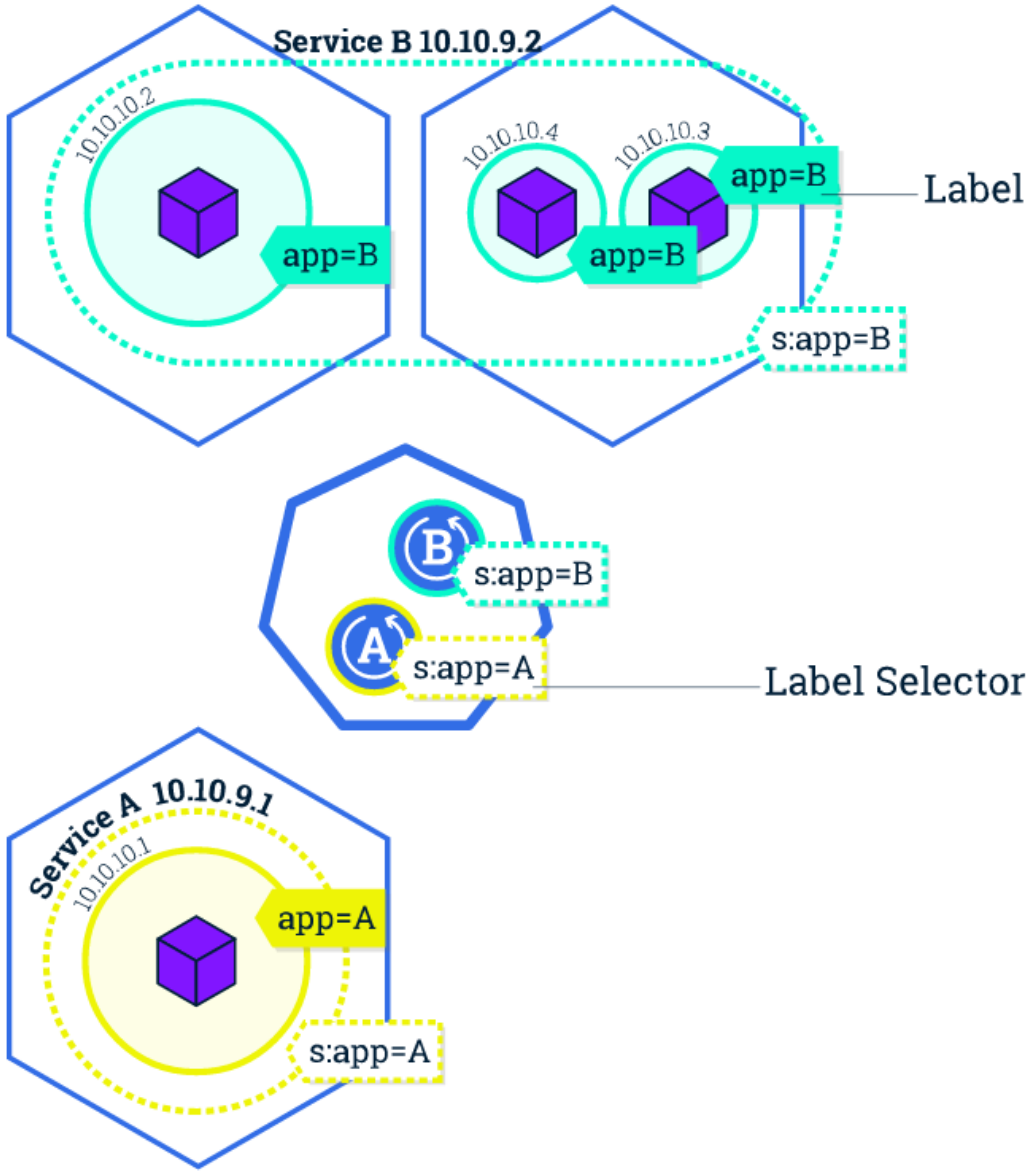
Service

- Although Pods each have a unique IP address, those IPs are not exposed outside the cluster without a Service.
- A Kubernetes **Service** is an abstraction which defines a logical set of **Pods** and a policy by which to access them - sometimes called a micro-service.
- Services match a set of Pods using labels and selectors, a grouping primitive that allows logical operation on objects in Kubernetes.
- Although Pods each have a unique IP address, those IPs are not exposed outside the cluster without a Service.
- A Service routes traffic across a set of Pods.

Service


Service B 10.10.9.2
10.10.10.2
10.10.10.4
10.10.10.3
Service
B
A
Deployment
Service A 10.10.9.1
10.10.10.1
Pod
Node

Service


Service B 10.10.9.2
10.10.10.2
app=B
10.10.10.4
10.10.10.3
app=B
app=B
Label
s:app=B
B
s:app=B
A
s:app=A
Label Selector
Service A 10.10.9.1
10.10.10.1
app=A
s:app=A

Expose a Service

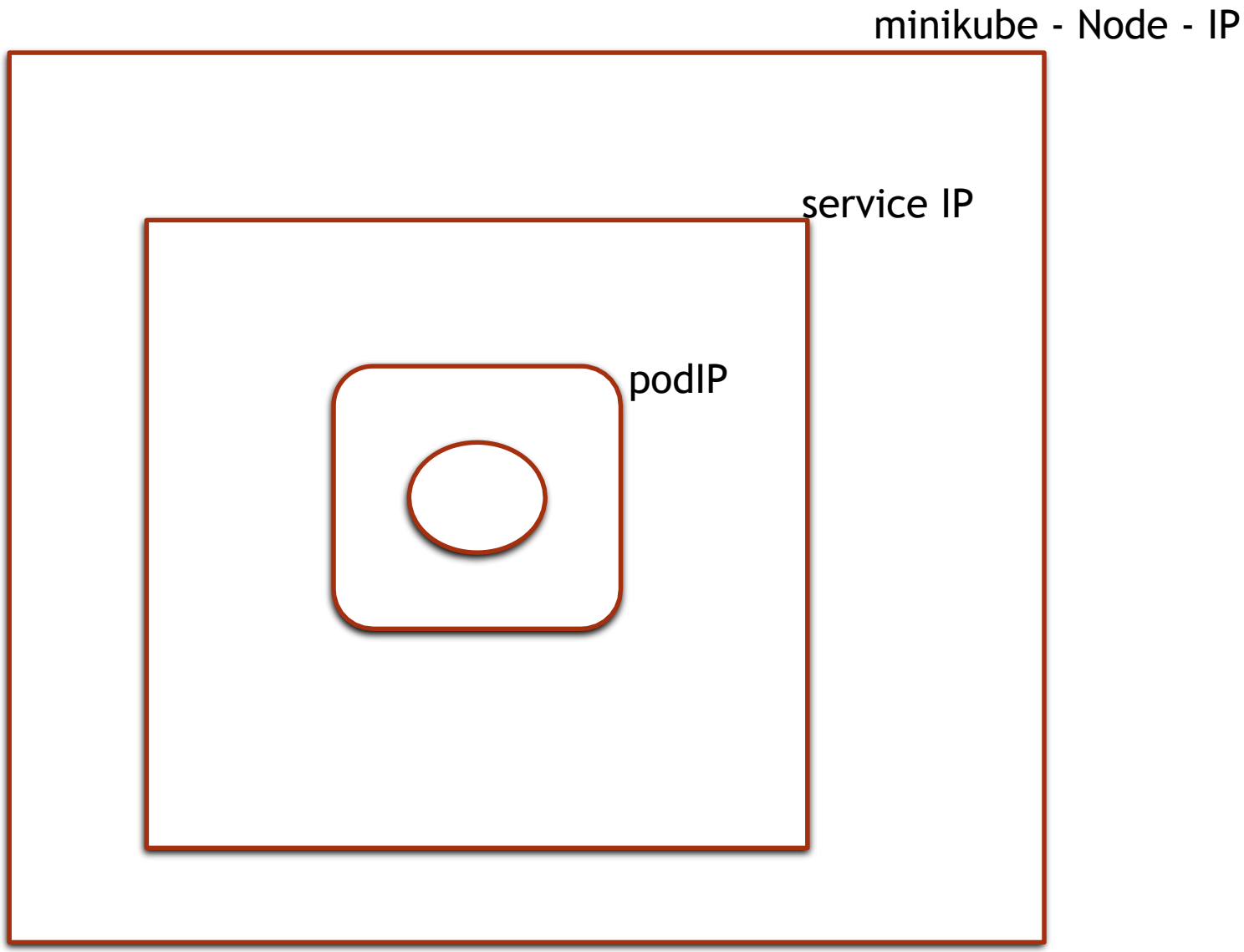
Kubernetes allows you to define 3 types of services using the ServiceType field in its yaml file.

Valid values for the ServiceType field are:

- ClusterIP: use a cluster-internal IP only - this is the default and is discussed above. Choosing this value means that you want this service to be reachable only from inside of the cluster.
- NodePort : on top of having a cluster-internal IP, expose the service on a port on each node of the cluster (the same port on each node). You'll be able to contact the service on any :NodePort address.
- LoadBalancer: on top of having a cluster-internal IP and exposing service on a NodePort also, ask the cloud provider for a load balancer which forwards to the Service exposed as a :NodePort for each Node.

Expose a Service

- **kubectl expose deployment hello-nginx —name=example-service1 —port=8080 —target-port=80**
- kubectl describe services example-service
- ssh into cluster - minikube ssh
- curl http://<ClusterIP>:<port>
- **kubectl expose deployment hello-nginx --type=NodePort —name=example-service2 —port=80**
- kubectl describe services example-service
- minikube service myservice —url


minikube - Node - IP
service IP
podIP

Creating Deployment using YAML

- Create an nginx pod, and note that it has a container port specification:
- This makes it accessible from any node in your cluster

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: nginx
        ports:
        - containerPort: 80
```

Creating a Service using YAML

This specification will create a Service which targets TCP port 80 on any Pod with the `run: my-nginx` label, and expose it on an abstracted Node port

You should now be able to curl the nginx Service on `<NodeIP>:<PORT>` from any node in your cluster.

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  type: NodePort
  ports:
  - port: 80
    protocol: TCP
  selector:
    run: my-nginx
```

- Exercise

sKillspeed

for the serious leamer