

Docker Container Course



Erdeniz Ünvan

Course Content

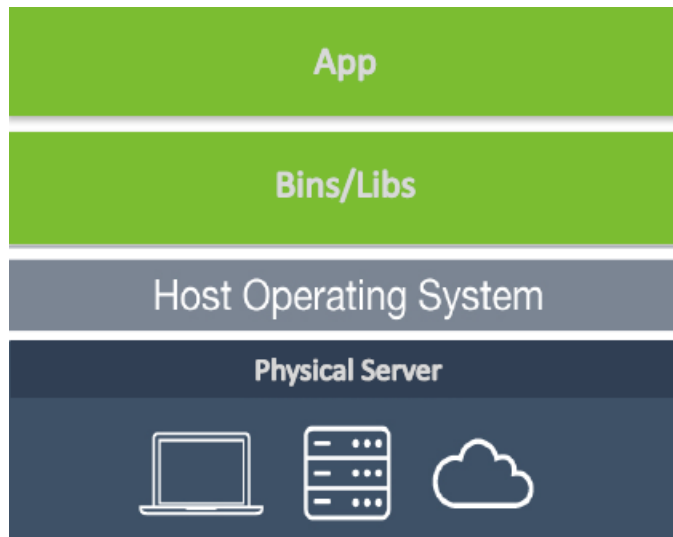
- Understanding the DevOps
- The Docker Technology
- Install Docker Server
- Docker Commands
- Docker Registry and Repositories
- Running and Managing Containers
- Creating and Running a Simple Web App
- Creating and Managing Docker Images
- Docker Volumes
- Docker Networking
- Docker Compose and YAML
- Orchestration with Swarm
- What is next?
 - Kubernetes, Ansible, Openshift

Background

- 2006, B.Sc., Anadolu University Business Administration
- 2008, d.Sc., HEPL - Haute Ecole de la Province de Liège
 - LSTM RNN Stock Market Prediction
- 2008, AISBL, Brussels, Data Scientist
- 2009, Fastlane, Ljubljana, IT Instructor
- 2011-2015, AcademyTech, Istanbul, IT Instructor
- 2016- Founder, Knowledge Club *Training & Consultancy*
 - *DevOps: Docker Container, Kubernetes*
 - *Python: Data Science, Machine Learning, Artificial Intelligence, Network Automation, Robotic Process Automation, Application Development*
 - *Developer*

1. Introduction

- What is Docker?
 - In 2013, started as opensource project at dotCloud,Inc.
 - Renamed as Docker,Inc. at October, 2013
- Infrastructure Shifts
- 90s Pre-Virtualization: Physical Servers (80s:Mainframes)

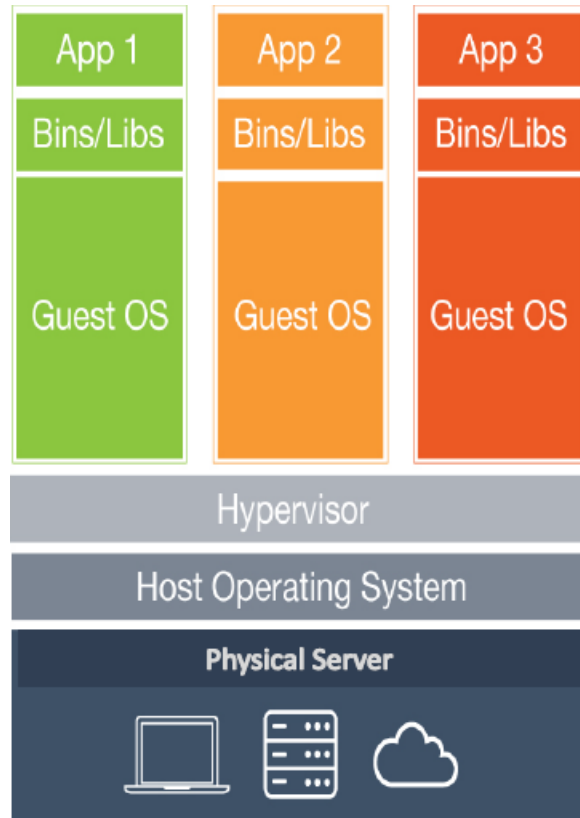


Problems:

- Huge Cost
- Slow Deployment
- Hard to Migrate

Hypervisor Virtualization

- 2000s Hypervisor Virtualization: VMWare, HyperV, Logical Domains



Benefits:

- Cost-Efficient
- Easy to Scale

Limitations:

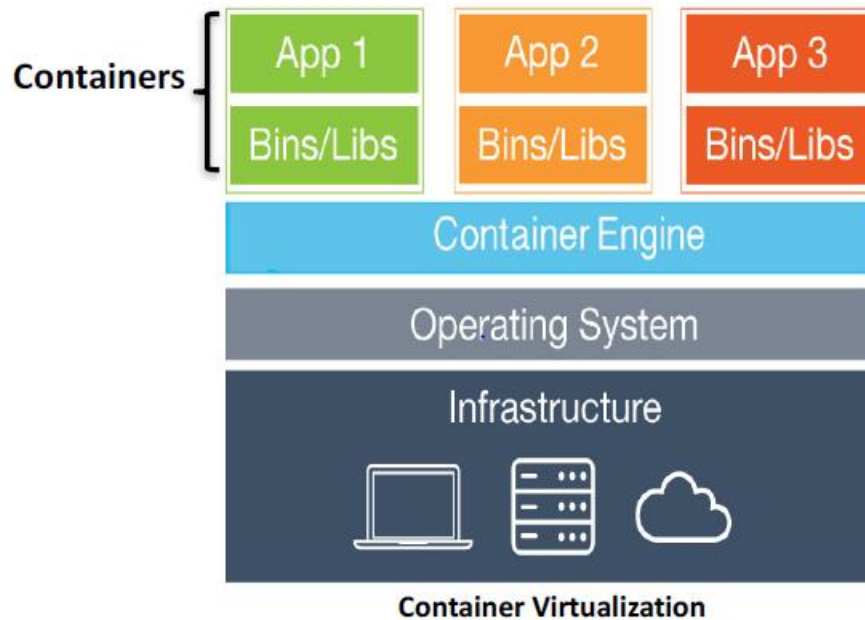
- Resource Duplication
- Application Portability

Cloud

- 2010s Cloud Technologies
 - Amazon Web Services, Microsoft Azure and Google Cloud Platform, IBM with 34b\$ Acquisition of Red Hat
 - ✓ Amazon's Flagship flagship AWS Lambda launched in 2014. Lambda can be triggered by AWS services such as Amazon Simple Storage Service (S3), DynamoDB, Kinesis, SNS, and CloudWatch
 - ✓ Google App Engine launched 2008. App Engine supports Node.js, Java, Ruby, C#, Go, Python, and PHP and database products are Cloud Datastore and Firebase. Kubernetes was created by Google in 2015 and is an open-source platform
 - ✓ Flagship, Azure Functions, allows users users to execute their code, written in languages including JavaScript, C#. Functions also interact with other Azure products including Azure Cosmos DB and Azure Storage.

Container Virtualization

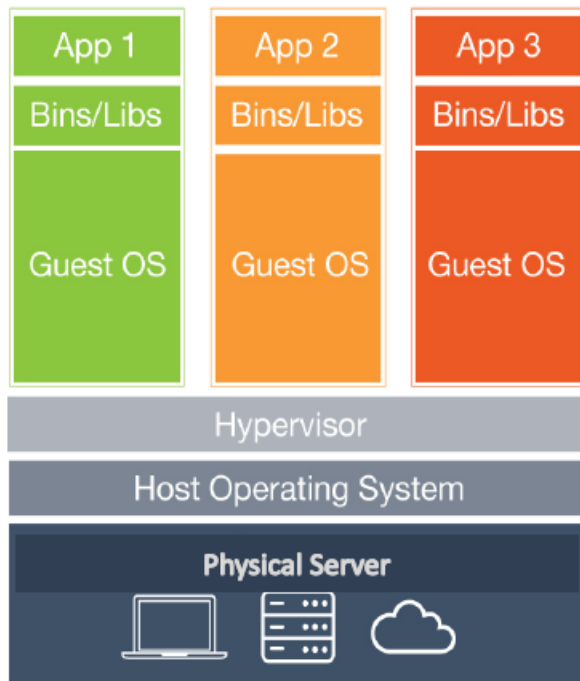
■ 2015s: Container Technologies



Benefits:

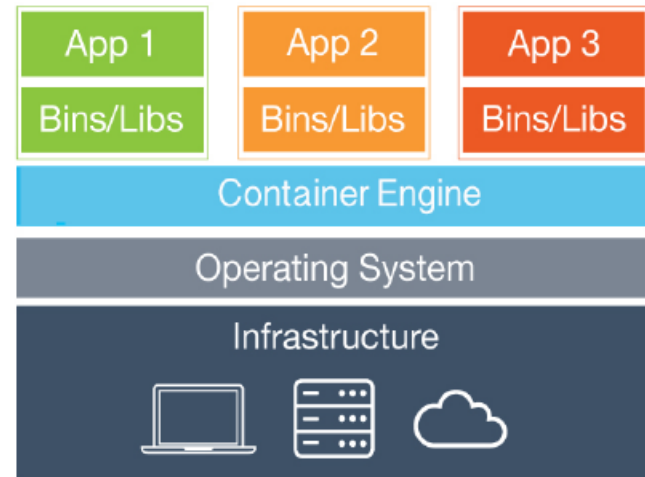
- Cost-Efficient
- Fast Deployment
- Portability

Hypervisor vs. Container Virtualization



Hypervisor-based Virtualization

Containers {



Container-based Virtualization

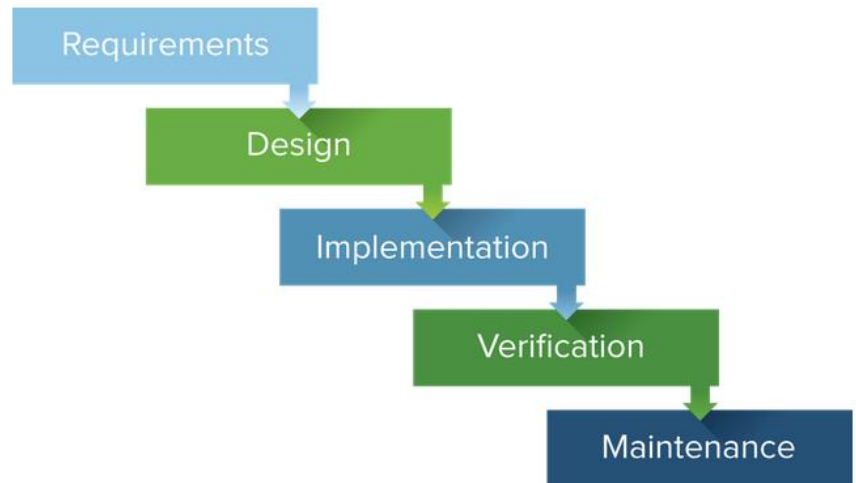
Software: Agile vs. Waterfall

Agile



- Continuous cycles
- Small, high-functioning, collaborative teams
- Multiple methodologies
- Flexible/continuous evolution
- Customer involvement

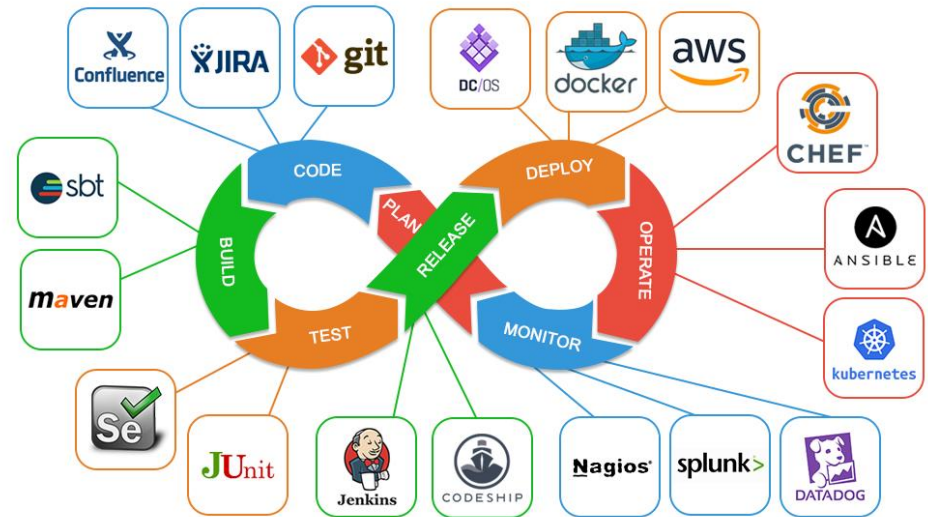
Waterfall



- Sequential/linear stages
- Upfront planning and in-depth documentation
- Contract negotiation
- Best for simple, unchanging projects
- Close project manager involvement

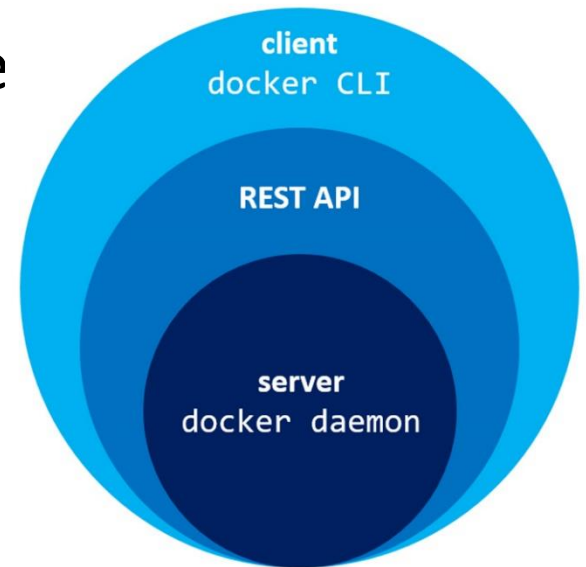
DevOps

- DevOps is an IT mindset that encourages communication, collaboration, integration and automation among software developers and IT operations in order to improve the speed and quality of delivering software
- DevOps is the offspring of agile software development
- DevOps Practices:
 - Continuous Integration
 - Continuous Delivery
 - Microservices
 - Infrastructure as Code
 - Monitoring and Logging
 - Communication and Collaboration

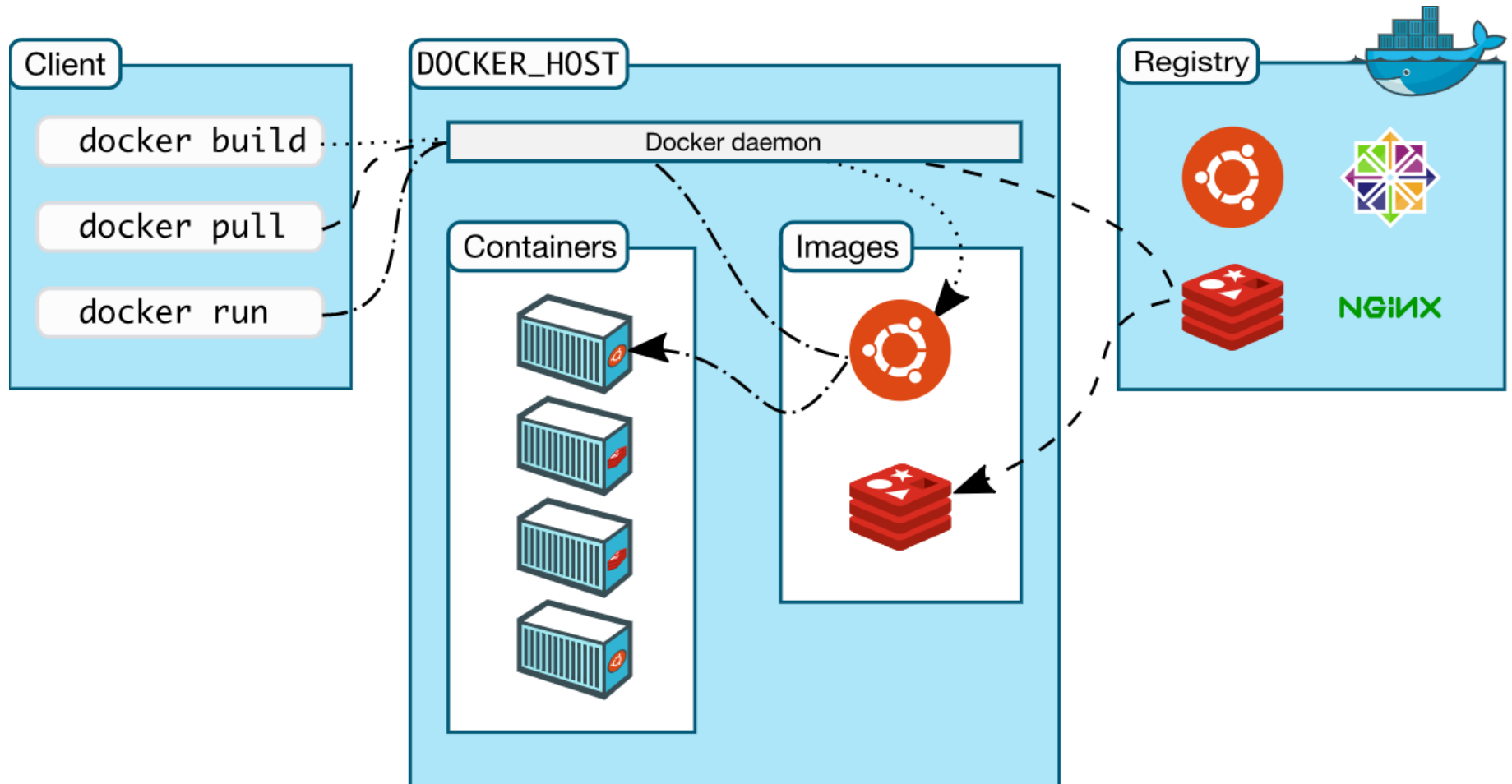


2. The Docker Technology

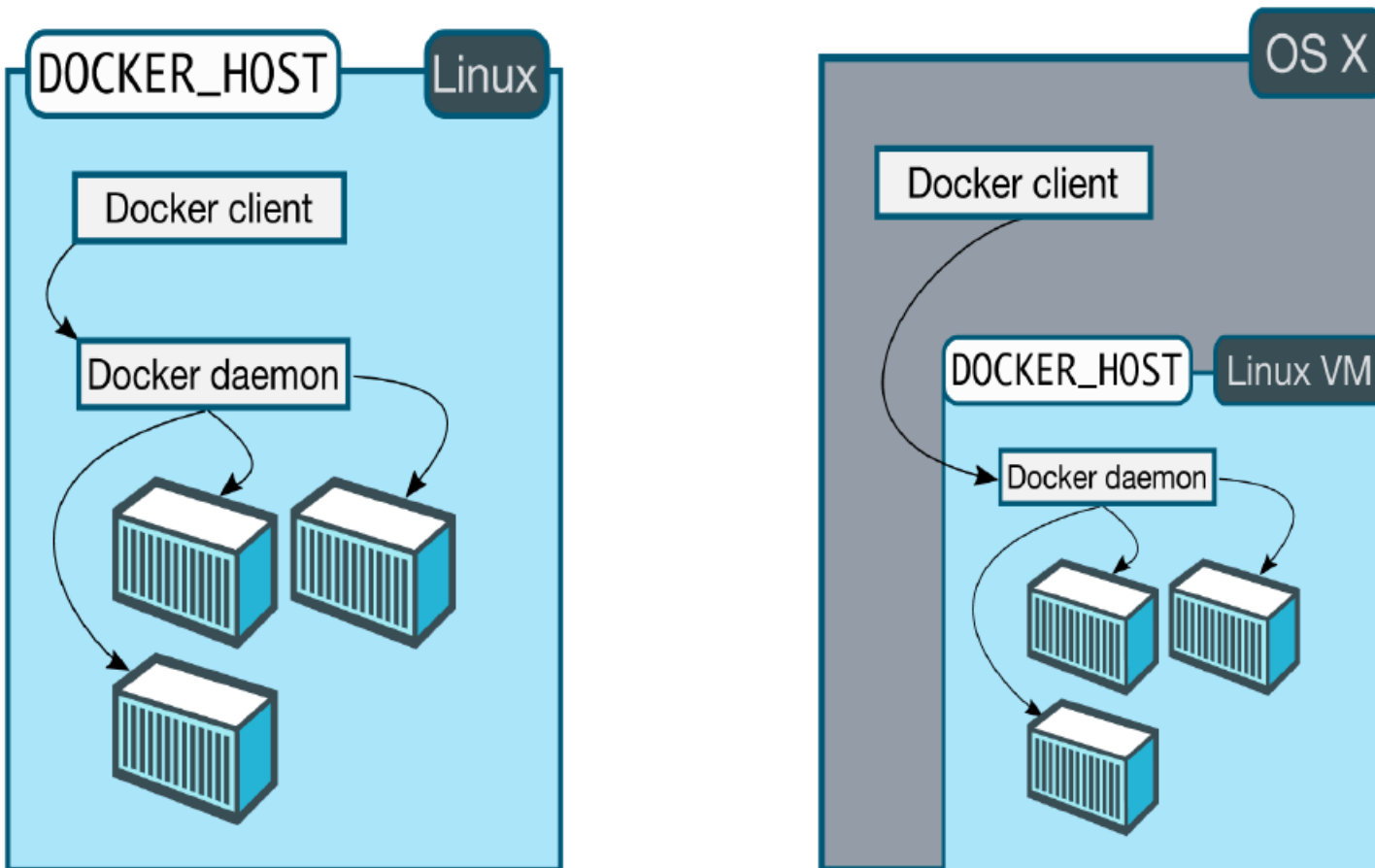
- Docker Client – Server Architecture
 - Docker Server
 - ✓ Docker Daemon running on Docker Host
 - ✓ Also referred as Docker Engine
 - Docker Client
 - ✓ CLI: `$ docker build/pull/run`
 - ✓ GUI: Kitematic
- Docker Fastest Growing Cloud Tech
- By 2020 %50 of global orgs use Docker
- Docker Hub Pulls: 2014:1M, 2015:1B, 2016:6B, 2017:24B



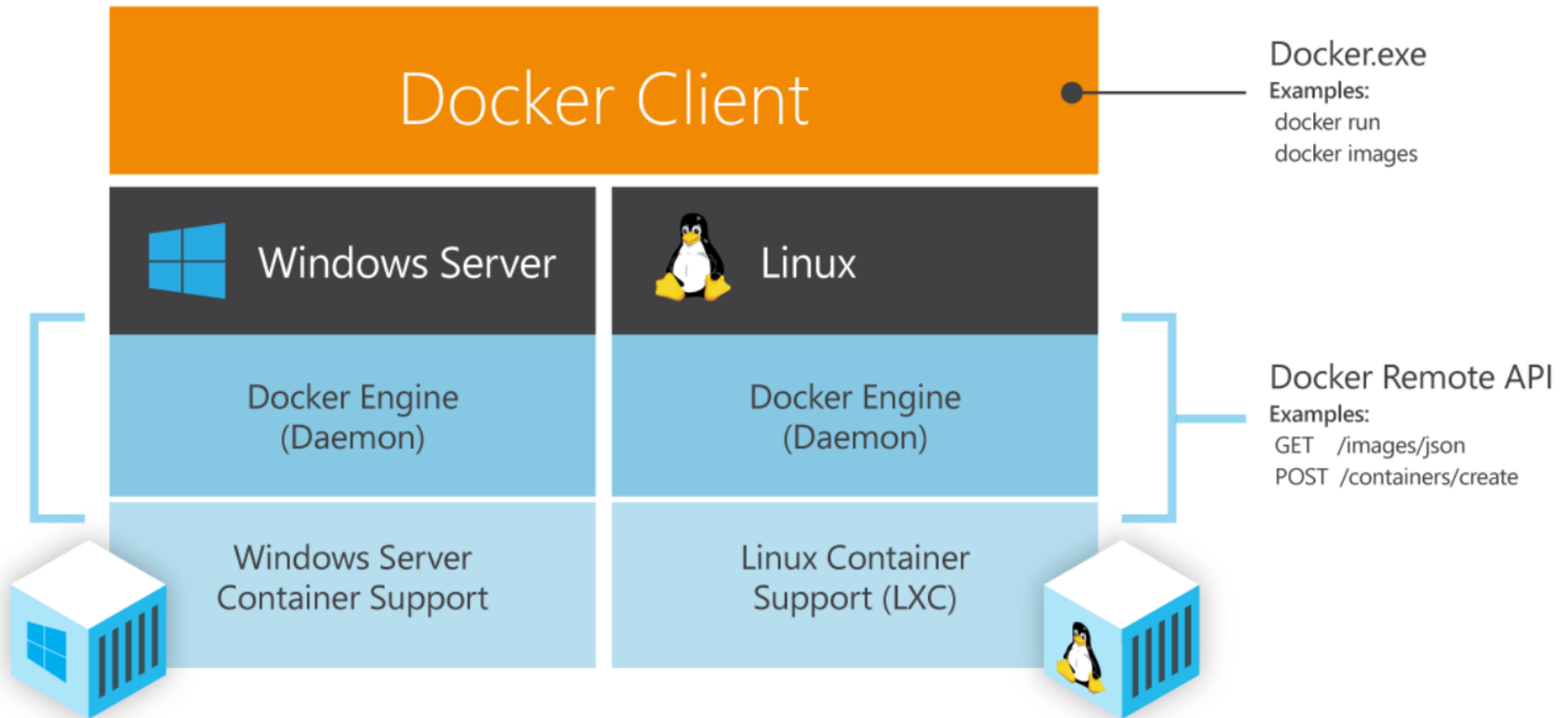
Docker Architecture



Docker on Linux and OSX

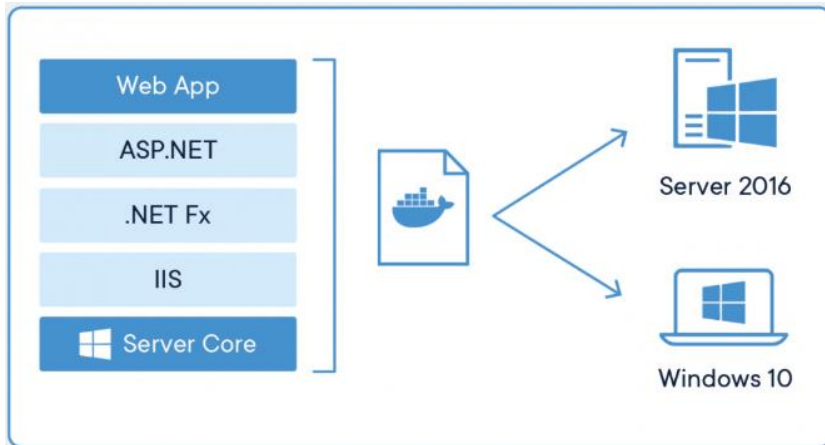


Docker on Windows



Docker on Windows

- Docker and Microsoft Bring Containers to Windows Apps
- All Windows Server 2016 and later versions come with Docker Engine. Additionally, developers can leverage Docker natively with Windows 10 via **Docker Desktop** (Development Environment)



Docker
Community Edition

Version 18.06.1-ce-win73 (19507)

Channel: stable

b0f14f1



Engine: 18.06.1-ce



Notary: 0.6.1



Compose: 1.22.0



Credential Helpers: 0.6.0



Machine: 0.15.0



Kubernetes: v1.10.3

[Release Notes](#)

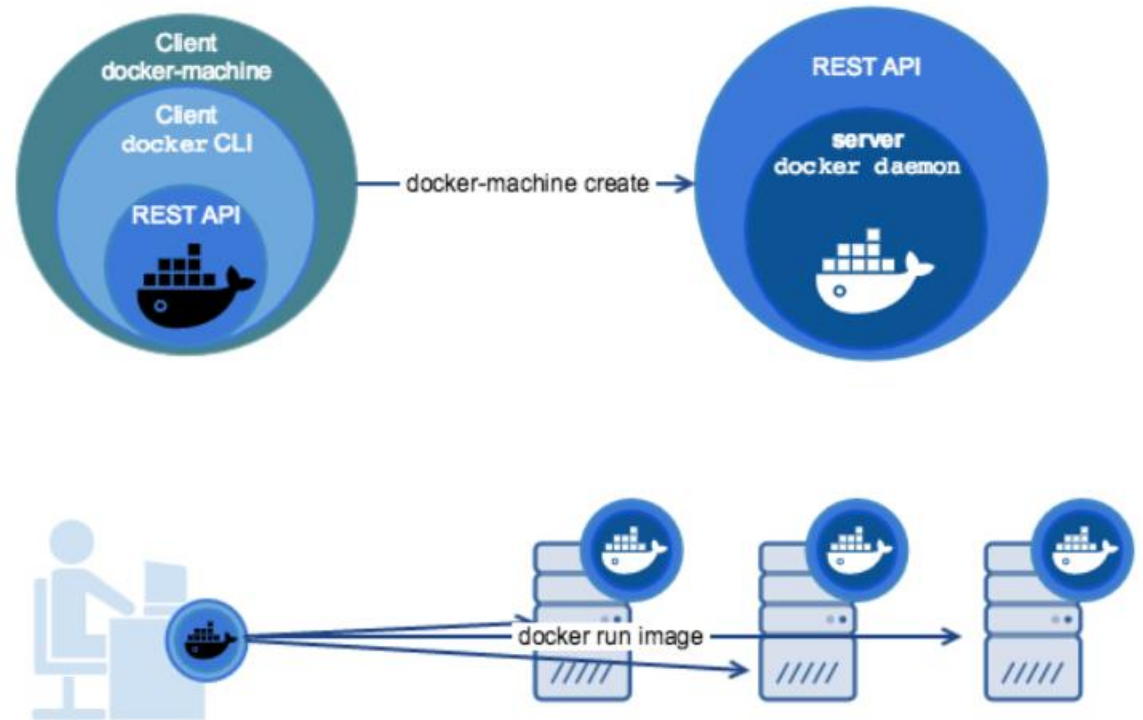
[Acknowledgments](#)

[License Agreement](#)

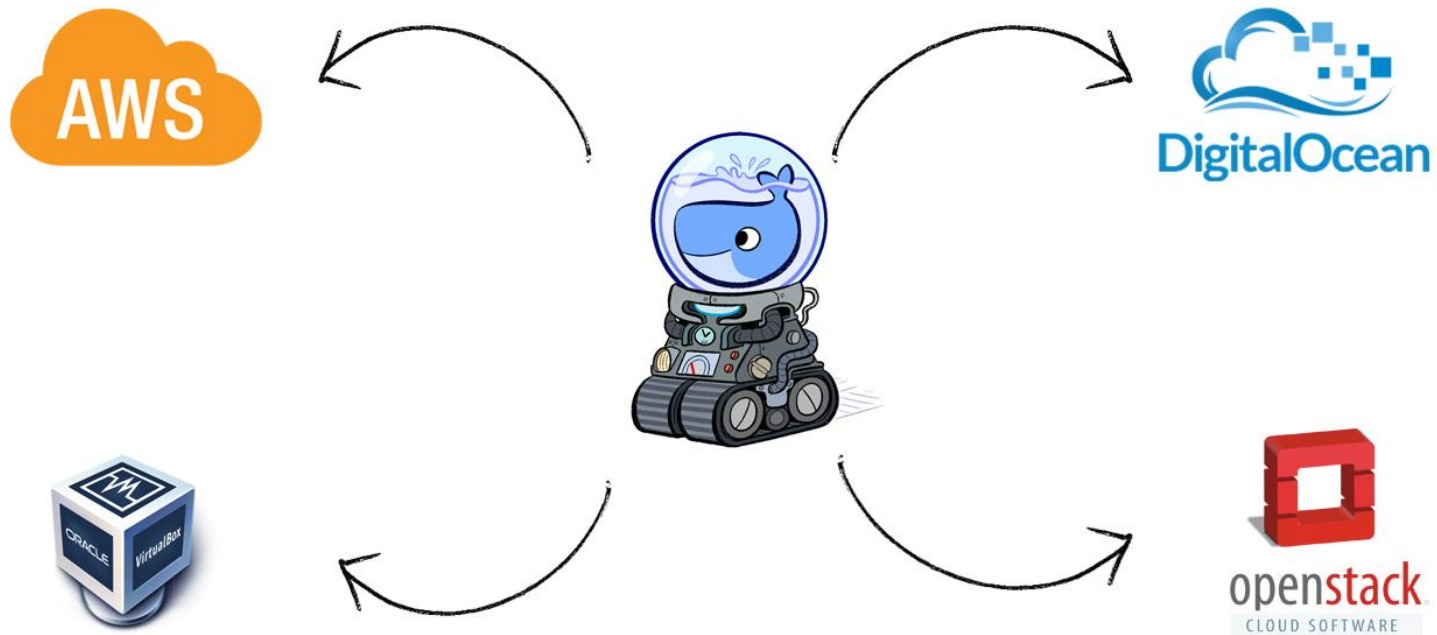
Docker Machine

- Docker Machine is a tool for provisioning and managing your Dockerized hosts (hosts with Docker Engine on them).
- Typically, you install Docker Machine on your local system. Docker Machine has its own command line client **docker-machine** and the Docker Engine client, **docker**.
- You can use Machine to install Docker Engine on one or more virtual systems. These virtual systems can be local (as when you use Machine to install and run Docker Engine in VirtualBox on Mac or Windows) or remote (as when you use Machine to provision Dockerized hosts on cloud providers).
- The Dockerized hosts themselves can be thought of, and are sometimes referred to as, managed “machines”.

Docker Machine

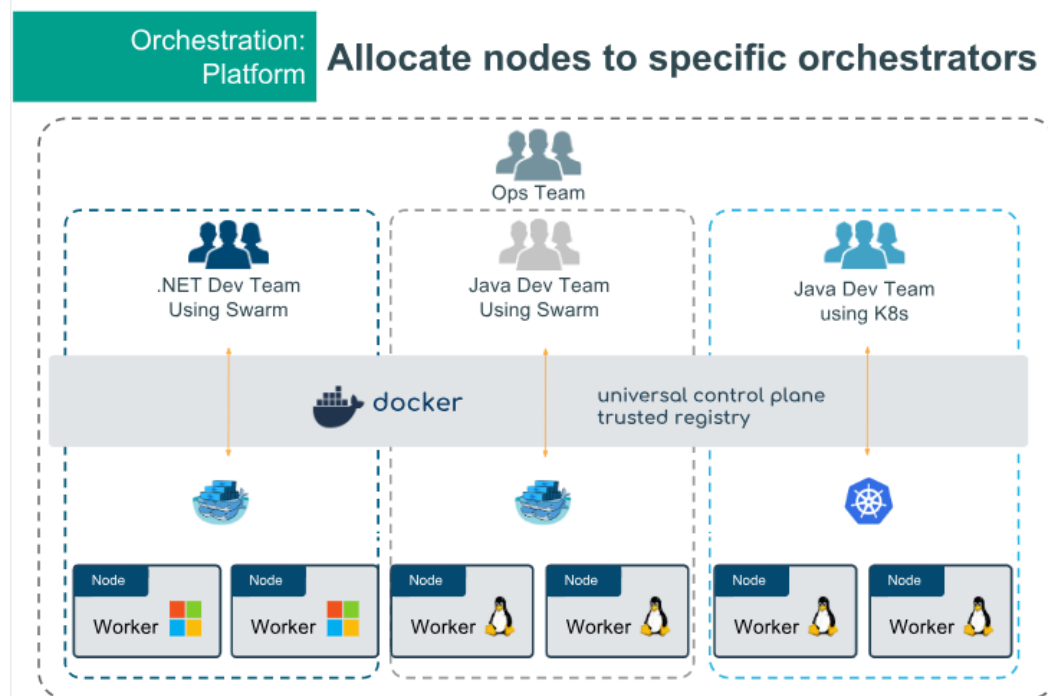


Docker Machine



Docker EE

- Docker Enterprise 2.1 is a Containers-as-a-Service (CaaS)
- The default Docker Enterprise installation includes both Kubernetes and Swarm components across the cluster



- Worker node settings: "Swarm", "Kubernetes" or "Mixed Workloads"
- UCP automatically ensures workloads are scheduled correctly
- Set on node details page and/or default for newly joined nodes
- "Mixed" not for production usage due to resource contention
- Windows and IBM Z support are Swarm only for now



3. Installation

- Install Docker for Windows
 - <https://docs.docker.com/desktop/install/windows-install/>
- **Docker Desktop for Windows**
 - Docker Desktop is a new visual tool and available with all OS types (Windows, Mac, Linux)
 - On windows you may select two alternatives as backend
 - ✓ **WSL2**: Windows Subsystem for Linux. This is our choice
 - ✓ **Hyper-V**: Not friendly with Virtualbox. Hyper-V support on Virtualbox experimental.
 - You need Windows 10 or Windows Server 2016 to install Docker Desktop for Windows.

Install Docker on Linux

■ Ubuntu

- <https://docs.docker.com/engine/install/ubuntu/>

■ CentOS/Rocky

- <https://docs.docker.com/engine/install/centos/>

- Add Docker repo and Install Docker CE

yum-config-manager --add-repo

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

yum install docker-ce

systemctl enable --now docker

- Postinstall Tasks for Linux: Create a Docker User

groupadd docker; sudo usermod -aG docker \$USER

- Logout and Login again and test docker

\$ docker version; docker run hello-world

Docker Compose

- Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services.
- Install Docker Compose on Linux
 - Docker Compose already installed for Docker Desktop
 - On Linux manually install
 - <https://docs.docker.com/compose/install/#install-compose>

```
# yum update
```

```
# yum install docker-compose-plugin
```

```
$ docker compose version
```

4. Using Containers

- Check Docker Server (Engine, Daemon) Running
 - **docker version**
 - **docker-compose version (New docker compose)**
 - **docker run hello-world**
 - **docker run -d -p 80:80 docker/getting-started**
- How to get help on commands
 - <https://docs.docker.com/>
- New style 2017: Management Commands
 - docker run vs. docker container run
 - docker ps vs. docker container ls

Image vs. Container

- An **Image** is the application we want to run
- A **Container** is an instance of that image running as a process
- You can have many containers running off the same image
- How to get images?
 - Default image **registry** is called Docker Hub
- Containers aren't **Mini-VM's**. They are just processes
- Limited to what resources they can access (file paths, network devices, running processes)
- Exit when process stops

Start a Simple Web Server

- Let's start a simple web server nginx as container
`docker container run --publish 80:80 nginx`
 - First look for the *nginx* image locally
 - if not found pull from Docker Hub
 - Start nginx and open port 80 on the host
 - Routes traffic to the container IP on port 80
 - Start a firefox and check localhost, refresh couple of times
 - if you have bind error => Apache or another Web Server running. Stop & Disable with `systemctl` or choose another port on the host like `--publish 8080:80`

Nginx Lab

- Now nginx running on the foreground and displaying logs on the terminal. Let's run it on the background

- Hit Control-c

docker container run --publish 80:80 --detach nginx

- Now running at background, unique Container ID
- Check running containers

docker ps (Old way)

docker container ls (New way)

- Notice random funny names at the end, we can specify names too. Now stop nginx

docker container stop <ID> or <Name>

docker container ls (No running containers)

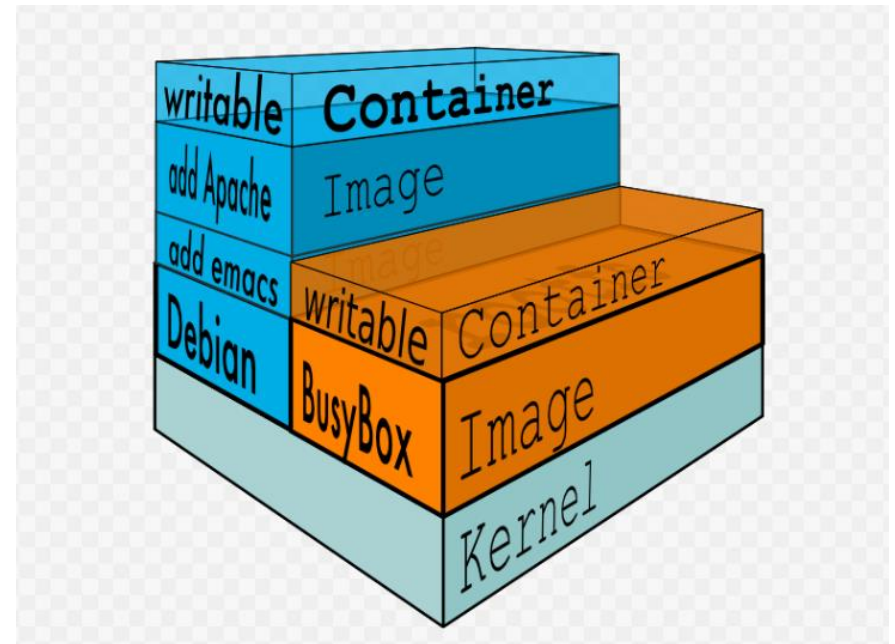
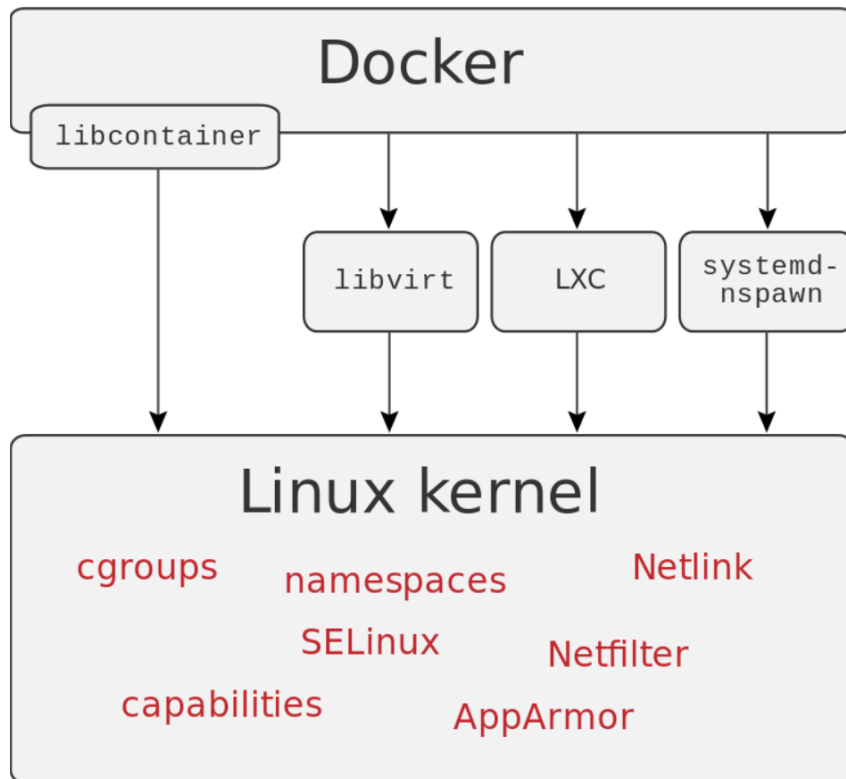
docker container ls -a (Running and Stopped containers)

Nginx

- Let's start a new container and give name "webhost"
docker container run --publish 80:80 --detach --name webhost nginx
docker container ls -a (Show both running and stopped)
 - Start firefox and refresh couple of times
 - Now let's check logs generated from container
docker container logs webhost
 - Check running processes on "webhost" container
docker container top webhost
 - Clean everything. All running and stopped containers
docker container rm <ID> (Add -f option to force)

Docker Internals

- How Dockers implemented on Linux?
 - Docker uses several Linux kernel properties like namespaces, cgroups, and UnionFS



Docker Internals

- Docker Engine uses the following namespaces on Linux:
 - **PID** namespace for process isolation.
 - **NET** namespace for managing network interfaces.
 - **IPC** namespace for managing access to IPC resources.
 - **MNT** namespace for managing filesystem mount points.
 - **UTS** namespace for isolating kernel and version
- Docker Engine uses the following cgroups:
 - **Memory cgroup** for managing accounting, limits and notifications.
 - **HugeTBL cgroup** for accounting usage of huge pages by process group.
 - **CPU group** for managing user / system CPU time and usage.
 - **CPUSet cgroup** for binding a group to specific CPU. Useful for real time applications and NUMA systems with localized memory per CPU.
 - **BlkIO cgroup** for measuring & limiting amount of blkIO by group.
 - **net_cls** and **net_prio cgroup** for tagging the traffic control.
 - **Devices cgroup** for reading / writing access devices.
 - **Freezer cgroup** for freezing a group. Useful for cluster batch scheduling, process migration and debugging without affecting prtrace.

Optional - Start and Inspect 3 Containers

- Start an nginx, a mysql, and a httpd (apache) server
 - Use logs, inspect, and stats to check details
- docker container inspect <Name>**
- docker container logs <Name>**
- docker container stats** (Like top utility)
- Run all of them --detach (or -d), name them with --name
 - nginx should listen on 80:80, httpd on 8080:80, mysql on 3306:3306
 - When running mysql, use the --env option (or -e) to pass in MYSQL_RANDOM_ROOT_PASSWORD=true
 - Use docker container logs on mysql to find the random password it created on startup
 - Use **docker container ls** to ensure everything is correct
 - LAB: Write a shell script to stop and remove all containers

Shell Access to Container

- How to get a shell access to containers, using ssh?
 - Each container starts with a default command and stops when that command exists, you can change it
 - Also you can use -i -t to get an interactive shell
 - Use exec to run additional command on any started container

docker container run -d -p 80:80 --name ng nginx

docker container exec -it ng bash

docker container top ng (you will see nginx and bash processes)

- LAB: Change index.html file and reload firefox to reflect changes

Shell Access Examples

- Let's start a Ubuntu Container with an interactive shell
 - Note that default command for Ubuntu is already bash

```
docker container run -it --name ubuntu ubuntu bash
```

```
apt-get update
```

```
dpkg -l | grep curl
```

```
apt-cache search curl
```

```
apt-get install curl
```

```
curl www.google.com
```

```
exit
```

- Now how to start and re-connect to it?

```
docker container ls -a
```

```
docker container start -ai ubuntu
```
- Also there is another mini-Linux distro called alpine ~5mb size!

```
docker container run -it --name alpine alpine bash (No Bash!)
```

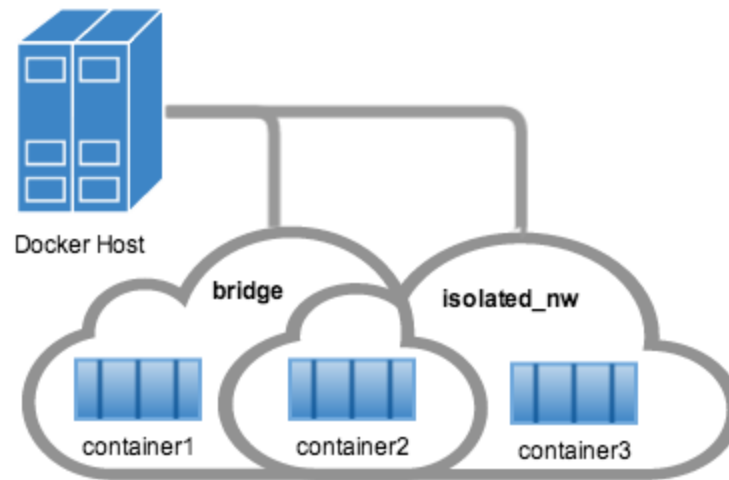
```
docker container run -it --name alpine alpine sh
```


5. Docker Networks

- Network Types
 - **bridge**: The default network driver. If you don't specify a driver, this is the type of network you are creating
 - **host**: For standalone containers, remove network isolation between the container and the Docker host, and use the host's networking directly
 - **none**: For this container, disable all networking
 - **overlay**: Overlay networks connect multiple Docker daemons together and enable swarm services to communicate with each other. You can also use overlay networks to facilitate communication between a swarm service and a standalone container, or between two standalone containers on different Docker daemons

Bridge Network

- Each container connected to a private virtual network "bridge"
- Each virtual network routes through NAT firewall on host IP
- All containers on a virtual network can talk to each other w/o -p
- Best practice is to create a new virtual network for each app:



Best Practice

- Default network is "bridge" for all created container
- Inside of the bridge network container can see each other
- However best practice is to create a new virtual network for the containers that needs to work together. There could be couple of seperated virtual networks. These networks are isolated and containers see each other over open ports
- Using a seperate virtual network has advantages. One of them is automatic DNS service. Each container within the same network can see each other with names instead of IP address, which is recommended way of operating since IP addresses can be changed frequently

Practice: Default Bridge Network

- Practice of using default bridge network
 - Let's create an alpine container in detach mode
docker container run -dit --name alpine1 alpine ash
 - alpine is a tiny linux distro, whenever we want to access it use: `docker attach alpine1` and `Control-pq` to detach again
 - Now on another terminal attach and check IP
docker attach alpine1
ip a (You see default network IP 172.17.0.2)
 - Create a second alpine and ping each other
docker container run -dit --name alpine2 alpine ash
docker attach alpine2
ip a
ping 172.17.0.2

Practice: Create New Virtual Network

- Check Network and Containers from Host Terminal

docker network ls

docker network inspect bridge

- Look at the Container Section you will see alpine1,2 with IP addresses. Now create a new net: "alp-net" and alpine3 container on this net

docker network create --driver bridge alp-net

docker network ls

**docker container run -dit --name alpine3 --network alp-net
alpine ash**

docker network inspect alp-net

docker attach alpine3

ifconfig (You will see new ip 172.18.0.x)

Practice: DNS on Virtual Network

- Now alpine1 and alpine2 on the default "bridge" and alpine3 on the alp-net. How to make alpine2 see alpine3? Create another network interface for alpine2 on the alp-net

docker network connect --help

docker network connect alp-net alpine2

docker network inspect alp-net (See Container Section)

docker attach alpine2

ifconfig

ping 172.18.0.2 (alpine2 can ping alpine3 IP using alp-net)

ping alpine3 (DNS Service enables using hostnames)

ping alpine1 (DNS is not available for default bridge)

6. Docker Images






<https://hub.docker.com/explore/>



[Dashboard](#) [Explore](#) [Organizations](#) [Create](#) ▾

tahsin42 ▾

Explore Official Repositories

 nginx official	10.4K STARS	10M+ PULLS	➤ DETAILS
 alpine official	4.7K STARS	10M+ PULLS	➤ DETAILS
 busybox official	1.4K STARS	10M+ PULLS	➤ DETAILS
 redis official	6.2K STARS	10M+ PULLS	➤ DETAILS
 mongo official	5.3K STARS	10M+ PULLS	➤ DETAILS

What is an Image?

- Official definition: "An Image is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime."
- Images are App binaries and dependencies
- Not a complete OS. No kernel, kernel modules
- Small as one file, Big as a CentOS Linux with yum, Nginx, Apache, MySQL, Mongo, etc.
- Usually use a Dockerfile to create them
- Stored in your Docker Engine image cache
- Permanent Storage in Image Registry => hub.docker.com

Create a Docker ID

- Create a free account at <https://hub.docker.com/> and login, so you can create public repositories and only one private repository. You can choose paid plans to have more private repositories.
- Hit Explore to view official images. Official images have approved by Docker Inc. with only names and "official" tag. When you create an image, it should have <your account name> /<repo name>
- Choosing the right image: Search for Nginx and choose the image with "official" tag and lots of pulls and stars
- Goto Details and Check Tags. To ensure the current version choose the "latest" tag which is default.

Practice: Pull Images

- Let's pull different versions of Nginx from Docker Hub
 - Check local nginx images first
 - If you see latest tag rename it, otherwise overwritten!

docker image tag nginx:latest nginx:old

docker pull nginx (Pulls the tag:latest)

docker pull nginx:1.23 (Check the latest version)

docker image ls

- Notice the speed, not downloading everything
- Check image size are identical but not consume disk

Image Layers

- Images are made up of file system changes and metadata
- Each layer is uniquely identified and only stored once on a host using SHA and UnionFS (like zfs)
- This saves storage space on host and transfer time on push/pull
- A container is just a single read/write layer on top of image using COW
- Use **docker image history** and **inspect** to see details
docker image history nginx:latest
docker image inspect nginx:latest

Image Layers

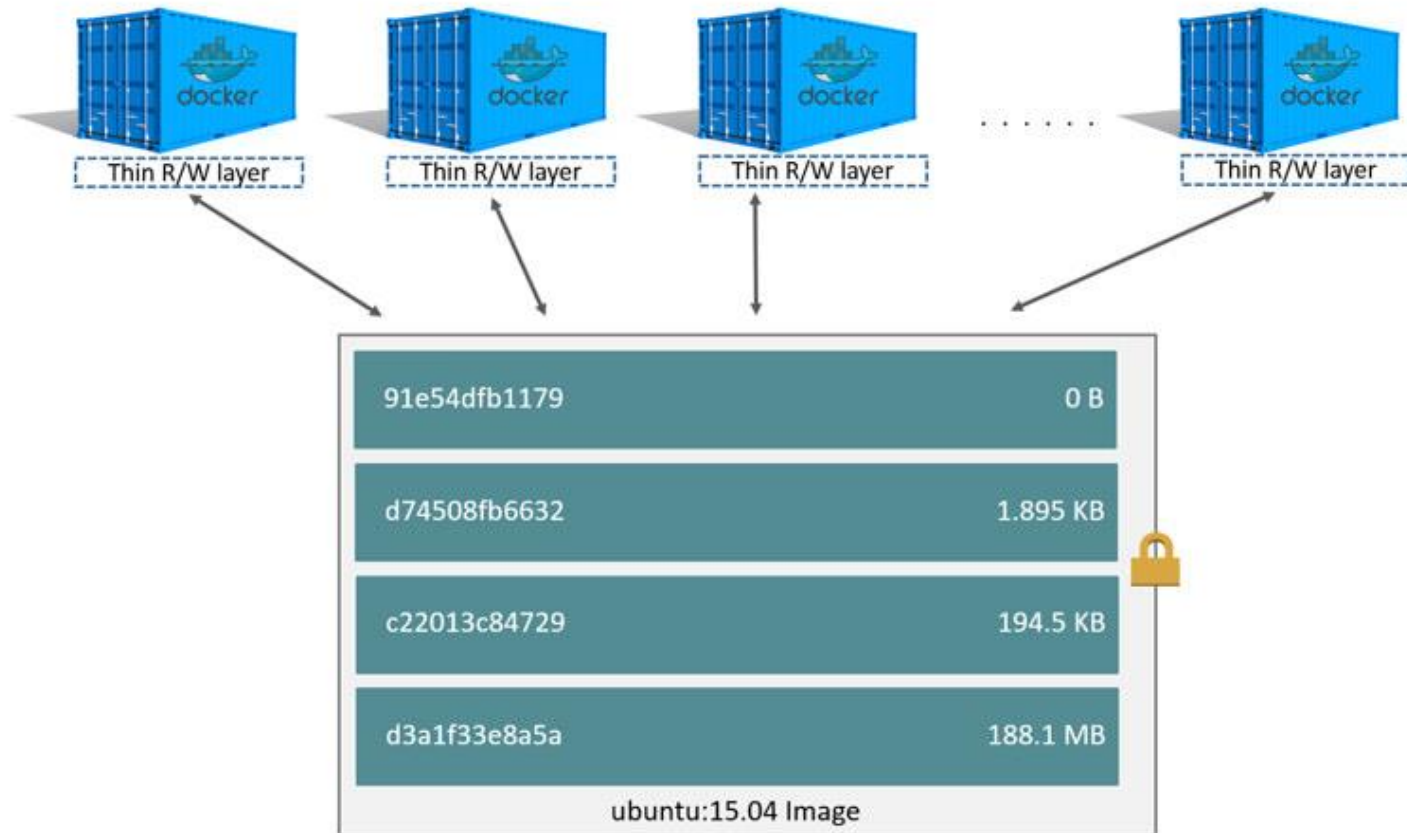


Image Layers



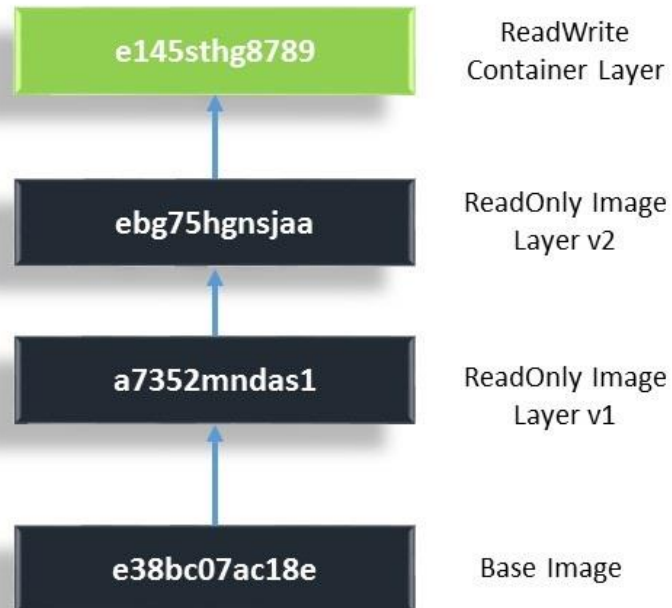
DOCKER IMAGE LAYERS

1. Started Image Layer 3 as a container and accessible by users

1. Started Image Layer v1 as a container.
2. Installed and Configured https web server.
3. Committed new layer v2

1. Started Base Image (**docker.io/centos**) as a container.
2. Package Updated on Base Image using “yum update”.
3. Committed new layer v1

Pulled CentOS image from Docker Hub using docker pull command. **Repo: docker.io/centos**



Docker Image Upload

- How to tag and upload an image to Docker Hub?
 - Use nginx image first tag and upload
docker image tag nginx tahsin42/mynginx (latest default)
docker image ls (Notice exactly same as official)
docker image push tahsin42/mynginx
 - Denied! You need to login with free docker account
docker login => user/pass
 - WARNING! Your password will be stored unencrypted in
/home/admin/.docker/config.json
 - Don't forget to **docker logout** to remove credentials
 - docker image push tahsin42/mynginx**
docker image tag nginx tahsin42/nginx:1.23
docker image push tahsin42/nginx:1.23 (Same image fast)

7. Dockerfile

- Dockerfile is recipe for creating Docker Image
- Dockerfile basics
 - FROM (base image)
 - ENV (environment variable)
 - RUN (any arbitrary shell command)
 - EXPOSE (open port from container to virtual network)
 - CMD (command to run when container starts)
 - docker image build (create image from Dockerfile)

Practice: Build Image from Dockerfile

- Let's create an image using a sample Dockerfile

```
cd dockerfile-sample-1
```

```
vim Dockerfile
```

```
docker build -t tahsin42/mynginx:2.20 .
```

- Order is important, try to make minimal changes, let's edit Dockerfile and add port 8080 on EXPOSE

```
docker build -t tahsin42/mynginx:2.21 .
```

- Very Fast Deployment since everything else is ready
- You can also clone nginx source code from github and build latest nginx version

Practice: Change index.html of Nginx

- Let's use the official nginx image and copy an index.html to create our own image, push it to Docker Hub

cd dockerfile-sample-2

vim index.html (Change it as you like)

vim Dockerfile (You can see only index.html copies)

docker build -t tahsin42/mynginx:hello .

docker push tahsin42/nginx:hello

docker run -p 80:80 --rm nginx (Hit control-c, auto rm)

docker run -p 8080:80 --rm tahsin42/nginx:hello

8. Data Volumes and Bind Mounts

- Containers are usually immutable and ephemeral
- "immutable infrastructure": only re-deploy containers, never change
- This is the ideal scenario, but what about databases, or unique data?
- Docker gives us features to ensure these "separation of concerns". This is known as "persistent data"
- Two ways: Volumes and Bind Mounts
 - Volumes: Special location outside of container UFS
 - Bind Mounts: Link container path to host path

Volumes vs. Mounts

- With **Volume**, a new directory is created within **Docker's storage directory** on the host machine, and **Docker manages** that directory's content.
- Volumes are easier to back up or migrate than bind mounts.
- You can manage volumes using Docker CLI commands or the Docker API.
- Volumes work on both Linux and Windows containers.
- Volumes can be more safely shared among multiple containers.
- Volume drivers allow you to store volumes on remote hosts or cloud providers, to encrypt the contents of volumes, or to add other functionality.
- A new volume's contents can be pre-populated by a container.

Bind Mounts

- With **Bind Mount**, a file or directory on the host machine is mounted into a container. The file or directory is referenced by its full or relative path on the host machine.
- Available since the early days of Docker.
- Bind mounts have limited functionality compared to volumes. The file or directory does not need to exist on the Docker host already. It is created on demand if it does not yet exist.
- Bind mounts are very performant, but they rely on the host machine's filesystem having a specific directory structure available.
- If you are developing new Docker applications, consider using named volumes instead. You can't use Docker CLI commands to directly manage bind mounts.

Practice: Volumes

- Let's explore volume operations using mysql database
- First check stop and remove all containers and delete existing volumes from previous work

myrm => docker container rm -f <ALL>

docker volume list

- if you ever run mysql there should be some anonymous volumes left, since deleting a container do not remove volumes. Default Location of volumes: */var/lib/docker/volumes*. Let's delete all for a fresh start

docker volume prune

Practice: Volumes

- Goto Docker Hub and Check MySQL Dockerfile about volume info => ***VOLUME /var/lib/mysql***

- Create two a mysql container and check volume names

docker pull mysql

docker image inspect mysql (Check Volume)

**docker container run -d --name mysql1 -e
MYSQL_ALLOW_EMPTY_PASSWORD=True mysql**

docker volume ls

docker volume inspect (No info about container name)

**docker container run -d --name mysql2 -e
MYSQL_ALLOW_EMPTY_PASSWORD=True mysql**

docker volume ls (We have a problem, use named volumes)

Practice: Volumes

- Clean and create two new container with named volumes
myrm; docker volume prune
**docker container run -d --name mysql1 **
**-e MYSQL_ALLOW_EMPTY_PASSWORD=True **
-v mysql-db1:/var/lib/mysql mysql
**docker container run -d --name mysql2 **
**-e MYSQL_ALLOW_EMPTY_PASSWORD=True **
-v mysql-db2:/var/lib/mysql mysql
docker volume ls
- Now stop and remove mysql2 and create mysql3 with mysql-db2, since volumes are not auto deleted with containers
docker container rm -f mysql2
docker container run -d --name mysql3 -e
**MYSQL_ALLOW_EMPTY_PASSWORD=True **
-v mysql-db2:/var/lib/mysql mysql

Practice: Bind Mount

- Clean all and create nginx1 and nginx2 containers
 - For nginx1 -p 80:80 manually connect to container and edit index.html
 - For nginx2 -p 8080:80 create a bind mount from host to container and change index.html see result
- docker container run -d --name nginx1 -p 80:80 nginx**
- Open browser on localhost and see test page
- docker container exec -it nginx1 bash**
- # echo '<h1>Welcome to Mars!</h1>' > /usr/share/nginx/html/index.html**
- Reload the browser

Practice: Bind Mount

- Create bind mount from host to nginx2 to achieve same thing w/o login into container

cd dockerfile-sample-2

docker container run -d --name nginx2 -p 8080:80 -v \$(pwd):/usr/share/nginx/html nginx

echo '<h1>Welcome to Venus!</h1>' > index.html

- Open browser <http://localhost:8080>

echo '<h1>Welcome to Jupiter!</h1>' >> index.html

- Reload browser. Very effective!
- However Host specific and can not specify in Dockerfile

9. Docker Compose

- What is it? Why do we need it?
 - Standalone Container App is not a real world scenario
 - You need many Containers working together
 - How do we specify all details about configurations, volumes, networks, etc.? Obviously not with the command line docker options
 - Docker Compose comes into act right there
- Docker compose consist of two parts:
 - YAML-formatted file that describes our solution options for:
 - ✓ Containers, networks, volumes
 - A CLI tool **docker-compose** used for local dev/test automation with those YAML files
- You need to install Docker Compose on Linux seperately
 - <https://docs.docker.com/compose/install/#install-compose>

YAML

- YAML: YAML Ain't Markup Language => <http://yaml.org>
- What It Is: YAML is a human friendly data serialization standard for all programming languages
- Default name for Docker: **docker-compose.yml**
 - if you want use other names you need **-f** options with **docker-compose** command. Similar idea with **Dockerfile** and **docker build** command
- In terms of YAML versions definitely use **v2** or higher
- **docker-compose.yml** can be used with docker directly in production with Swarm (as of v1.13)

docker-compose CLI

- Docker-compose CLI tool comes with Docker Desktop, but separate download for Linux and not a production tool but ideal for local development and test. Not in SWARM mode.
- Two most common commands are
 - **docker-compose up** # ssetup volumes/networks and start all containers
 - **docker-compose down** # stop all containers and remove containers/volumes/networks
- Compose can also build your custom images
 - Will build them with **docker-compose up** if not in cache
 - Also rebuild with **docker-compose up --build**

Template YAML File

version: '3.1' # if no version is specified then v1 is assumed. Recommend v2 minimum

services: # containers same as docker run

servicename: # a friendly name. this is also DNS name inside network

image: # Optional if you use build:

command: # Optional, replace the default CMD specified by the image

environment: # Optional, same as -e in docker run

volumes: # Optional, same as -v in docker run

servicename2:

volumes: # Optional, same as docker volume create

networks: # Optional, same as docker network create

Sample YAML File

version: '2'

services:

wordpress:

image: wordpress

ports:

- 8080:80

environment:

WORDPRESS_DB_HOST: mysql

WORDPRESS_DB_NAME: wordpress

volumes:

- ./wordpress-data:/var/www/html

mysql:

image: mariadb

environment:

MYSQL_ROOT_PASSWORD: exemplerootpW

MYSQL_DATABASE: wordpress

volumes:

- mysql-data:/var/lib/mysql

volumes:

mysql-data:

Practice1: Create a docker-compose.yml

- Goal: Create a compose config for a local Drupal CMS website
- This empty directory is where you should create a docker-compose.yml
 - - Use the `drupal` image along with the `postgres` image
 - - Set the version to 2
 - - Use `ports` to expose Drupal on 8080
 - - Be sure to setup POSTGRES_PASSWORD on postgres image
 - - Walk through Drupal config in browser at <http://localhost:8080>
 - - Tip: Drupal assumes DB is localhost, but it will actually be on the compose service name you give it
 - - Use Docker Hub documentation to figure out the right environment and volume settings

Practice1: docker-compose.yml

version: '2'

services:

drupal:

image: drupal

ports:

- "8080:80"

volumes:

- drupal-themes:/var/www/html/themes

postgres:

image: postgres

environment:

- POSTGRES_PASSWORD=mypasswd

volumes:

drupal-themes:

Practice2: Docker Compose Build

- In YAML file you can specify build if you want to create your own images. Here is an example: Goto Practice2 folder:

cat docker-compose.yml

version: '2'

services:

proxy:

build:

context: .

dockerfile: nginx.Dockerfile

ports:

- '80:80'

web:

image: httpd

volumes:

- ./html:/usr/local/apache2/htdocs/

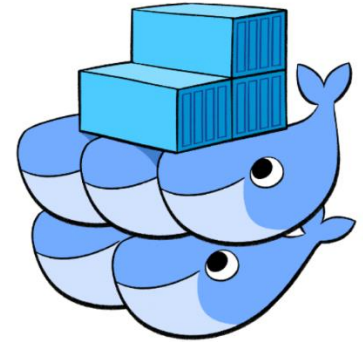
cat nginx.Dockerfile

FROM nginx:1.13

COPY nginx.conf /etc/nginx/conf.d/default.conf

docker-compose up and **docker-compose down --rmi local**

10. Swarm

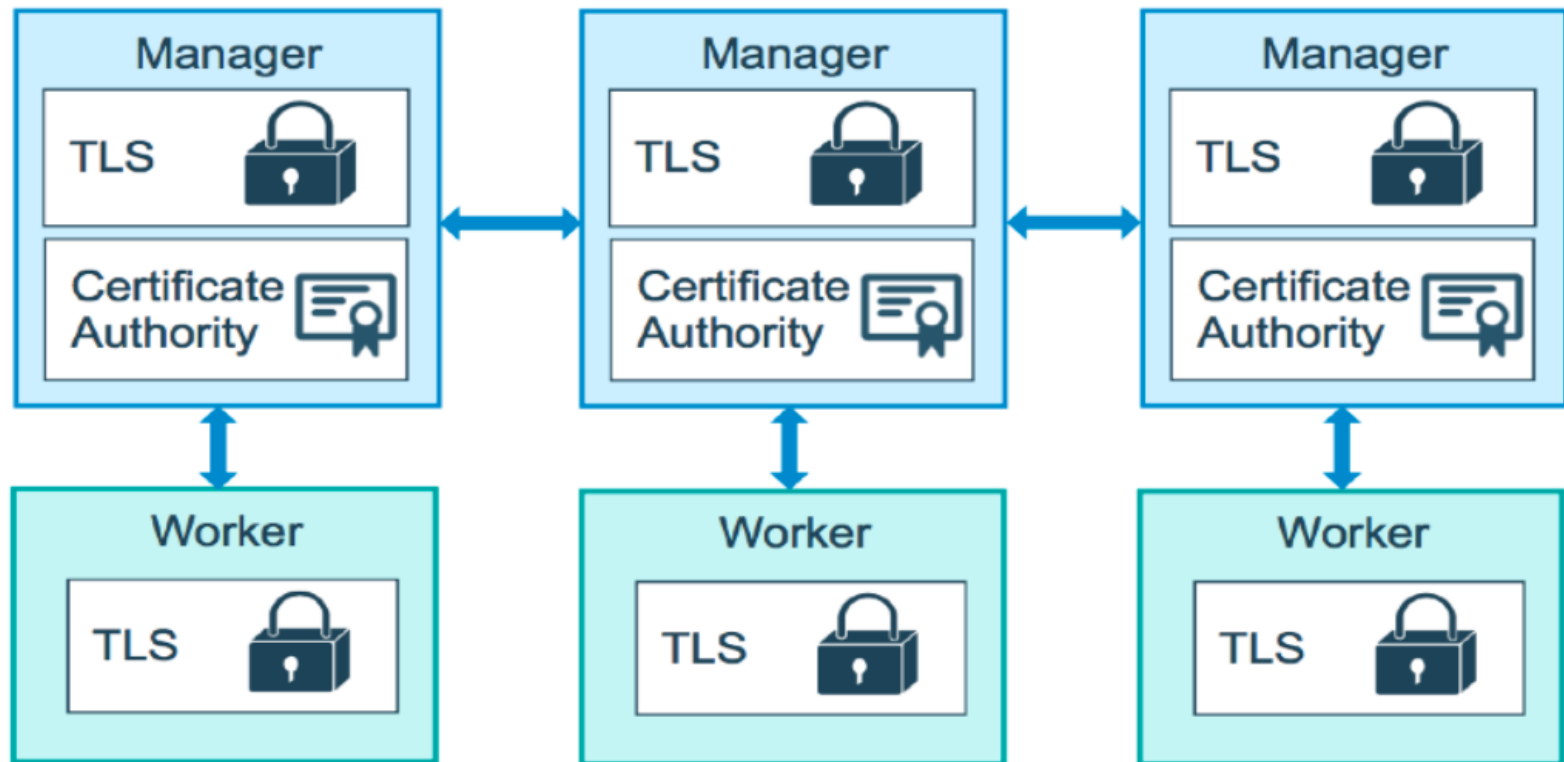


- How do we automate container lifecycle?
- How can we easily scale up/down?
- How can we ensure our containers are re-created if they fail?
- How can we replace containers without downtime (blue/green deploy)?
- How can we control/track where containers get started?
- How can we create cross-node virtual networks?
- How can we ensure only trusted servers run our containers?
- How can we store secrets, keys, passwords and get them to the right container (and only that container)?

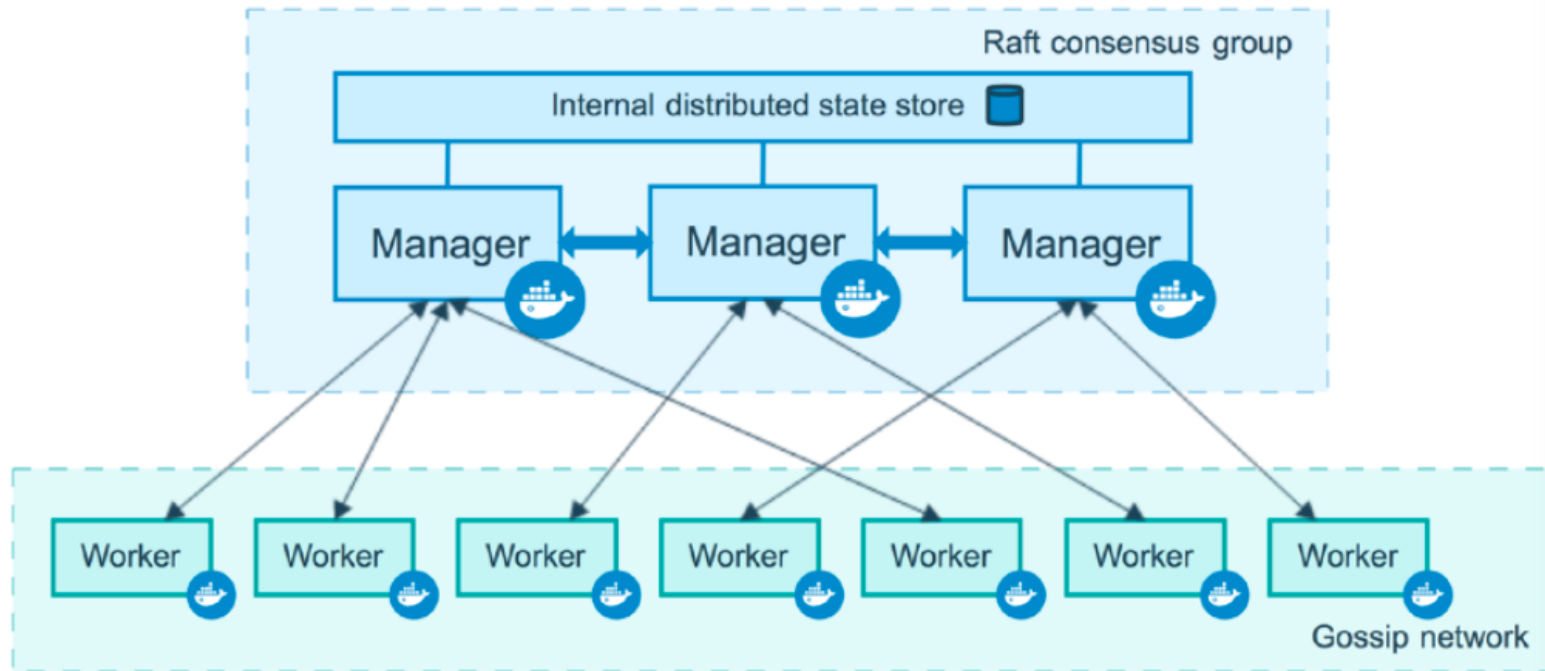
Swarm Mode: Built-In Orchestration

- Swarm Mode is a clustering solution built inside Docker
- Not related to Swarm "classic" for pre-1.12 versions
- Added in 1.12 (Summer 2016) via SwarmKit toolkit
- Enhanced in 1.13 (January 2017) via Stacks and Secrets
- Not enabled by default, new commands once enabled
 - `docker swarm`, `docker node`, `docker service`
 - `docker stack`, `docker secret`
- **`docker swarm init` => Enabled!** What Happened?
 - Lots of PKI and security automation, Root Signing Certificate created for our Swarm, Certificate is issued for first Manager node
 - Join tokens are created, Raft database created to store root CA, configs and secrets, Encrypted by default on disk (1.13+)
 - No need for another key/value system to hold orchestration/secrets, Replicates logs amongst Managers via mutual TLS in "control plane"

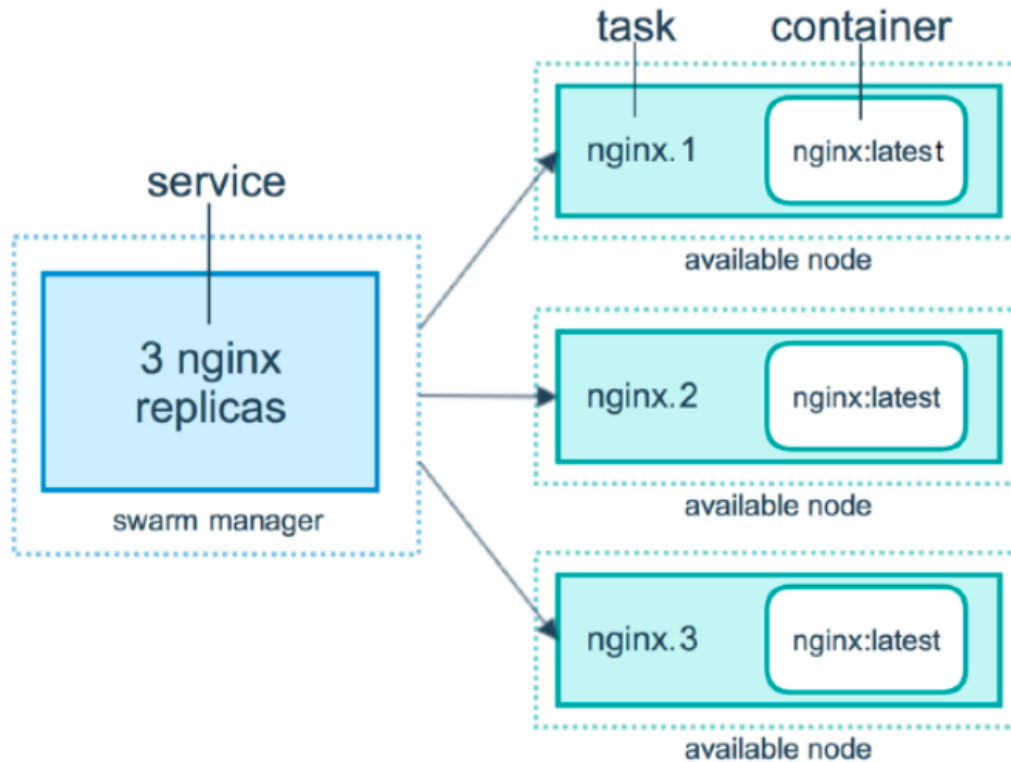
Manager and Worker Nodes



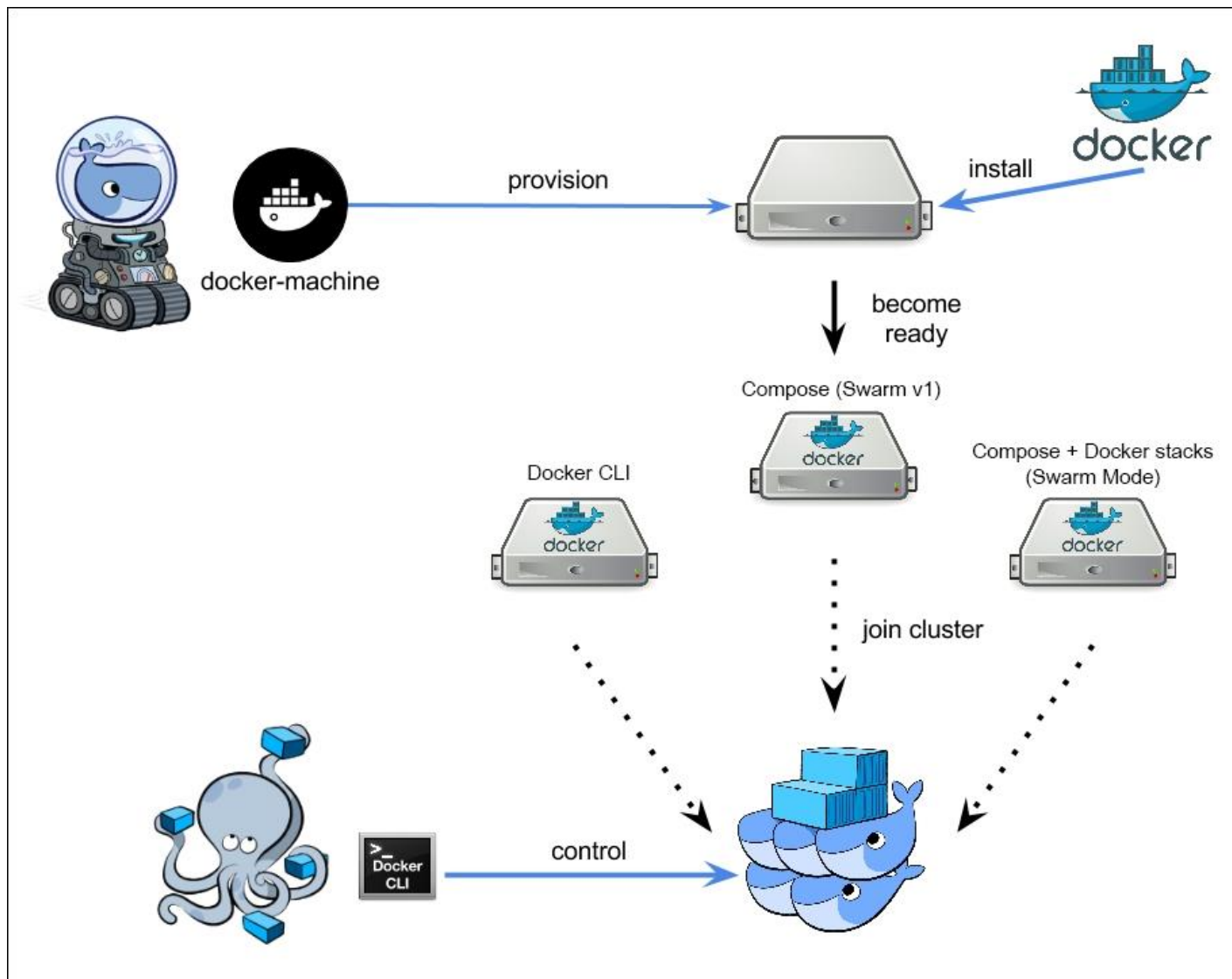
Nodes and Raft



Swarm Service



Docker Swarm



Practice: Enable Swarm in Single Node

- Check Swarm status and enable

docker info | grep -i swarm (inactive)

- Enable swarm

docker swarm init (Error: Multiple interfaces, select one)

docker swarm init --advertise-addr 192.168.56.111

- Success => Swarm initialized: current node (oz14e3meqzbwfdgtja3hh01sp) is now a manager.
- To add a worker to this swarm, run the following command:

docker swarm join --token SWMTKN-1-

2t1p9h62eqmendsqhm05f137w68jgwaeje66w2patt8gnd17b0-0blsc43iaemb6w9u6871sxhes 192.168.56.111:2377

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

Practice: Single Node Swarm

- Check nodes

docker node ls (One node, manager => *Leader*)

- *docker service create* replaces *docker container run* in swarm mode

- Create a service alpine, name it "homer", single replica

docker service create --name homer alpine ping 8.8.8.8

docker service ls

- Service "homer" is running with only 1 replica
- Use `docker service ps` to get which node it is running

docker service ps homer

docker container ls

docker container logs <container-name>

docker service logs <service-name>

Practice: Single Node Swarm

- Now, make it 3 replicas

docker service update --replicas 3 homer

docker service ls (Check all up: 3/3)

docker service ps homer (Which is running on which)

- Let's remove one container manually with `docker container rm -f` and see if swarm re-creates

docker container ls

docker container rm -f

docker service ls (if you don't see, give a little time)

- Remove service, see all 3 containers removed

docker service rm homer

docker service ls

docker container ls

How to Create Multi-Node Swarm?

- A. play-with-docker.com
 - Only needs a browser, but resets after 4 hours
- B. Local install with docker-machine + VirtualBox
 - Free and runs locally, but requires a machine with 8GB memory
- C. Digital Ocean /AWS/Google Cloud + Docker install
 - Most like a production setup, but costs monthly
- D. Create your own on the Cloud with docker-machine
 - docker-machine can provision machines for Amazon, Azure, Google

Practice: Multi Node Swarm

- Goto <https://labs.play-with-docker.com/>
- Spin-up 3 machines: node1, node2, node3
- Login node1 and ping others
ping <node2-ip>
docker info | grep -i swarm
docker swarm init
docker swarm init --advertise-addr 192.168.0.43
- On Node2 and join as worker (Later we will convert to Manager if we want)
docker swarm join --token SWMTKN-1-0xb15jzxv2zvp45d9mrbvmnvnlf9zs9h2nxqone5tqjb5uvmte-2q1e92xf2mvwdzjyk5keuicmc 192.168.0.43:2377
- On node1 run: **docker node ls** (node1 is manager:leader and node2 is worker)

Practice: Multi Node Swarm

- On Node1: Promote node2 as Manager
docker node update --role manager node2
docker node ls (Reachable)
- Add Node3 as Manager directly
 - On Node1: **docker swarm join-token manager**
 - On Node3: **docker swarm join --token SWMTKN-1-0xb15jzxv2zvp45d9mrbvmnvnlf9zs9h2nxqone5tqjb5uvmte-2j66b6wafcym7p6uotgashohv192.168.0.43:2377**
 - On Node1: **docker node ls** (Node3 also reachable)

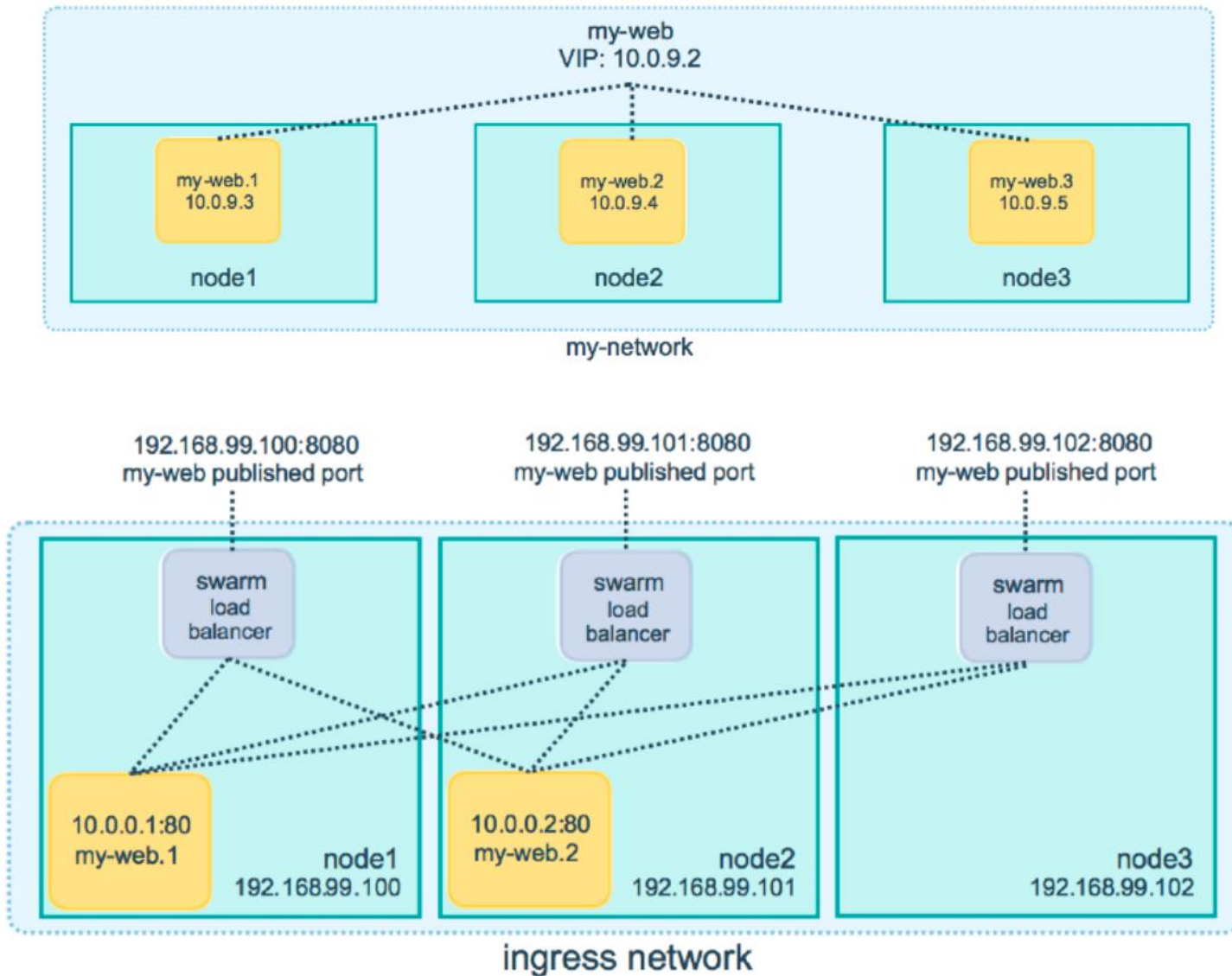
Practice: Multi Node Swarm

- Create a service again with alpine and 3 replicas
docker service create --name homer --replicas 3 alpine ping 8.8.8.8
docker service ls
docker node ps [node2]
docker service ps homer (To see containers on nodes)
- On node2 remove container, check recovery
docker container ls
docker container rm -f <name>
- On node1 check service and remove
docker service ls
docker service update --replicas 5 homer
docker service ps homer
docker service rm homer

Appendix A. Swarm Network

- Overlay Multi-Host Networking
 - Just choose --driver overlay when creating network
 - For container-to-container traffic inside a single Swarm
 - Optional IPSec encryption on network creation
 - Each service can be connected to multiple networks
- Routing Mesh
 - Routes ingress (incoming) packets for a Service to proper Task
 - Spans all nodes in Swarm
 - Uses IPVS from Linux Kernel
 - Load balances Swarm Services across their Tasks

Overlay Network



Routing Mesh

- This works Two ways:
 - Container-to-container in a Overlay network (uses VIP)
 - External traffic incoming to published ports (all nodes listen)
- This is stateless load balancing
- This LB is at Layer 3, not Layer 4
- Both limitation can be overcome with:
 - Nginx or HAProxy LB proxy, or:
 - Docker Enterprise Edition, which comes with built-in L4 web proxy

Practice: Overlay Network

- Create an overlay network "mydrupal" and start 2 service: drupal and postgres. After you start check on all: curl http://localhost

docker network create --driver overlay mydrupal

docker network ls

docker service create --name psql --network mydrupal -e POSTGRES_PASSWORD=mypass postgres

docker service ls

docker service ps psql

docker container logs psql<Tab>

docker service create --name drupal --network mydrupal -p 80:80 drupal

docker service ls

docker service ps drupal

docker service inspect drupal

Practice: Routing Mesh

- Create a search app elasticsearch 3 replicas, each container has different initial string. Run curl <http://localhost:9200> on different nodes. Observe it doesn't matter which node you run, always load balancing on existing nodes

```
docker service create --name search --replicas 3 -p 9200:9200 elasticsearch:2
```

```
docker service ps search
```

```
Node2> curl http://localhost:9200
```

```
Node3> curl http://localhost:9200
```

```
Node1> watch curl http://localhost:9200
```

Appendix B. Stack and Secret

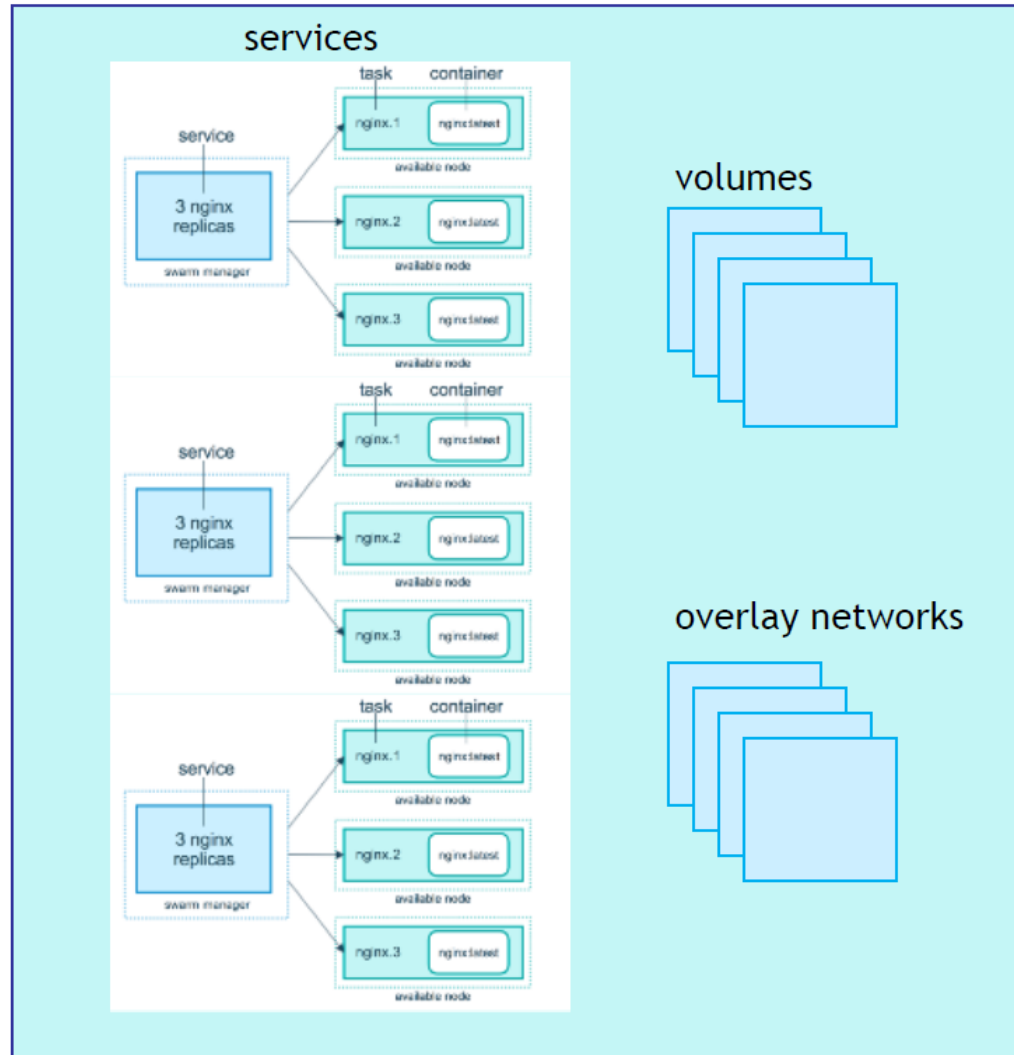
- In 1.13 Docker adds a new layer of abstraction to Swarm called Stacks
- Stacks accept Compose files as their declarative definition for services, networks, and volumes
- Use **docker stack deploy** rather than **docker service create**
- Stacks manages all those objects for us, including overlay network per stack.
- New **deploy**: key in Compose file. Can't do **build**:
- Compose now ignores **deploy**:, Swarm ignores **build**:
- **docker-compose** cli not needed on Swarm server

Docker Stack vs Docker Compose

- Conceptually, both files serve the same purpose - deployment and configuration of your containers on docker engines.
- Think docker-compose for developer tool on your local machine and docker stack as deployment tool on Swarm.
- Docker-compose tool was created first and its purpose is "for defining and running multi-container Docker applications" on a single docker engine.
- You use docker-compose up to create/update your containers, networks, volumes and so on.
- Where Docker Stack is used in Docker Swarm (Docker's orchestration and scheduling tool) and, therefore, it has additional configuration parameters (i.e. replicas, deploy, roles) that are not needed on a single docker engine.
- The stack file is interpreted by docker stack command. This command can be invoked from a docker swarm manager only
- Specify a group of Docker containers to configure and deploy two ways:
 - Docker compose (**docker-compose up**)
 - Docker swarm (**docker swarm init; docker stack deploy --compose-file docker-stack.yml mystack**)

Stack

stack!



Secret Storage

- Easiest "secure" solution for storing secrets in Swarm
- What is a Secret?
 - Usernames and passwords
 - TLS certificates and keys
 - SSH keys
 - Supports generic strings or binary content up to 500kb
- As of Docker 1.13.0 Swarm Raft DB is encrypted on disk
- Only stored on disk on Manager nodes
- Default is Managers and Workers "control plane" is TLS + Mutual Auth
- Secrets are first stored in Swarm, then assigned to a Service(s)
- Only containers in assigned Service(s) can see them

Practice: Secrets

- Let's create secrets on the command line for postgres service and then do the same for stack in the yaml file. Do it on the swarm manager node.
 - Two ways to create: Use file or command line
- ```
echo "mypsquuser" > psqu_user.txt
docker secret create psqu_user psqu_user.txt
echo "mysecretpass123" | docker secret create psqu_pass -
docker secret ls
docker secret inspect psqu_user
docker service create --name psqu --secret psqu_user --secret psqu_pass
-e POSTGRES_PASSWORD_FILE=/run/secrets/psqu_pass -e
POSTGRES_USER_FILE=/run/secrets/psqu_user postgres
docker service ps psqu (Learn node and docker container ls)
docker exec -it psqu.1.<CONNAME> bash
cat /run/secrets/psqu_user; cat /run/secrets/psqu_pass; exit
docker service rm psqu
```



# Practice: Secrets

- Now let's copy docker-compose.yml psql\_password.txt psql\_user.txt to one of the manager node with drag and drop

version: "3.1"

services:

psql:

image: postgres

secrets:

- psql\_user
- psql\_password

environment:

POSTGRES\_PASSWORD\_FILE: /run/secrets/psql\_password

POSTGRES\_USER\_FILE: /run/secrets/psql\_user

secrets:

psql\_user:

file: ./psql\_user.txt

psql\_password:

file: ./psql\_password.txt

## Practice: Secrets

- Now we can deploy our db service with stack using secret. Note that we need to use yaml version 3.1 for secrets. Also for stacks version should be at least 3.

**docker stack deploy -c docker-compose.yml mydb**

**docker secret ls**

**docker stack ls**

**docker service ls**

**docker service ps mydb\_mysql**

**docker stack rm mydb**

# Practice: Voting App Stack Example

- Let's create and run full swarm stack app designed as an example by Docker. You can check details:

<https://github.com/dockersamples/example-voting-app>

- First open <https://labs.play-with-docker.com/> and create 5 node Managers using template just to avoid manual swarm setup we have done earlier
- On Manager1 explore Swarm and Copy voting.yml file from your local machine to Manager1 with Drag and Drop

```
cat voting.yml
```

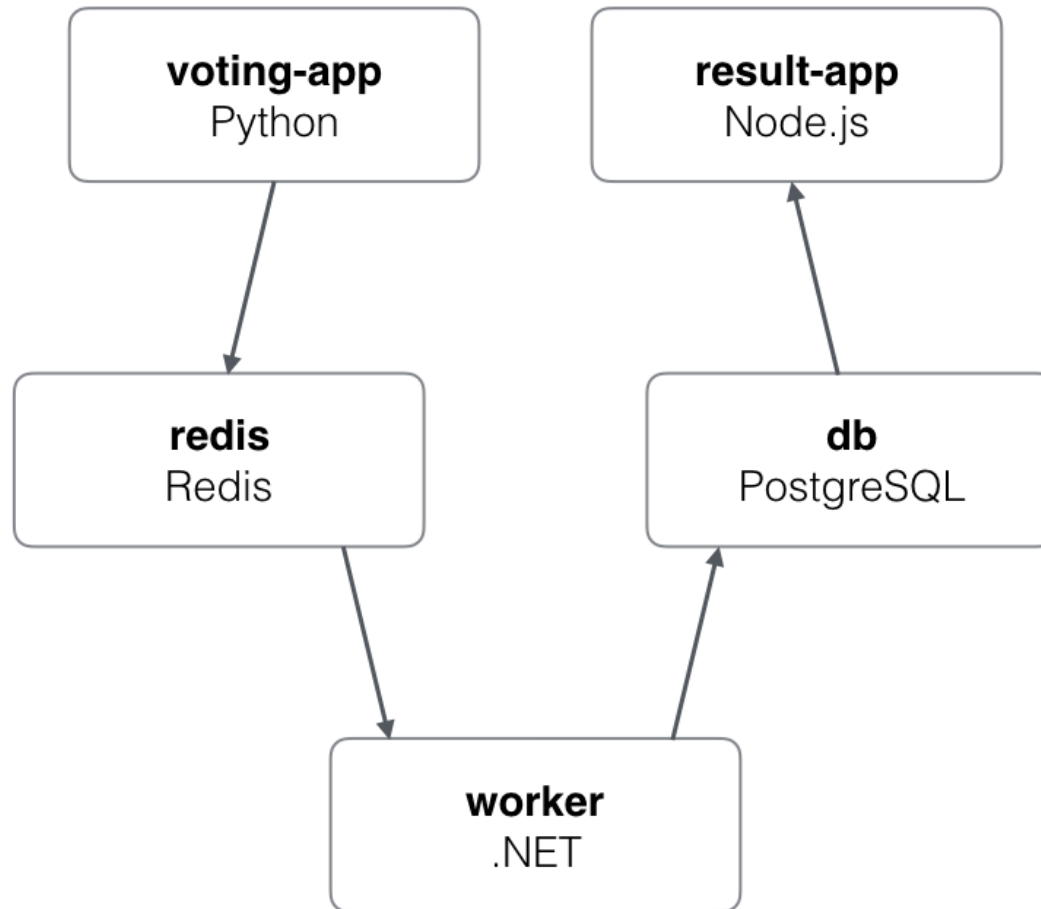
```
docker node ls
```

```
docker service ls
```

```
docker stack ls
```

```
docker stack deploy -c voting.yml voteapp
```

# Practice: Voting App Stack Example



# Practice: Voting App Stack Example

- On Manager1 explore voting app
  - First you see ports running 5000, 5001, 8080
  - Open Chrome first on 5000 to vote
  - Check result on 5001. Open firefox and vote again
  - Finally look 8080 visualizer to see which service is running on which node

**docker stack ls**

**docker stack ps voteapp**

**docker stack services voteapp**

**docker network ls**

- Now change voting.yml and change vote replicas to 5. Deploy again (it will update) Finally look at the visualizer

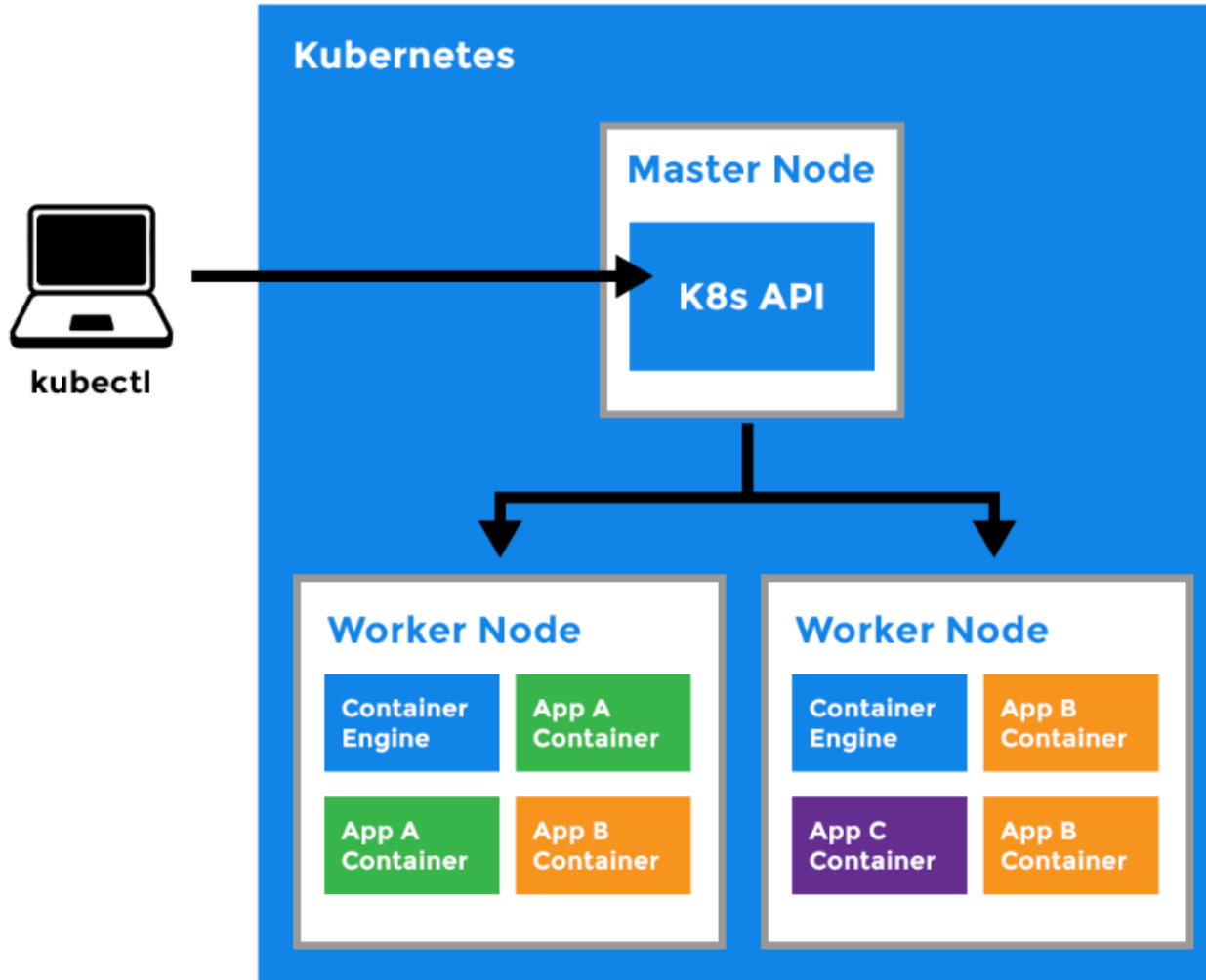
**docker stack deploy -c voting.yml voteapp**

# Appendix C. Kubernetes

- Greek for pilot or **Helmsman** of a ship
- Project started at Google as an open source container orchestration platform
- Successor of **Borg** and **Omega** projects created at Google.
- All services within Kubernetes natively Load Balanced
- Autoscale Workloads
- Blue/Green Deployments
- Released v1.0 at July 2015. Cloud Native Computing Foundation (CNCF) serves as the **vendor-neutral** home for many of the projects on GitHub, including Kubernetes, Prometheus and Envoy



# Kubernetes - Key Concepts



# Kubernetes - Key Concepts

## ■ **kubectl**

- Command-line program for interacting with the Kubernetes API, to control the Kubernetes cluster

## ■ **Master Node**

- The main machine that controls the nodes
- Main entrypoint for all administrative tasks
- It handles the orchestration of the worker nodes

## ■ **Worker Node**

- It is a worker machine in Kubernetes (used to be known as minion)
- This machine performs the requested tasks. Each Node is controlled by the Master Node
- Runs containers inside pods
- This is where the Docker engine runs and takes care of downloading images and starting containers



# Kubernetes - Key Concepts

## ■ **Kubernetes Pod**

- A Pod can host one (or more) containers
- Pods are smallest deployable units in Kubernetes that can be created scheduled and managed.
- Pods are scheduled to Nodes. Pod contains containers and volumes. Containers in a same Pod share the same network namespace and can communicate with each other
- Pods are instances of Deployments. One Deployment can have multiple pods
- With Horizontal Pod Autoscaling, Pods of a Deployment can be automatically started and halted based on usage
- Each Pod has its unique IP Address within the cluster
- Any data saved inside the Pod will disappear without a persistent storage

# Kubernetes - Key Concepts

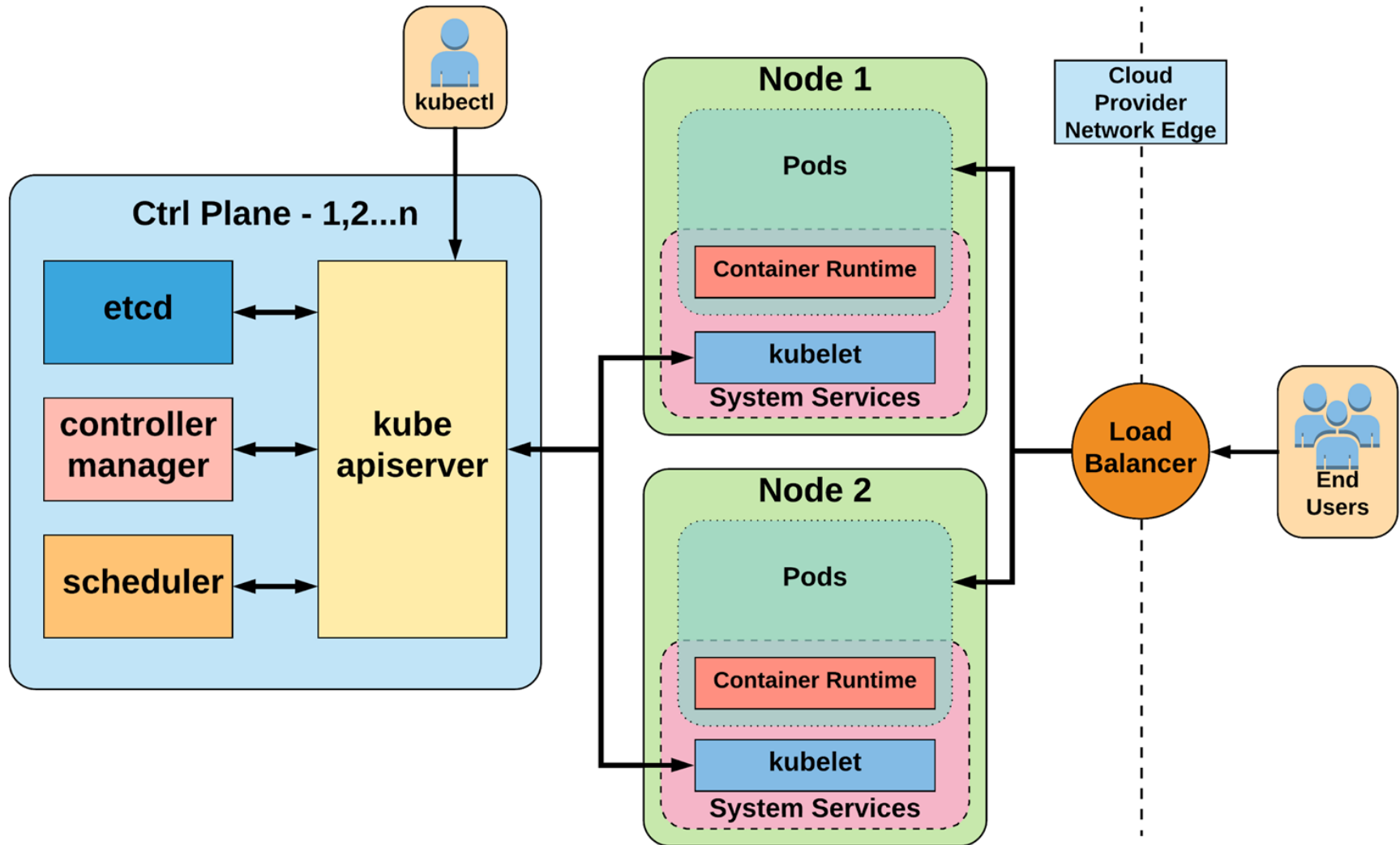
## ■ **Service:**

- A service is responsible for making our Pods discoverable inside the network or exposing them to the internet
- A Service identifies Pods by its LabelSelector

## ■ **Deployment**

- A deployment is a blueprint for the Pods to be create
- Handles update of its respective Pods.
- A deployment will create and keep the Pods running and update them by it's spec from the template.
- Pod(s) resource usage can be specified in the deployment.
- Deployment can scale up replicas of Pods.

# Kubernetes - Architecture



# Control Plane Components

- **kube-apiserver**
  - Provides a forward facing REST interface into the kubernetes control plane and datastore. All clients and other applications interact with kubernetes strictly through the API Server
- **etcd**
  - etcd acts as the cluster datastore.
  - Purpose in relation to Kubernetes is to provide a strong, consistent and highly available key-value store for persisting cluster state.
- **kube-controller-manager**
  - Serves as the primary daemon that manages all core components.
  - Monitors the cluster state via the apiserver and steers the cluster towards the desired state
- **cloud-controller-manager**
  - Runs controllers that interact with cloud providers. Think cloud *interface*
- **kube-scheduler**
  - Selects the Nodes for new Pods

# Node Components

## ■ **kubelet**

- Acts as the node agent responsible for managing the lifecycle of every pod on its host.
- Kubelet understands YAML container manifests that it can read from several sources:
  - ✓ File path, HTTP Endpoint, etcd

## ■ **kube-proxy**

- Manages the network rules on each node.
- Performs connection forwarding or load balancing for Kubernetes cluster services.

## ■ **Container Runtime Engine**

- A container runtime is a CRI (Container Runtime Interface) compatible application that executes and manages containers.
- Containerd (docker)



