Docker and Kubernetes

Lesson Objectives

- Session 1 – Getting started with Docker
- Session 2 – Building Docker images
- Session 3 – Docker Networking
- Session 4 – Docker storage
- Session 5 – Making a real project with Docker and NodeJs
- Session 6 – Docker Swarm
- Session 7-10 – Kubernetes
- Session 11 – Final test

Session 1

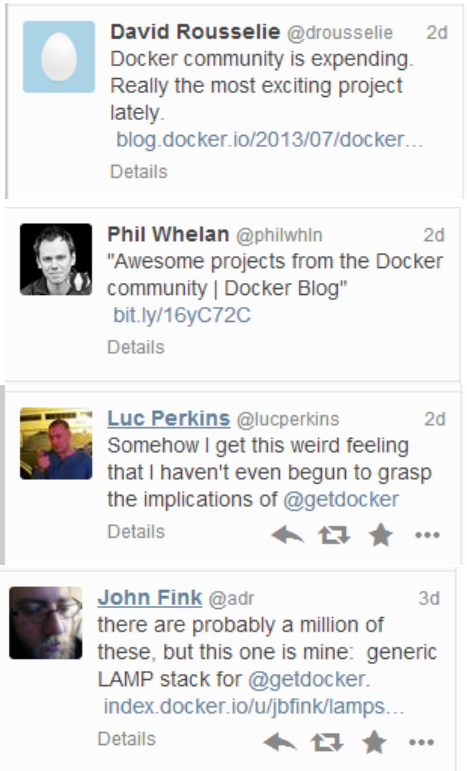
Getting started with Docker

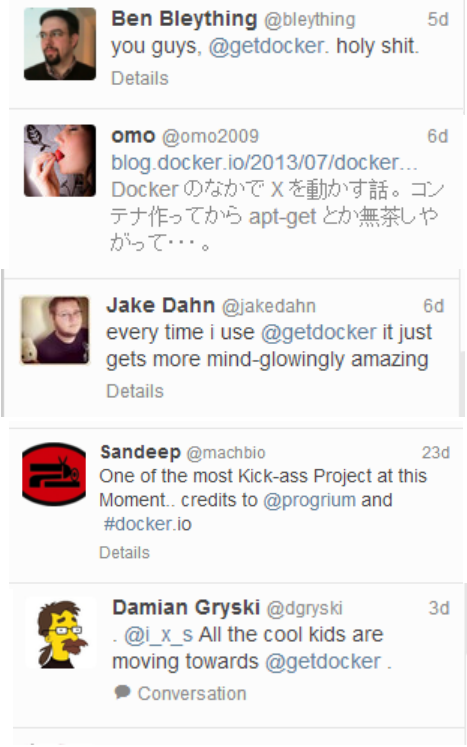
Agenda

- Introduction
- Docker Overview
- Getting Started with Docker
- Basic commands

In the 8 months since we launched

- >200,000 pulls
- >7,500 github stars
- >200 significant contributors
- >200 projects built on top of docker
  - ✓ UIs, mini-PaaS, Remote Desktop….
- 1000's of Dockerized applications
  - ✓ Memcached, Redis, Node.js…and Hadoop
- Integration in Jenkins, Travis, Chef, Puppet, Vagrant and OpenStack
- Meetups arranged around the world…with organizations like Ebay, Cloudflare, Yandex, and Rackspace presenting on their use of Docker





WHY ALL THE EXCITEMENT?

The Challenge

Multiplicity of Stacks

Static website

nginx 1.5 + modsecurity + openssl + bootstrap 2

User DB

postgresql + pgv8 + v8

Queue

Redis + redis-sentinel

Analytics DB

hadoop + hive + thrift + OpenJDK

Background workers

Python 3.0 + celery + pyredis + libcurl + ffmpeg + libopencv + nodejs + phantomjs

Web frontend

Ruby + Rails + sass + Unicorn

API endpoint

Python 2.7 + Flask + pyredis + celery + psycopg + postgresql-client

Do services and apps interact appropriately?

Multiplicity of hardware environments

Development VM

QA server

Customer Data Center

Public Cloud

Disaster recovery

Production Servers

Production Cluster

Contributor's laptop

Can I migrate smoothly and quickly?

The Matrix From Hell

	Development VM	QA Server	Single Prod Server	Onsite Cluster	Public Cloud	Contributor's laptop	Customer Servers
Static website	?	?	?	?	?	?	?
Web frontend	?	?	?	?	?	?	?
Background workers	?	?	?	?	?	?	?
User DB	?	?	?	?	?	?	?
Analytics DB	?	?	?	?	?	?	?
Queue	?	?	?	?	?	?	?

Cargo Transport Pre-1960

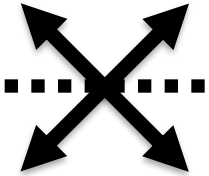
Multiplicity of Goods





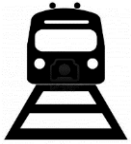


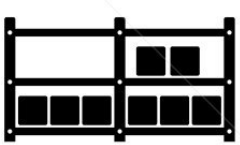
Do I worry about how goods interact (e.g. coffee beans next to spices)



Multiplicity of methods for transporting/storing

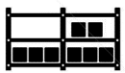





Can I transport quickly and smoothly (e.g. from boat to train to truck)

Also a matrix from hell

	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
							

Solution: Intermodal Shipping Container

Multiplicity of Goods





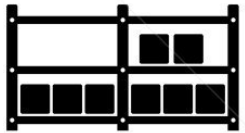


Do I worry about how goods interact (e.g. coffee beans next to spices)

A standard container that is loaded with virtually any goods, and stays sealed until it reaches final delivery.

…in between, can be loaded and unloaded, stacked, transported efficiently over long distances, and transferred from one mode of transport to another

Multiplicity of methods for transporting/storing




Can I transport quickly and smoothly (e.g. from boat to train to truck)

Docker is a shipping container system for code

Multiplicity of Stacks

Static website

User DB

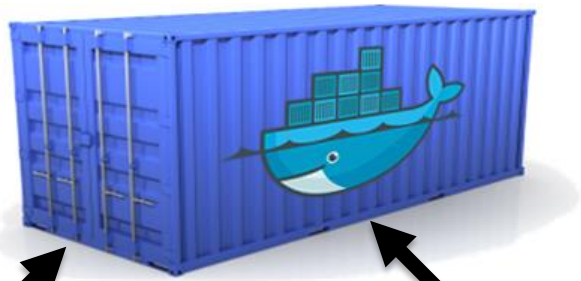
Web frontend

Queue

Analytics DB

Do services and apps interact appropriately?

An engine that enables any payload to be encapsulated as a lightweight, portable, self-sufficient container…



…that can be manipulated using standard operations and run consistently on virtually any hardware platform

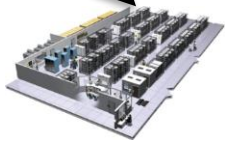
Multiplicity of hardware environments

Can I migrate smoothly and quickly

Development VM

QA server

Customer Data Center

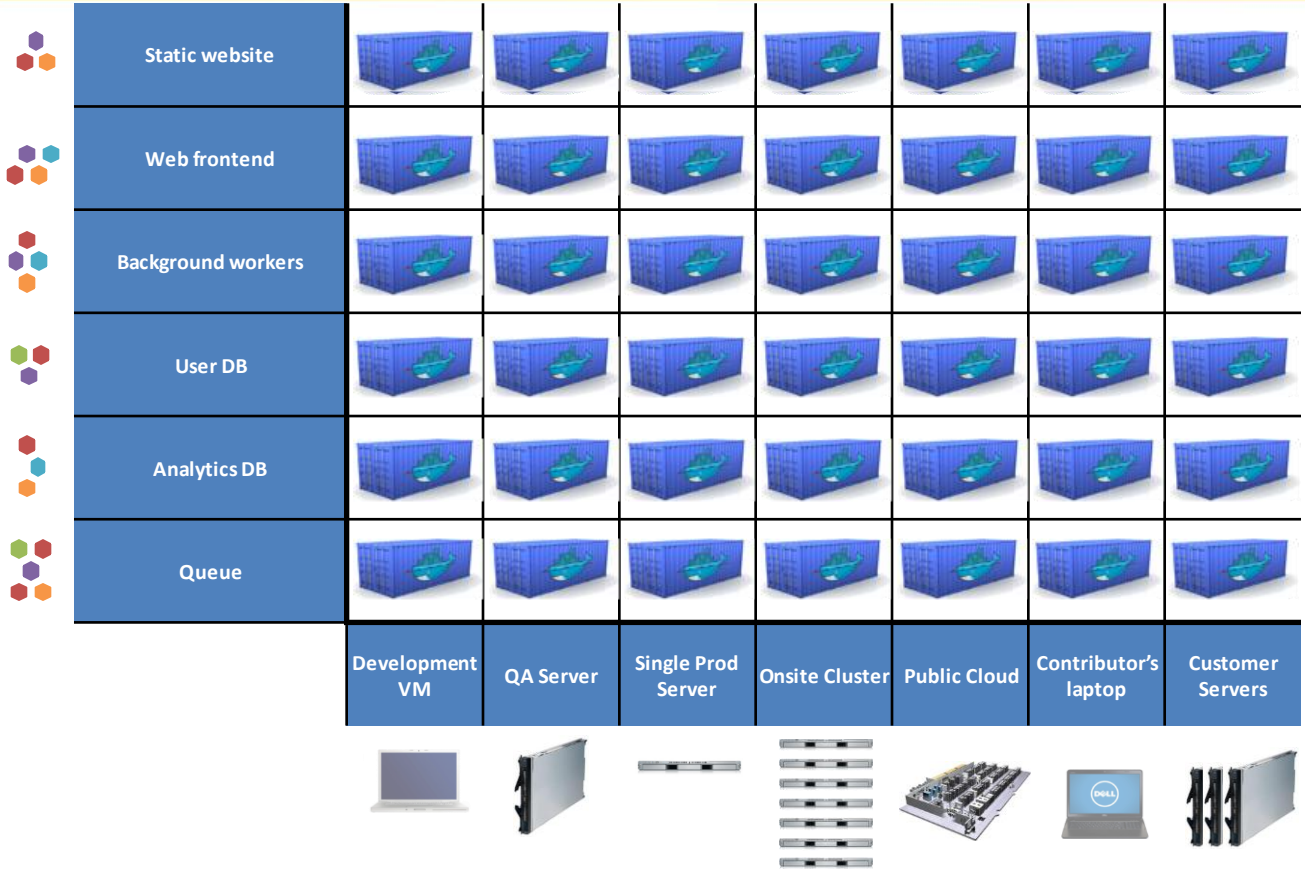


Public Cloud

Production Cluster

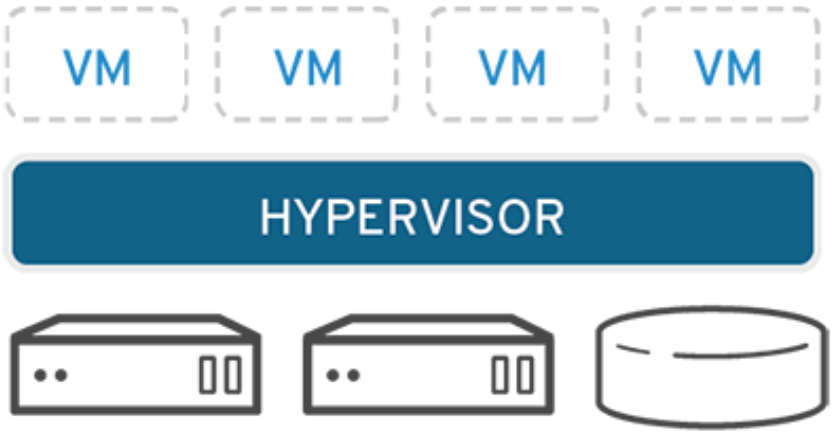
Contributor's laptop

Docker eliminates the matrix from Hell



Virtualization

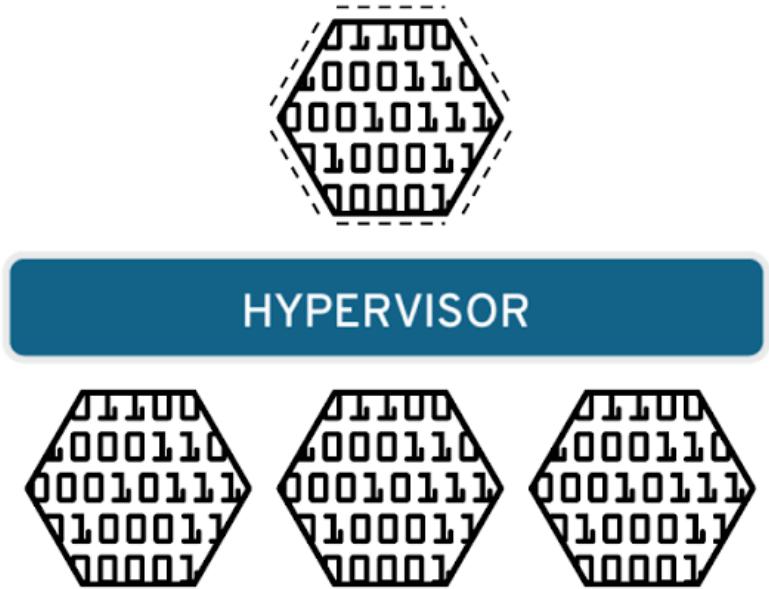
Virtualization is technology that lets you create useful IT services using resources that are traditionally bound to hardware. It allows you to use a physical machine's full capacity by distributing its capabilities among many users or environments.



Types of virtualization

Data virtualization

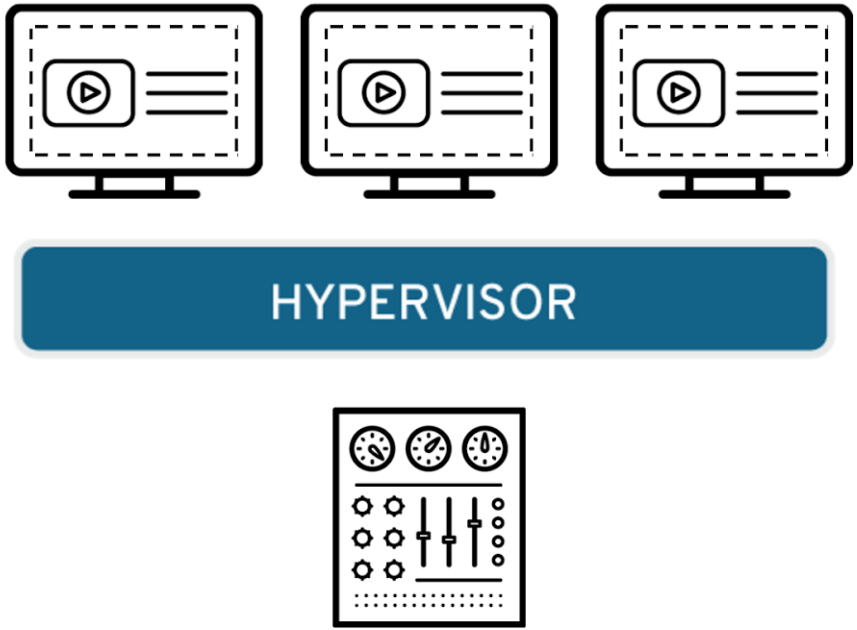
Data that's spread all over can be consolidated into a single source. Data virtualization allows companies to treat data as a dynamic supply—providing processing capabilities that can bring together data from multiple sources, easily accommodate new data sources, and transform data according to user needs



Types of virtualization

Desktop virtualization

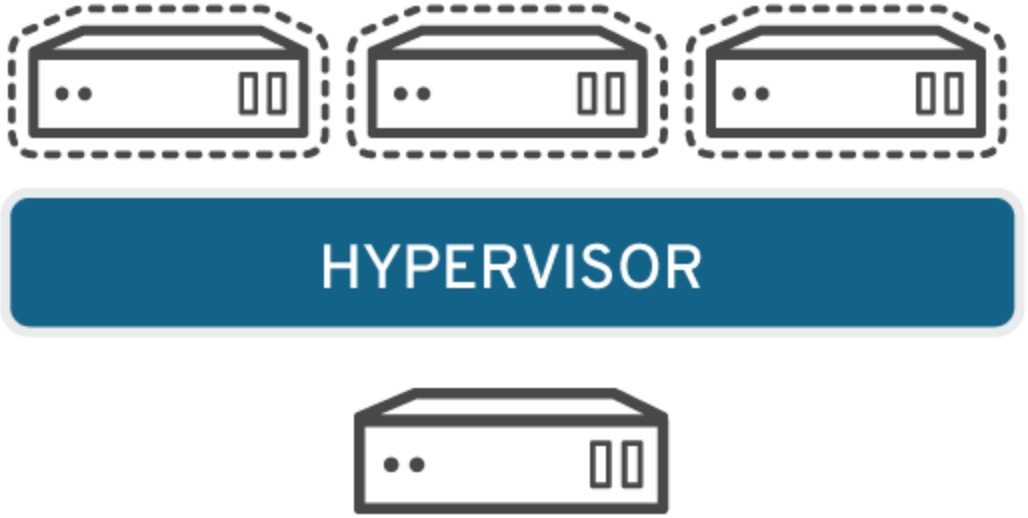
Easily confused with operating system virtualization—which allows you to deploy multiple operating systems on a single machine—desktop virtualization allows a central administrator (or automated administration tool) to deploy simulated desktop environments to hundreds of physical machines at once



Types of virtualization

Server virtualization

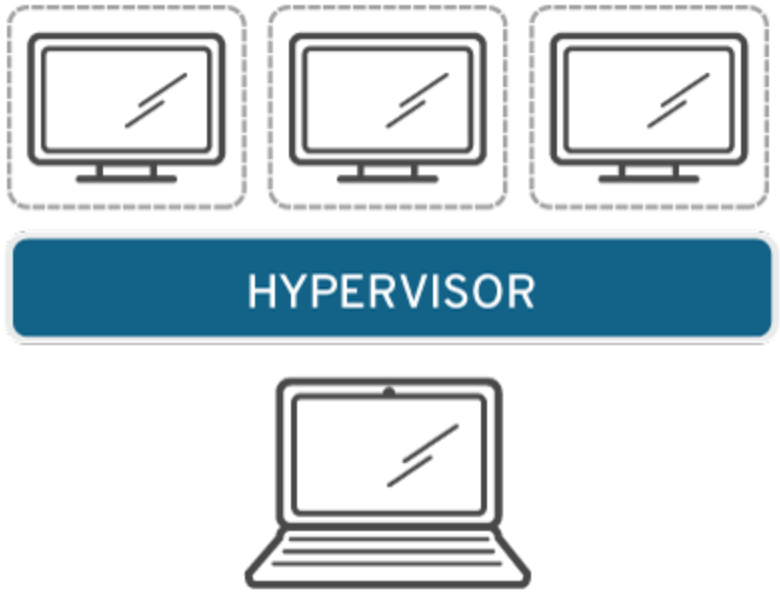
Servers are computers designed to process a high volume of specific tasks really well so other computers—like laptops and desktops—can do a variety of other tasks. Virtualizing a server lets it to do more of those specific functions and involves partitioning it so that the components can be used to serve multiple functions.



Types of virtualization

Operating system virtualization

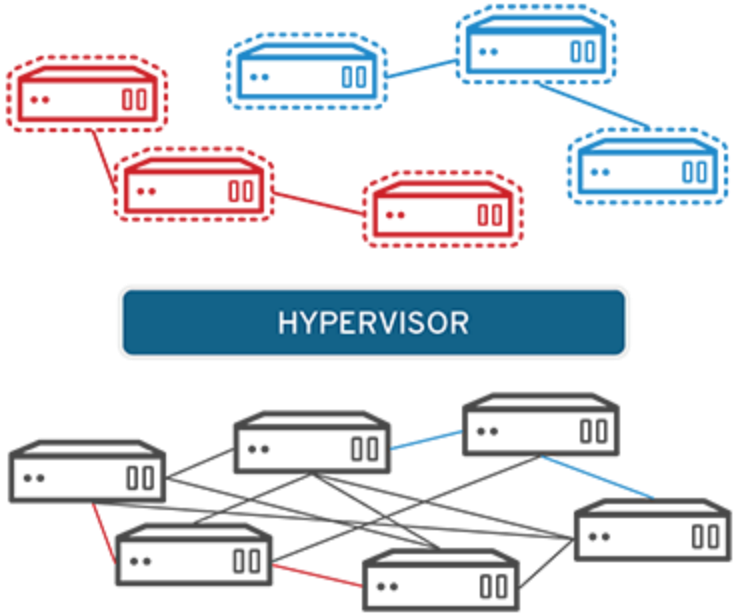
Operating system virtualization happens at the kernel—the central task managers of operating systems. It's a useful way to run Linux and Windows environments side-by-side



Types of virtualization

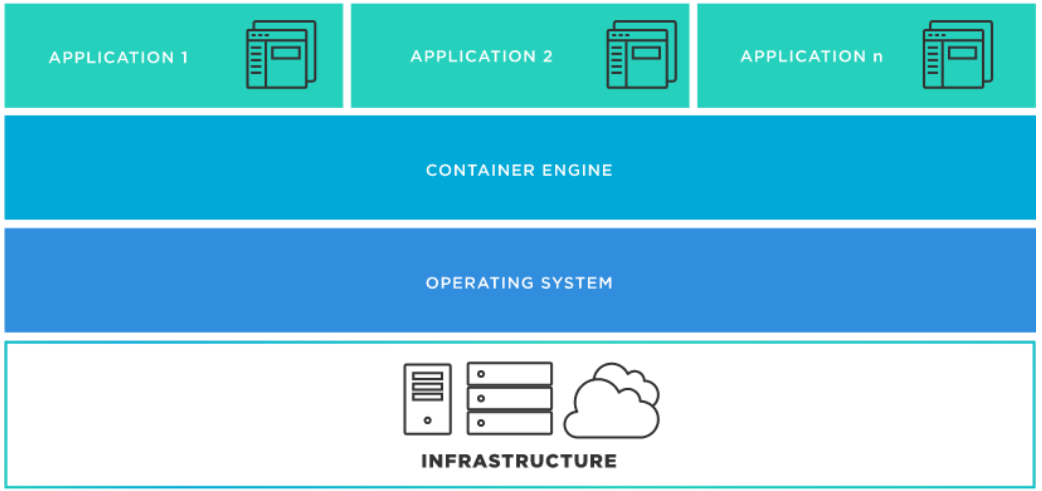
Network functions virtualization

Network functions virtualization (NFV) separates a network's key functions (like directory services, file sharing, and IP configuration) so they can be distributed among environments.

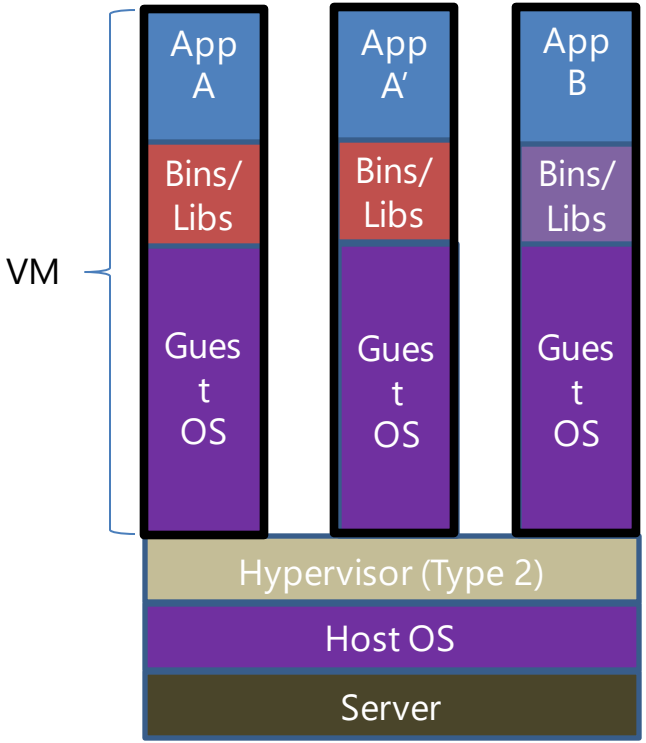


What is containerization?

Containerization is the packaging of software code with just the operating system (OS) libraries and dependencies required to run the code to create a single lightweight executable—called a container—that runs consistently on any infrastructure

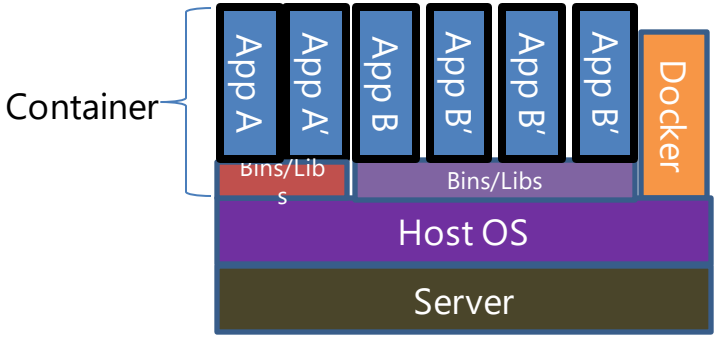


Containers vs. VMs



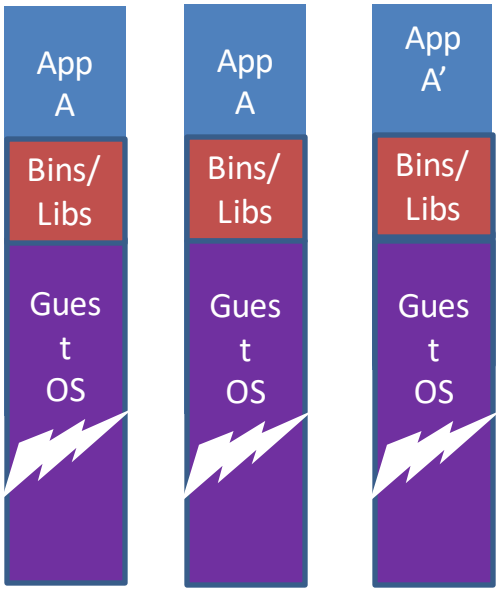
Containers are isolated, but share OS and, where appropriate, bins/libraries

...result is significantly faster deployment, much less overhead, easier migration, faster restart



Why are Docker containers lightweight?

VMs



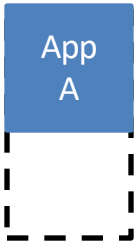
VMs

Every app, every copy of an app, and every slight modification of the app requires a new virtual server

Containers

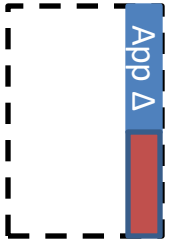
Original App

(No OS to take up space, resources, or require restart)



Copy of App

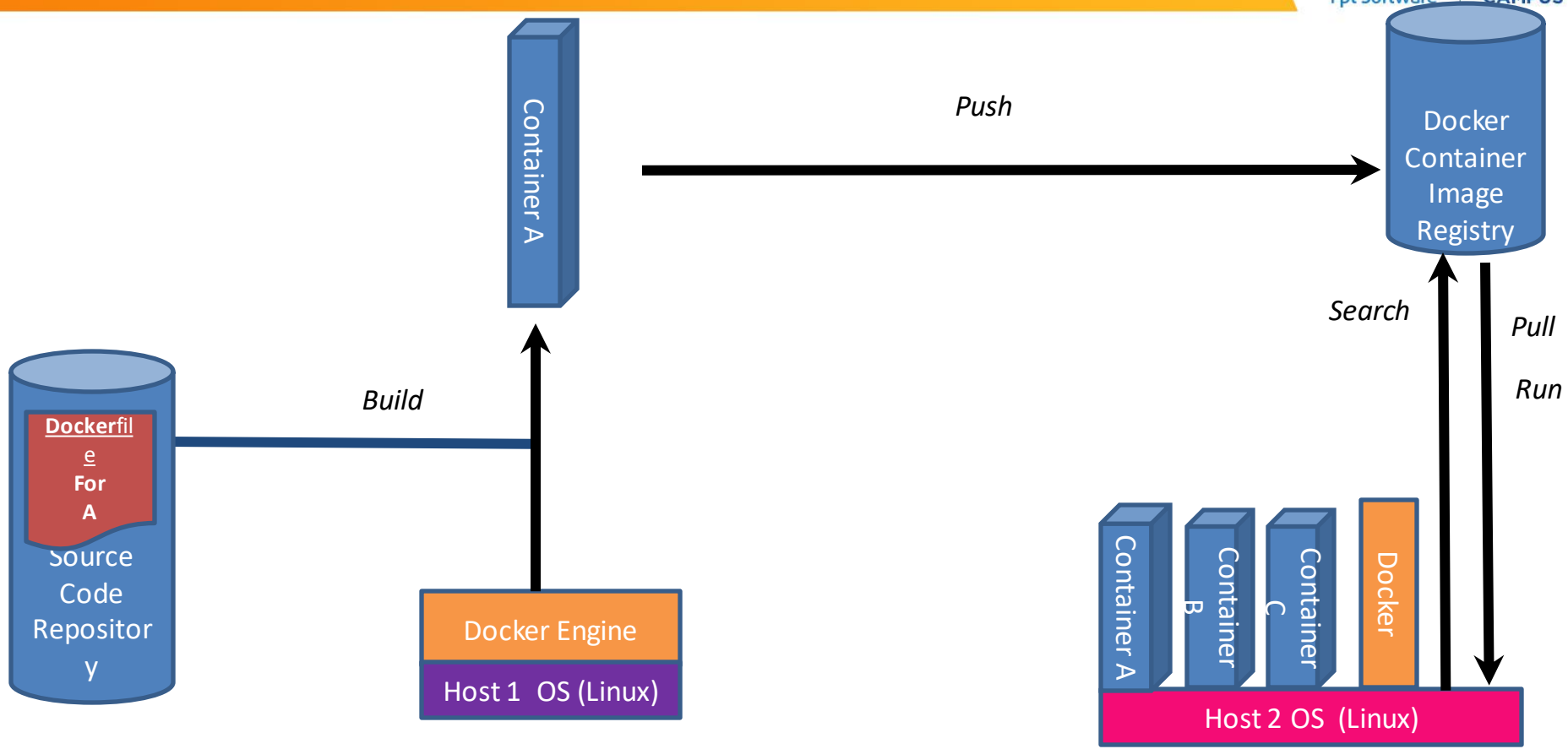
No OS. Can Share bins/libs



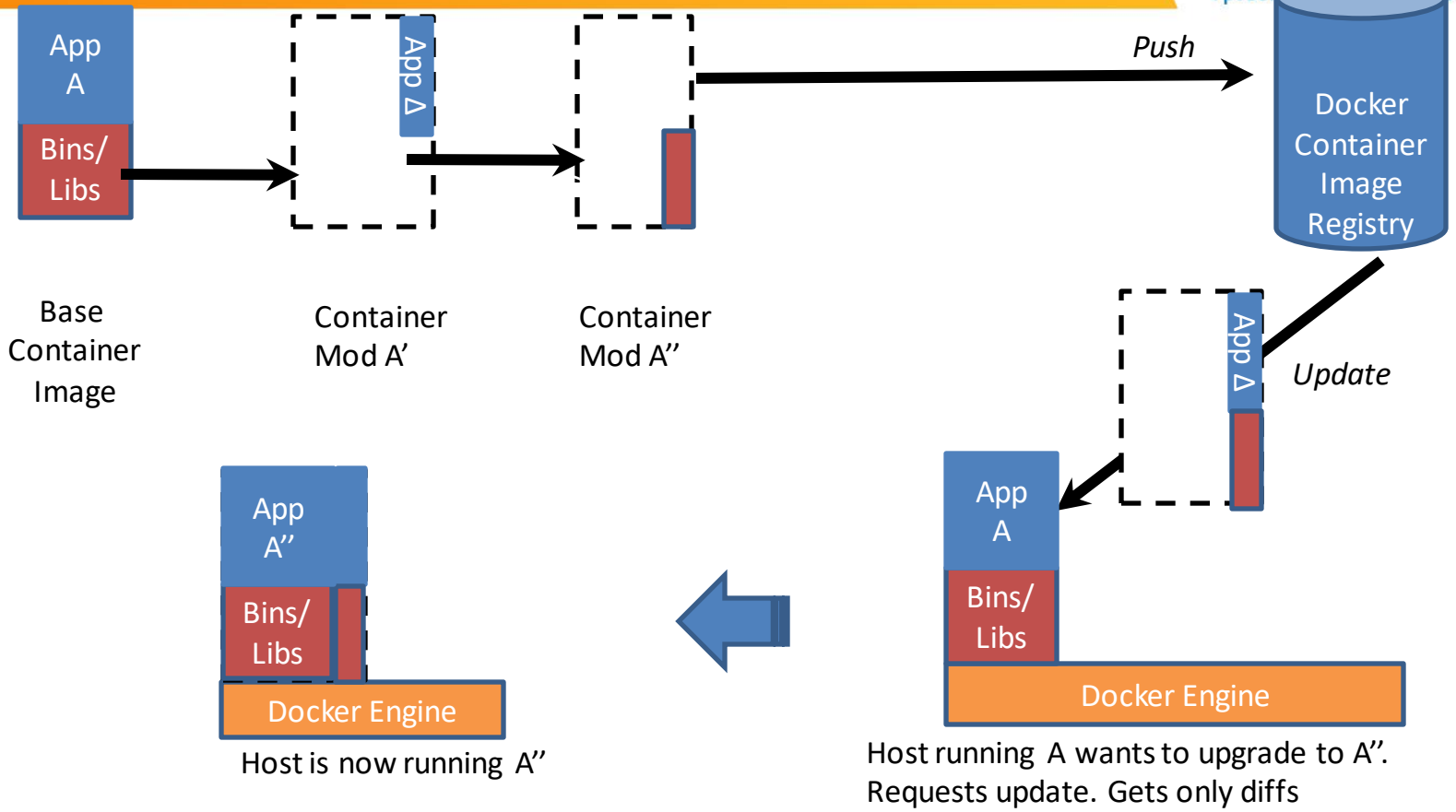
Modified App

Copy on write capabilities allow us to only save the diffs Between container A and container A'

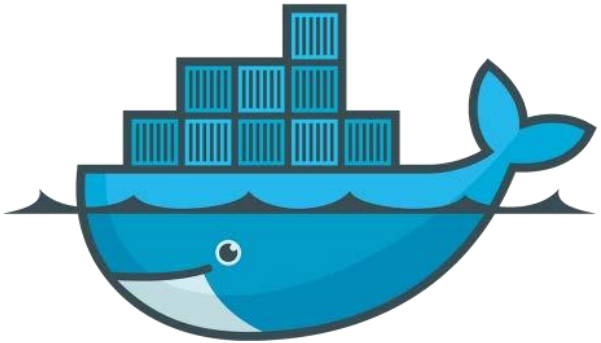
What are the basics of the Docker system?



Changes and Updates

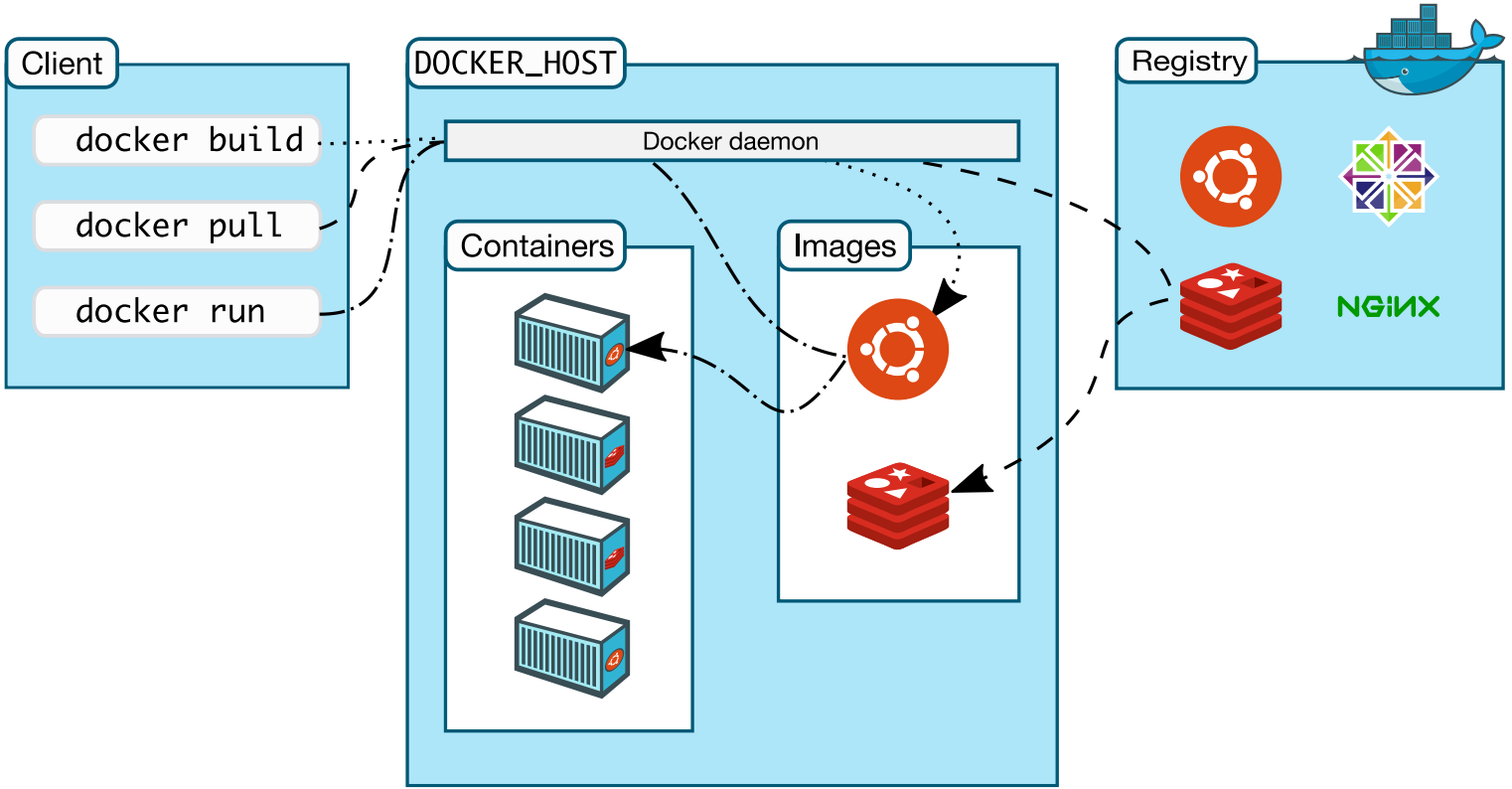


What Is Docker?

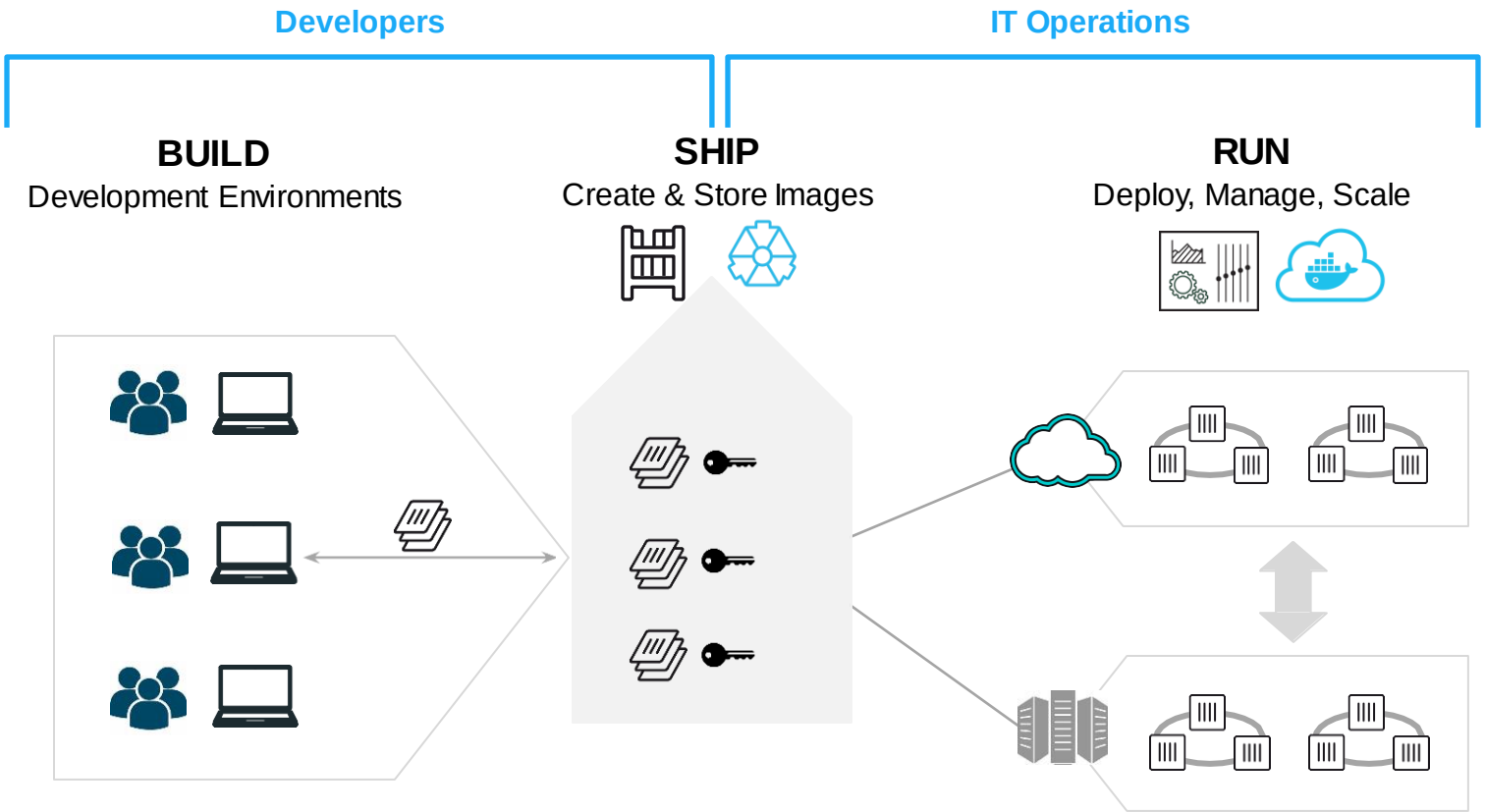


- **Lightweight, open, secure platform**
- **Simplify building, shipping, running apps**
- Runs natively on Linux or Windows Server
- Runs on Windows or Mac Development machines (with a virtual machine)
- Relies on "images" and "containers"

Docker architecture



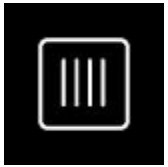
Using Docker: Build, Ship, Run Workflow



Some Docker vocabulary

Docker Image

The basis of a Docker container. Represents a full application



Docker Container

The standard unit in which the application service resides and executes

Docker Engine

Creates, ships and runs Docker containers deployable on a physical or virtual, host locally, in a datacenter or cloud service provider

28

Registry Service (Docker Hub(Public) or Docker Trusted Registry(Private))

Cloud or server based storage and distribution service for your images

Basic Docker Commands

```
$ docker image pull node:latest

$ docker image ls
$ docker container run –d –p 5000:5000 --name node node:latest

$ docker container ps

$ docker stop <container id>

$ docker rm <container id>

$ docker image rmi (or <image id>)

$ docker build –t node:2.0 .

$ docker image push node:2.0

$ docker --help
```

Q&A

Session 2

Building Docker images

Agenda

- Dockerfile
- Build parameter
- Environment variables
- CMD and ENTRYPOINT
- Practice building images

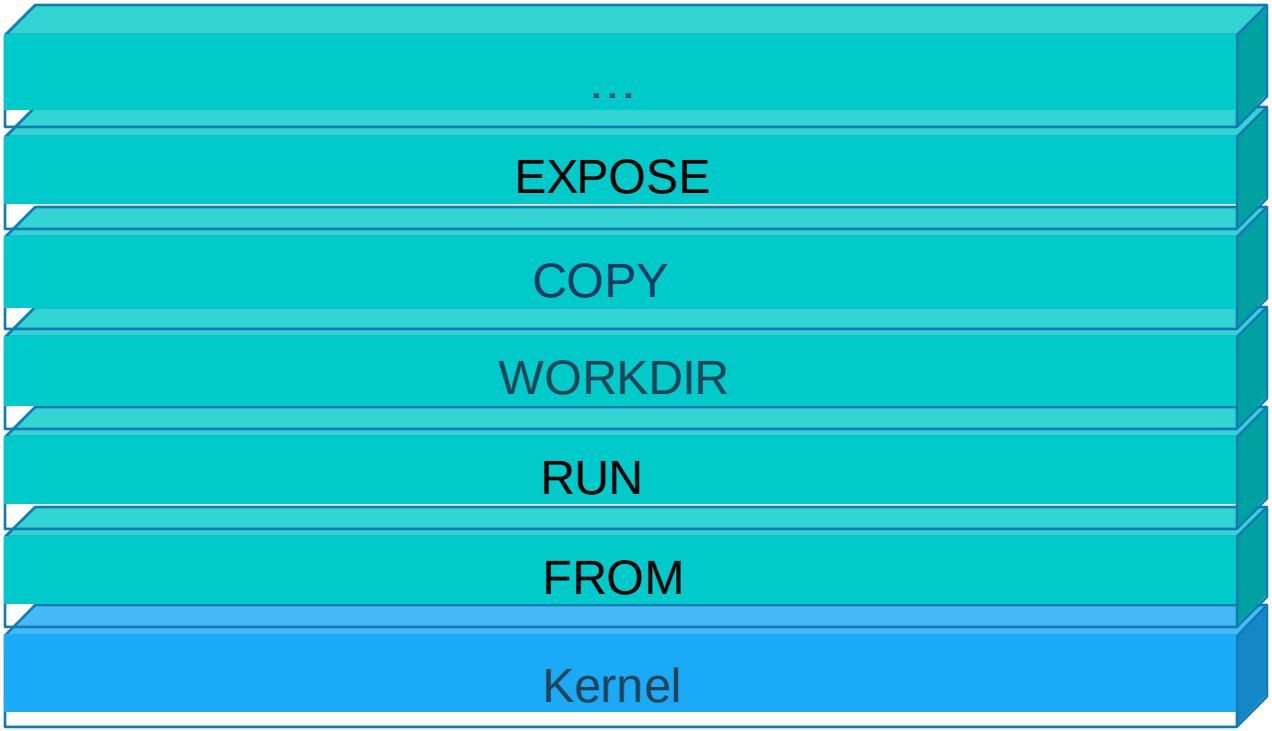
Dockerfile – Linux Example

Dockerfile

```
# Create image based on the official Node 6 image from dockerhub
FROM node:latest

# Create a directory where our app will be placed
RUN mkdir -p /usr/src/app

# Change directory so that our commands run inside this new directory
WORKDIR /usr/src/app

# Copy dependency definitions
COPY package.json /usr/src/app

# Install dependecies
RUN npm install

# Get all the code needed to run the app
COPY . /usr/src/app

# Expose the port the app runs in
EXPOSE 4200

# Serve the app
CMD ["npm", "start"]
```

- Instructions on how to build a Docker image
- Looks very similar to “native” commands
- Important to optimize your Dockerfile

Each Dockerfile Command Creates a Layer



Docker Image Pull: Pulls Layers

```
Alexander@DESKTOP-90ATKET MINGW64 ~/Docker/Demo
$ docker pull nginx:latest
latest: Pulling from library/nginx
bc95e04b23c0: Pull complete
f3186e650f4e: Pull complete
9ac7d6621708: Pull complete
Digest: sha256:b81f317384d7388708a498555c28a7cce778a8f291d90021208b3eba3fe74887
Status: Downloaded newer image for nginx:latest
```

Dockerfile format

Here is the format of the Dockerfile:

```
# Comment
INSTRUCTION arguments
```

The instruction is not case-sensitive. However, convention is for them to be UPPERCASE to distinguish them from arguments more easily.

Docker runs instructions in a Dockerfile in order. A Dockerfile must begin with a FROM instruction. This may be after parser directives, comments, and globally scoped ARGs

Dockerfile format

Here is the format of the Dockerfile:

```
# Comment
INSTRUCTION arguments
```

The instruction is not case-sensitive. However, convention is for them to be UPPERCASE to distinguish them from arguments more easily.

Docker runs instructions in a Dockerfile in order. A Dockerfile must begin with a FROM instruction. This may be after parser directives, comments, and globally scoped ARGs

FROM

The **FROM** instruction initializes a new build stage and sets the Base Image for subsequent instructions. As such, a valid Dockerfile must start with a **FROM** instruction. The image can be any valid image – it is especially easy to start by pulling an image from the Public Repositories.

FROM can appear multiple times within a single Dockerfile to create multiple images or use one build stage as a dependency for another. Simply make a note of the last image ID output by the commit before each new **FROM** instruction. Each **FROM** instruction clears any state created by previous instructions.

Understand how ARG and FROM interact

FROM instructions support variables that are declared by any ARG instructions that occur before the first **FROM**.

```
ARG CODE_VERSION=latest
FROM base:${CODE_VERSION}
CMD /code/run-app
FROM extras:${CODE_VERSION}
CMD /code/run-extras
```

An **ARG** declared before a **FROM** is outside of a build stage, so it can't be used in any instruction after a **FROM**. To use the default value of an **ARG** declared before the first **FROM** use an **ARG** instruction without a value inside of a build stage:

```
ARG VERSION=latest
FROM busybox:$VERSION
ARG VERSION
RUN $VERSION > image_version
```

RUN

RUN has 2 forms:

- **RUN** <command> (shell form, the command is run in a shell, which by default is /bin/sh -c on Linux or cmd /S /C on Windows)
- **RUN** ["executable", "param1", "param2"] (exec form)

The default shell for the shell form can be changed using the SHELL command.

```
RUN ["/bin/bash", "-c", "echo hello"]
```

WORKDIR

```
WORKDIR /path/to/workdir
```

The **WORKDIR** instruction sets the working directory for any **RUN, CMD, ENTRYPOINT, COPY** and **ADD** instructions that follow it in the Dockerfile. If the **WORKDIR** doesn't exist, it will be created even if it's not used in any subsequent Dockerfile instruction.

The WORKDIR instruction can be used multiple times in a Dockerfile. If a relative path is provided, it will be relative to the path of the previous WORKDIR instruction

```
WORKDIR /a
WORKDIR b
WORKDIR c
RUN pwd
```

COPY

COPY has two forms:

```
COPY [--chown=<user>:<group>] <src>... <dest>
COPY [--chown=<user>:<group>] ["<src>",... "<dest>"]
```

The COPY instruction copies new files or directories from <src> and adds them to the filesystem of the container at the path <dest>.

Multiple <src> resources may be specified but the paths of files and directories will be interpreted as relative to the source of the context of the build.

EXPOSE

The **EXPOSE** instruction informs Docker that the container listens on the specified network ports at runtime. You can specify whether the port listens on TCP or UDP, and the default is TCP if the protocol is not specified.

By default, **EXPOSE** assumes TCP. You can also specify UDP:

```
EXPOSE 80/udp
```

ENV

The **ENV** instruction sets the environment variable <key> to the value <value>.

The value will be interpreted for other environment variables, so quote characters will be removed if they are not escaped. Like command line parsing, quotes and backslashes can be used to include spaces within values

```
ENV MY_NAME="John Doe"
ENV MY_DOG=Rex\ The\ Dog
ENV MY_CAT=fluffy
```

CMD

The **CMD** instruction has three forms:

- CMD ["executable","param1","param2"] (exec form, this is the preferred form)
- CMD ["param1","param2"] (as default parameters to ENTRYPOINT)
- CMD command param1 param2 (shell form)

ENTRYPOINT

An **ENTRYPOINT** allows you to configure a container that will run as an executable.

ENTRYPOINT has two forms:

The exec form, which is the preferred form:

```
ENTRYPOINT ["executable", "param1", "param2"]
```

The shell form:

```
ENTRYPOINT command param1 param2
```

Understand how CMD and ENTRYPOINT interact

The table below shows what command is executed for different **ENTRYPOINT / CMD** combinations:

	No ENTRYPOINT	ENTRYPOINT exec_entry p1_entry	ENTRYPOINT ["exec_entry", "p1_entry"]
No CMD	*error, not allowed*	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry
CMD ["exec_cmd", "p1_cmd"]	exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry exec_cmd p1_cmd
CMD ["p1_cmd", "p2_cmd"]	p1_cmd p2_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry p1_cmd p2_cmd
CMD exec_cmd p1_cmd	/bin/sh -c exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry /bin/sh -c exec_cmd p1_cmd

Q&A

Session 3

Docker networking

Agenda

- Networking overview
- Docker network types
- Disable networking for a container
- How to use networks

Docker networking

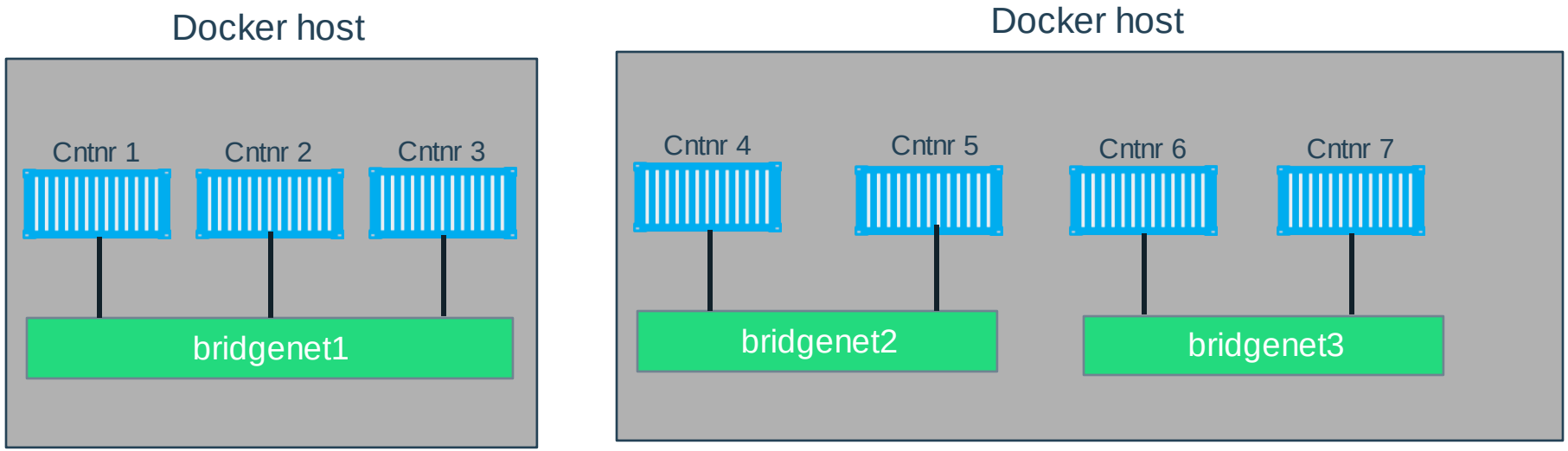
Network driver:

- Bridge
- Host
- None
- Overlay
- Macvlan
- Network plugins: 3rd network plugins

Network driver summary

- **User-defined bridge networks** are best when you need multiple containers to communicate on the same Docker host.
- **Host networks** are best when the network stack should not be isolated from the Docker host, but you want other aspects of the container to be isolated.
- **Overlay networks** are best when you need containers running on different Docker hosts to communicate, or when multiple applications work together using swarm services.
- **Macvlan networks** are best when you are migrating from a VM setup or need your containers to look like physical hosts on your network, each with a unique MAC address.
- **Third-party network plugins** allow you to integrate Docker with specialized network stacks.

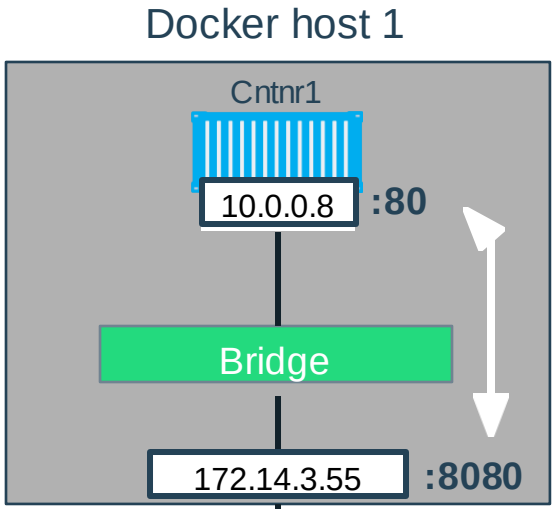
What is Docker Bridge Networking



53

```
docker network create -d bridge --name bridgenet1
```

Docker Bridge Networking and Port Mapping



Host port

Container port

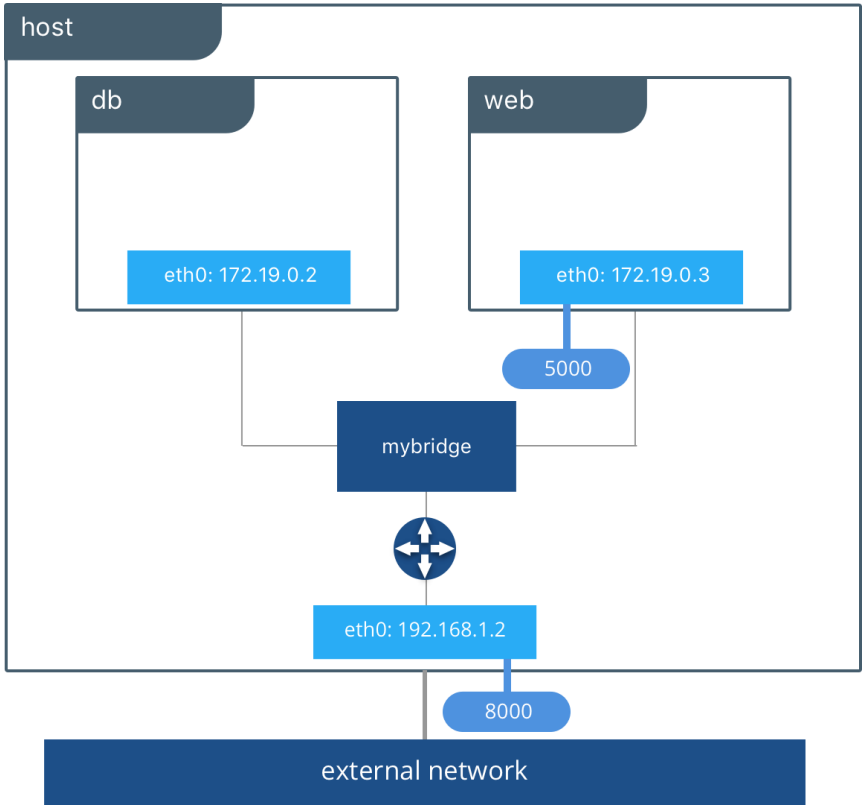
```
$ docker container run -p 8080:80 ...
```

54

Docker Bridge Networking and Port Mapping

With no extra configuration the Docker Engine does the necessary wiring, provides service discovery for the containers, and configures security rules to prevent communication to other networks

55



Differences between user-defined bridges and the default bridge

- User-defined bridges provide automatic DNS resolution between containers
- User-defined bridges provide better isolation
- Containers can be attached and detached from user-defined networks on the fly
- Each user-defined network creates a configurable bridge
- Linked containers on the default bridge network share environment variables.

Manage a user-defined bridge

Use the **docker network create** command to create a user-defined bridge network.

```
$ docker network create my-net
```

You can specify the subnet, the IP address range, the gateway, and other options

Use the **docker network rm** command to remove a user-defined bridge network. If containers are currently connected to the network, disconnect them first.

```
$ docker network rm my-net
```

Connect a container to a user-defined bridge

When you create a new container, you can specify one or more --network flags

```
$ docker create --name my-nginx --network my-net --publish 8080:80 nginx:latest
```

To connect a running container to an existing user-defined bridge, use the **docker network connect** command

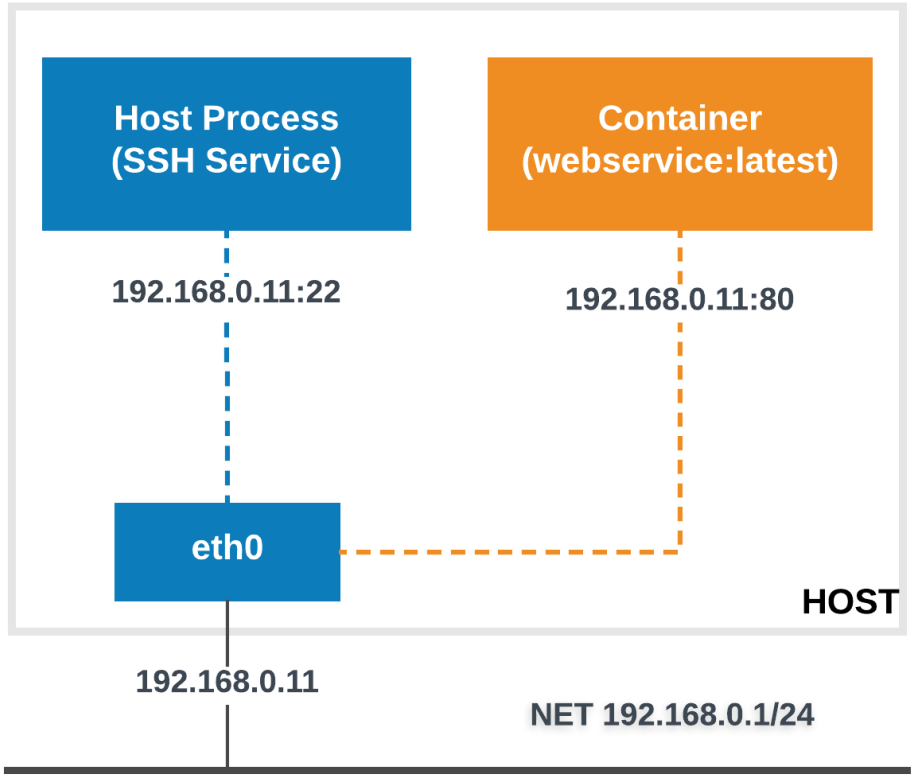
```
$ docker network connect my-net my-nginx
```

Disconnect a container to a user-defined bridge

To disconnect a running container from a user-defined bridge, use the **docker network disconnect** command

```
$ docker network disconnect my-net my-nginx
```

Docker host networking



Use host networking

If you use the host network mode for a container, that container's network stack is not isolated from the Docker host (the container shares the host's networking namespace), and the container does not get its own IP-address allocated.

> Note: Given that the container does not have its own IP-address when using `host` mode networking, port-mapping does not take effect, and the `-p`, `--publish`, `-P`, and `--publish-all` option are ignored, producing a warning instead:

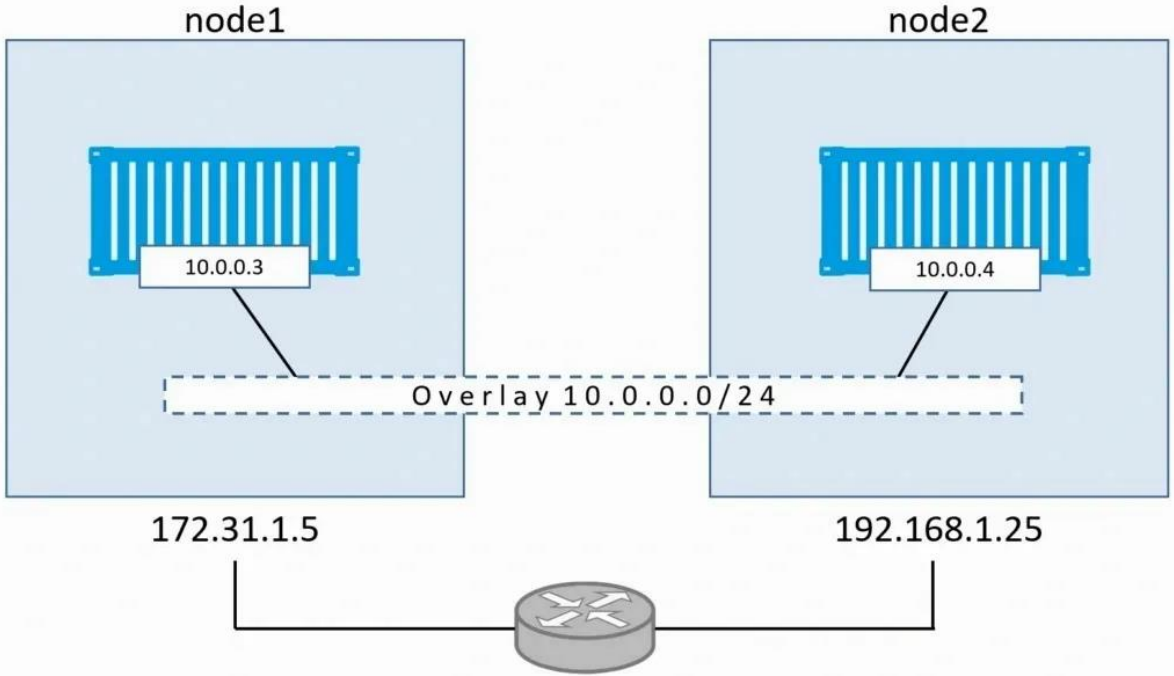
```
WARNING: Published ports are discarded when using host network mode
```

Docker overlay networking

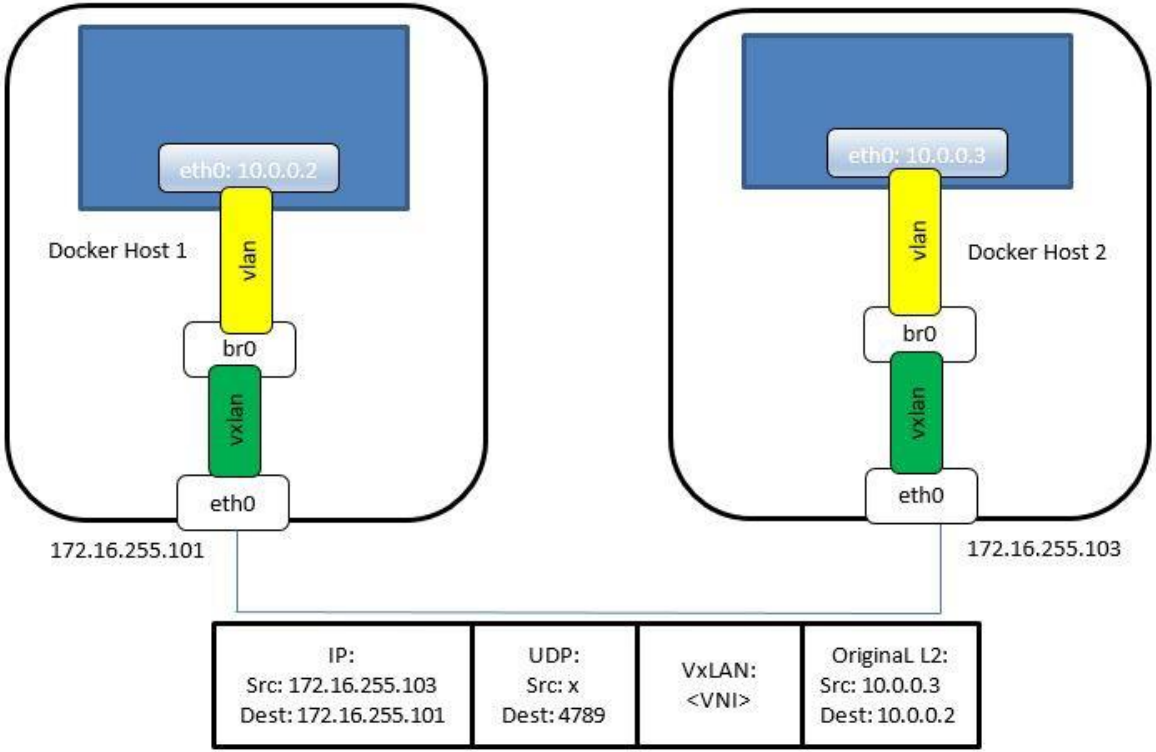
The **overlay network** driver creates a distributed network among multiple Docker daemon hosts.

This network sits on top of (overlays) the host-specific networks, allowing containers connected to it (including swarm service containers) to communicate securely when encryption is enabled.

Docker overlay networking

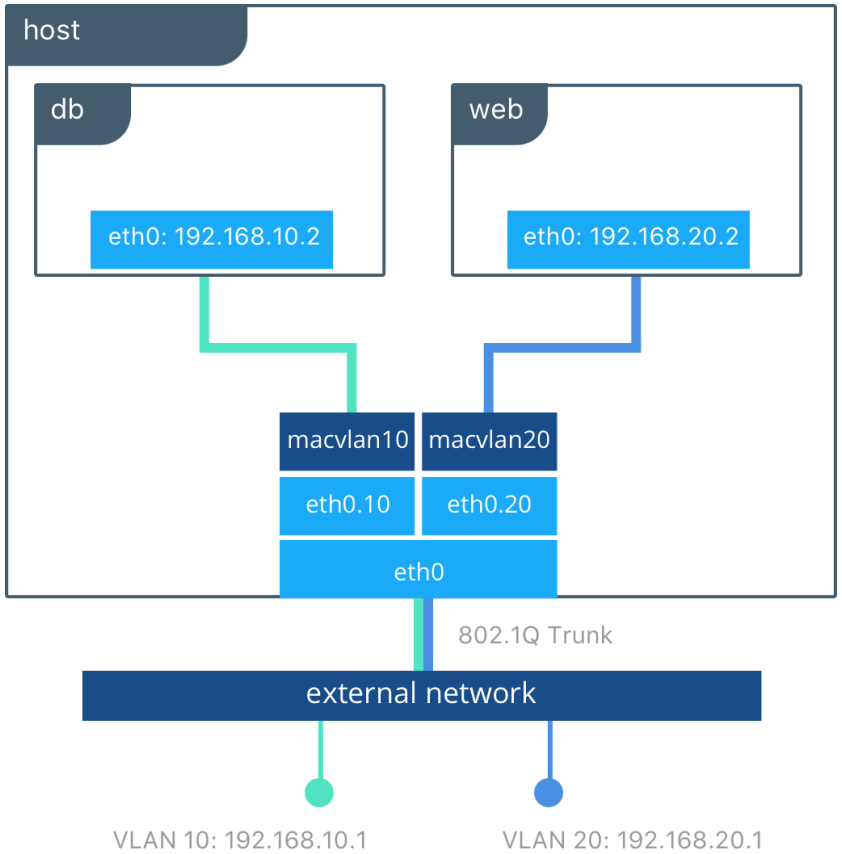


Docker overlay networking

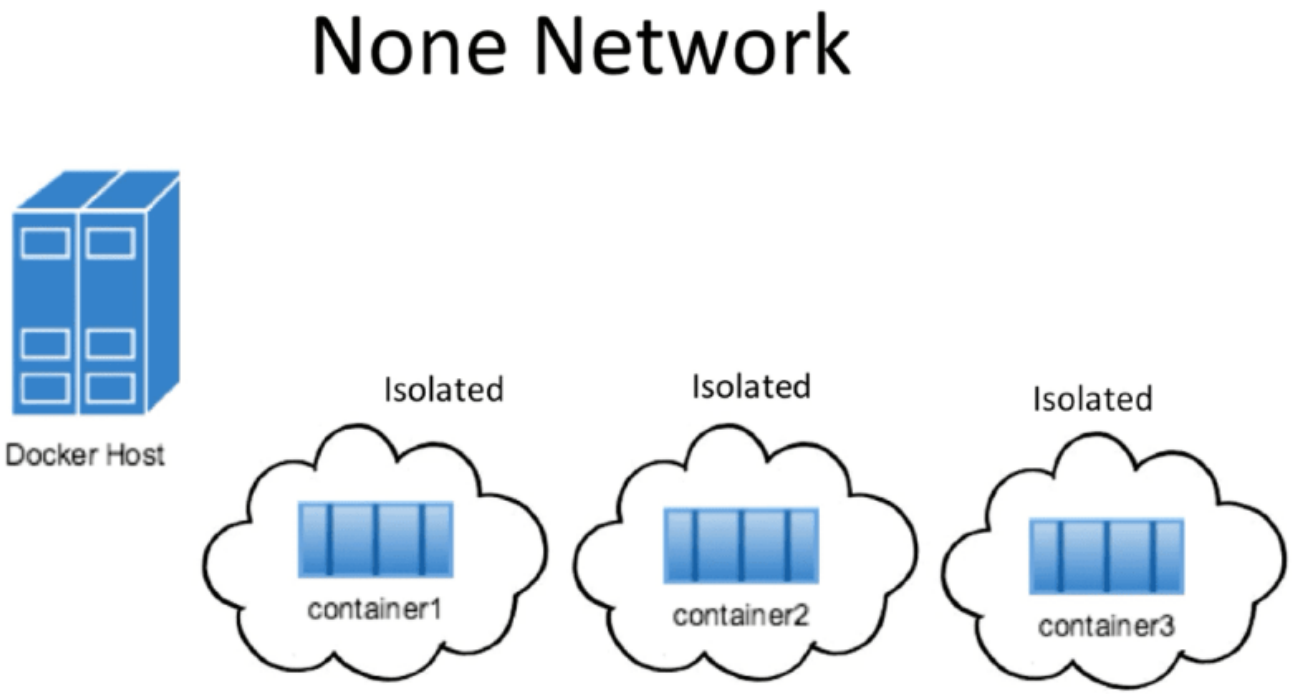


Docker macvlan networking

Some applications, especially legacy applications or applications which monitor network traffic, expect to be directly connected to the physical network.



Docker none network



PRACTICE DOCKER NETWORKING

Q&A

Session 4

Docker storage

Agenda

- Assignment Review & Guides
- Storage overview
- Docker volumes
- Bind mounts
- Store data within containers

Manage data in Docker

By default all files created inside a container are stored on a **writable container layer**.

This means that:

- The data doesn't persist when that container no longer exists
- A container's writable layer is tightly coupled to the host machine where the container is running
- Writing into a container's writable layer requires a storage driver to manage the filesystem. This extra abstraction reduces performance as compared to using ***data volumes***

Choose the right type of mount

- **Volumes** are stored in a part of the host filesystem which is managed by Docker (/var/lib/docker/volumes/ on Linux).
- **Bind mounts** may be stored anywhere on the host system. They may even be important system files or directories.
- **tmpfs** mounts are stored in the host system's memory only, and are never written to the host system's filesystem.

Docker Volumes

- **Volumes** mount a directory on the host into the container at a specific location
- Can be used to share (and persist) data between containers
- Directory persists after the container is deleted (Unless you explicitly delete it)
- Can be created in a Dockerfile or via CLI

Use volumes

Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts

Create and manage volumes

Create a volume:

```
$ docker volume create my-vol
```

List volumes:

```
$ docker volume

Local               my-vol
```

Inspect a volume:

```
$ docker volume inspect my-vol
```

Remove a volume:

```
$ docker volume rm my-vol
```

Start a container with a volume

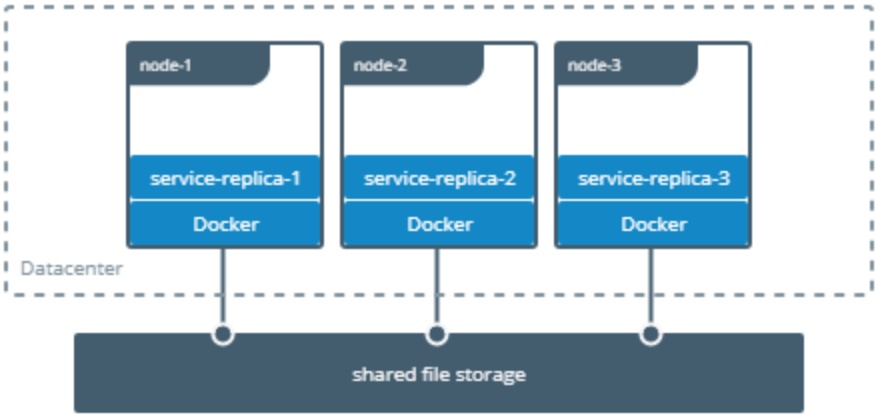
If you start a container with a volume that does not yet exist, Docker creates the volume for you

```
$ docker run -d \
  --name devtest \
  -v myvol2:/app \
  nginx:latest
```

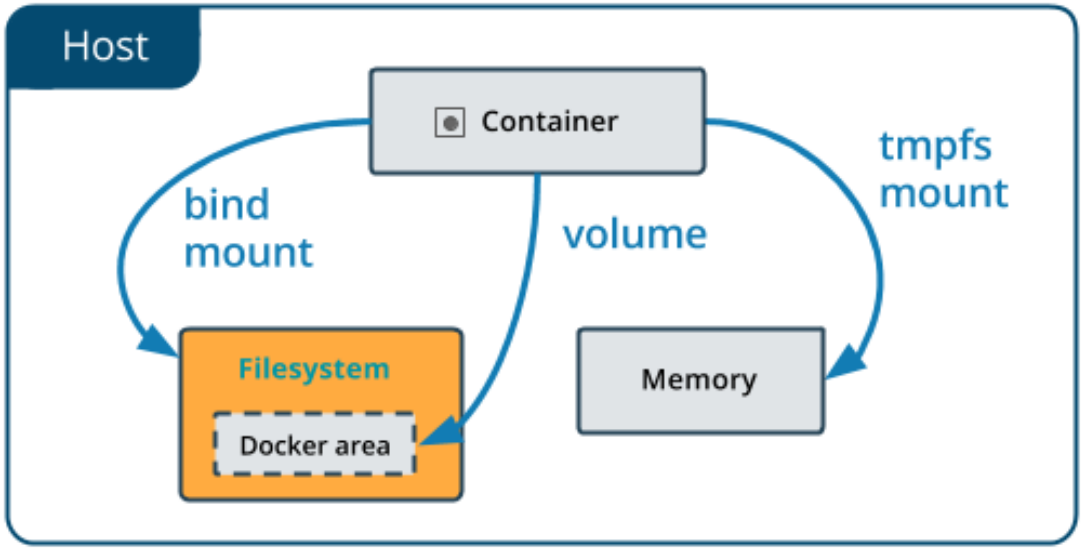
Use docker inspect devtest to verify that the volume was created and mounted correctly

Share data among machines

When building fault-tolerant applications, you might need to configure multiple replicas of the same service to have access to the same files.



Docker bind mount



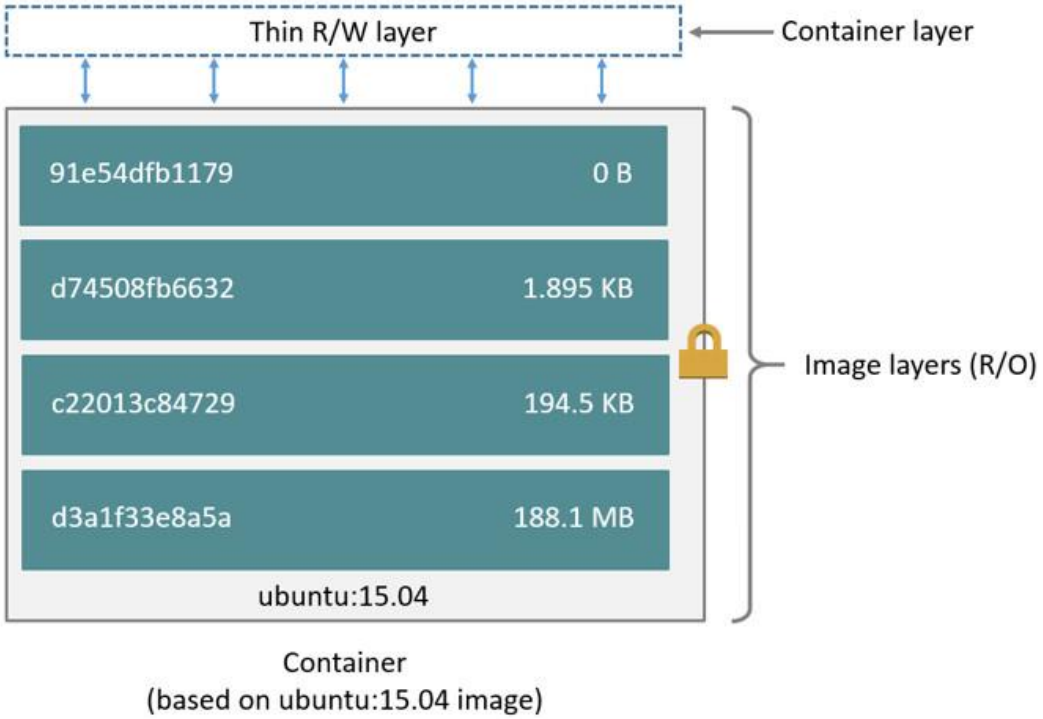
Use tmpfs mounts

Volumes and **bind mounts** let you share files between the host machine and container so that you can persist data even after the container is stopped.

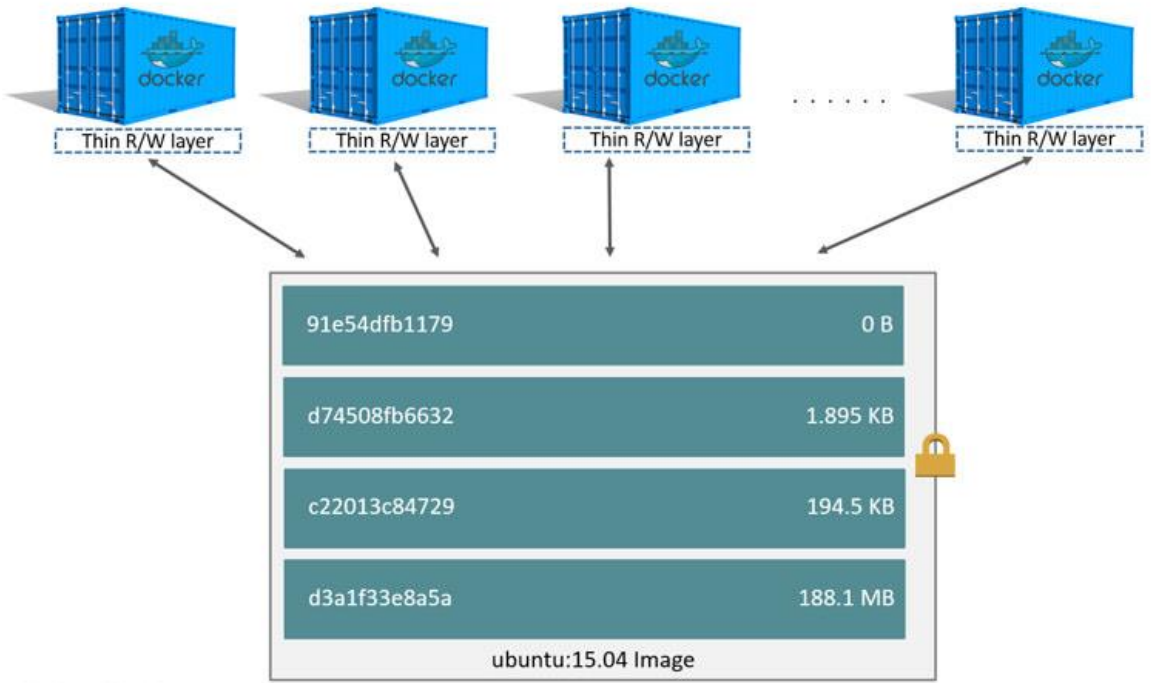
If you're running Docker on Linux, you have a third option: tmpfs mounts. When you create a container with a tmpfs mount, the container can create files outside the container's writable layer.

Storage data within container

A ***storage driver*** handles the details about the way these layers interact with each other. Different storage drivers are available, which have advantages and disadvantages in different situations.



Container and layers



Docker storage drivers

Docker supports several **storage drivers**, using a pluggable architecture. The **storage driver** controls how images and containers are stored and managed on your Docker host.

After you have read the storage driver overview, the next step is to choose the best storage driver for your workloads. Use the storage driver with the best overall performance and stability in the most usual scenarios.

Docker storage drivers

Driver	Description
`overlay2`	`overlay2` is the preferred storage driver for all currently supported Linux distributions, and requires no extra configuration.
`fuse-overlayfs`	`fuse-overlayfs` is preferred only for running Rootless Docker on a host that does not provide support for rootless `overlay2`. On Ubuntu and Debian 10, the `fuse-overlayfs` driver does not need to be used, and `overlay2` works even in rootless mode. Refer to the rootless mode documentation for details.
`btrfs` and `zfs`	The `btrfs` and `zfs` storage drivers allow for advanced options, such as creating "snapshots", but require more maintenance and setup. Each of these relies on the backing filesystem being configured correctly.
`vfs`	The `vfs` storage driver is intended for testing purposes, and for situations where no copy-on-write filesystem can be used. Performance of this storage driver is poor, and is not generally recommended for production use.

Docker storage drivers

`aufs`	The `aufs` storage driver Was the preferred storage driver for Docker 18.06 and older, when running on Ubuntu 14.04 on kernel 3.13 which had no support for `overlay2`. However, current versions of Ubuntu and Debian now have support for `overlay2`, which is now the recommended driver.
`devicemapper`	The `devicemapper` storage driver requires `direct-lvm` for production environments, because `loopback-lvm`, while zero-configuration, has very poor performance. `devicemapper` was the recommended storage driver for CentOS and RHEL, as their kernel version did not support `overlay2`. However, current versions of CentOS and RHEL now have support for `overlay2`, which is now the recommended driver.
`overlay`	The legacy `overlay` driver was used for kernels that did not support the "multiple-lowerdir" feature required for `overlay2` All currently supported Linux distributions now provide support for this, and it is therefore deprecated.

Docker storage drivers

Docker supports several **storage drivers**, using a pluggable architecture. The **storage driver** controls how images and containers are stored and managed on your Docker host.

After you have read the storage driver overview, the next step is to choose the best storage driver for your workloads. Use the storage driver with the best overall performance and stability in the most usual scenarios.

PRACTICE DOCKER STORAGE

Q&A

Session 5

Making a real project with Docker and NodeJs

Agenda

- Assignment Review & Guides
- Keep container alive during daemon downtime
- Run multiple service in a container
- Container logging
- Docker compose introduction
- Project outline

Keep containers alive during daemon downtime

By default, when the Docker daemon terminates, it **shuts down** running containers. You can configure the daemon so that containers remain running if the daemon becomes unavailable.

This functionality is called **live restore**. The live restore option helps reduce container downtime due to daemon crashes, planned outages, or upgrades

Enable live restore

There are two ways to enable the **live restore** setting to keep containers alive when the daemon becomes unavailable

- Add the configuration to the daemon configuration file. On Linux, this defaults to **/etc/docker/daemon.json**

```
{
    "live-restore":true
}
```

- Start the dockerd process manually with the **--live-restore** flag

Run multiple service in a container

A container's main running process is the **ENTRYPOINT** and/or **CMD** at the end of the Dockerfile. It is generally recommended that you separate areas of concern by using one service per container

If you need to run more than one service within a container, you can accomplish this in a few different ways.

- Put all of your commands in a wrapper script, then run the wrapper script as your CMD
- Use bash's job control
- Use a process manager like supervisord

Using wrapper script

The Dockerfile and my_wrapper_script:

```
FROM ubuntu:latest
COPY my_first_process my_first_process
COPY my_second_process my_second_process
COPY my_wrapper_script.sh my_wrapper_script.sh
CMD ./my_wrapper_script.sh
```

```
#!/bin/bash
# Start the first process
./my_first_process -D
status=$?
if [ $status -ne 0 ]; then
    echo "Failed to start my_first_process: $status"
    exit $status
fi
# Start the second process
./my_second_process -D
status=$?
if [ $status -ne 0 ]; then
     echo "Failed to start my_second_process: $status"
     exit $status
fi

while sleep 60; do
   ps aux |grep my_first_process |grep -q -v grep

   PROCESS_1_STATUS=$?
   ps aux |grep my_second_process |grep -q -v grep

   PROCESS_2_STATUS=$?
   if [ $PROCESS_1_STATUS  -ne 0 -o $PROCESS_2_STATUS  -ne 0 ]; then
       echo "One of the processes has already exited."
       exit 1
   fi
done
```

Use bash's job control

The Dockerfile and my_wrapper_script:

```
FROM ubuntu:latest
COPY my_main_process my_main_process
COPY my_helper_process my_helper_process
COPY my_wrapper_script.sh my_wrapper_script.sh
CMD ./my_wrapper_script.sh
```

```
#!/bin/bash

# turn on bash's job control
 set -m

# Start the primary process and put it in the background
./my_main_process &

# Start the helper process
./my_helper_process

# the my_helper_process might need to know how to wait on the
# primary process to start before it does its work and returns
# now we bring the primary process back into the foreground
# and leave it there
 fg %1
```

Use a process manager

This is a moderately heavy-weight approach that requires you to package supervisord and its configuration in your image (or base your image on one that includes supervisord), along with the different applications it manages

```
FROM ubuntu:latest

RUN apt-get update && apt-get install -y supervisor

RUN mkdir -p /var/log/supervisor

COPY supervisord.conf /etc/supervisor/conf.d/supervisord.conf

COPY my_first_process my_first_process

COPY my_second_process my_second_process

CMD ["/usr/bin/supervisord"]
```

Runtime metrics

You can use the **docker stats** command to live stream a container's runtime metrics. The command supports CPU, memory usage, memory limit, and network IO metrics.

The following is a sample output from the **docker stats** command

```
$docker stats redis1 redis2
```

```
CONTAINER           CPU %               MEM USAGE / LIMIT     MEM %               NET I/O             BLOCK I/O
redis1              0.07%               796 KB / 64 MB        1.21%               788 B / 648 B       3.568 MB / 512 KB
redis2              0.07%               2.746 MB / 64 MB      4.29%               1.266 KB / 648 B    12.4 MB / 0 B
```

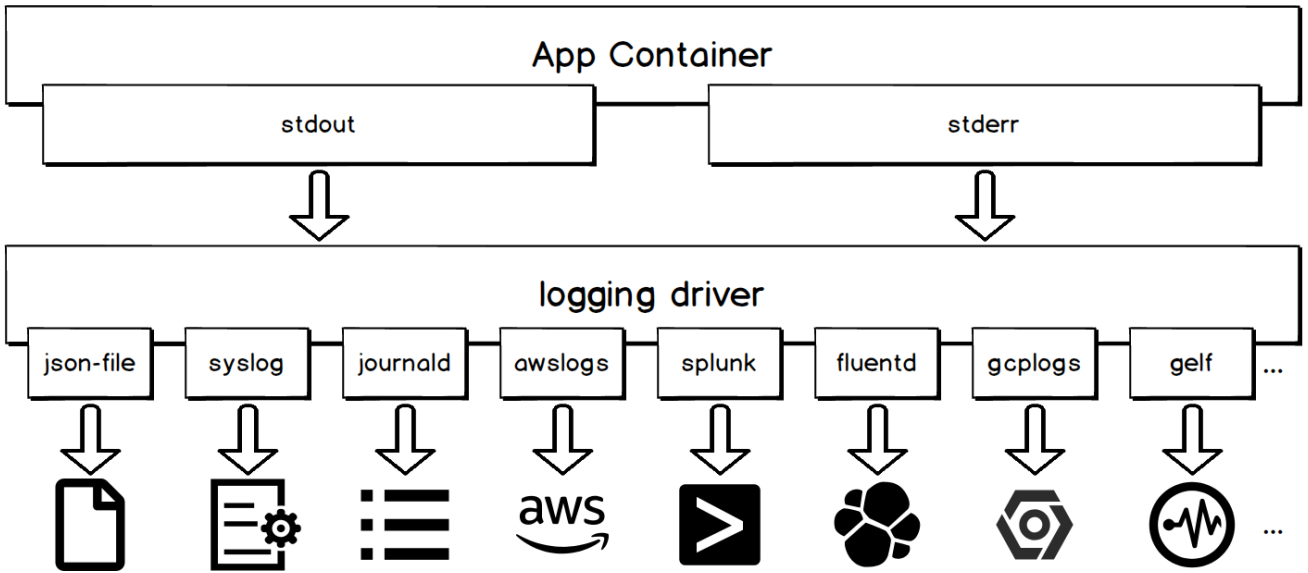
Control groups

Linux Containers rely on **control groups** which not only track groups of processes, but also expose metrics about CPU, memory, and block I/O usage. You can access those metrics and obtain network usage metrics as wellControl groups are exposed through a pseudo-filesystem.

The metrics can be exposed from cgroups are memory, CPU, block I/O

View logs for a container

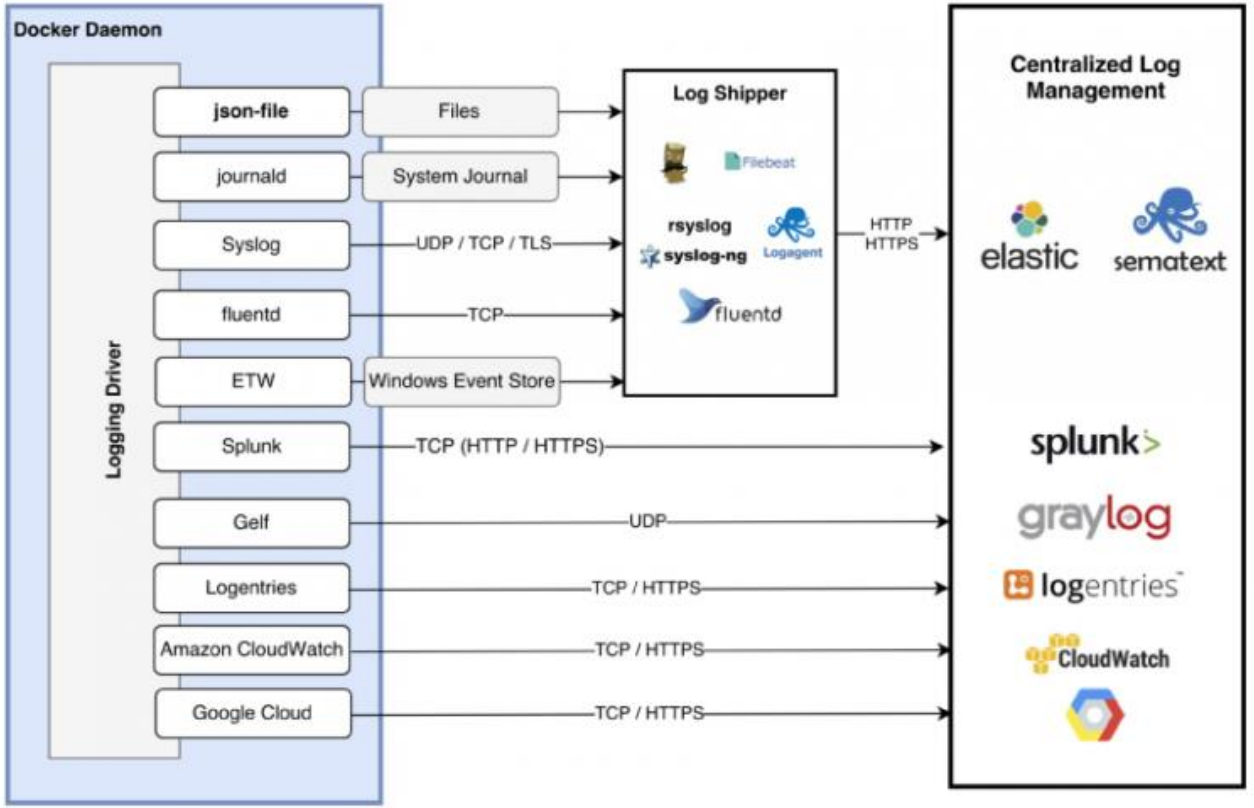
The **docker logs** command shows information logged by a running container



Docker logging driver overview

Docker includes multiple logging mechanisms to help you get information from running containers and services. These mechanisms are called logging drivers

As a default, Docker uses the json-file logging driver, which caches container logs as JSON internally



Configure the default logging drivers

To configure the Docker daemon to default to a specific logging driver, set the value of log-driver to the name of the logging driver in the daemon.json configuration file

The default logging driver is json-file. The following example sets the default logging driver to the local log driver:

```
{
    "log-driver": "local"
}
```

We can config logging option with the key log-opts

```
{
    "log-driver": "json-file",
    "log-opts": {
        "max-size": "10m",
        "max-file": "3",
        "labels": "production_status",
        "env": "os,customer"
    }
}
```

Configure the logging driver for a container

When you start a container, you can configure it to use a different logging driver than the Docker daemon's default, using the **--log-driver** flag. If the logging driver has configurable options, you can set them using one or more instances of the **--log-opt <NAME>=<VALUE>** flag

The following example starts an Alpine container with the none logging driver.

```
$ docker run -it --log-driver none alpine ash
```

To find the current logging driver for a running container, use docker inspect command

```
$ docker inspect -f '{{.HostConfig.LogConfig.Type}}' <CONTAINER>

  json-file
```

Install the logging driver plugin

To install a logging driver plugin, use **docker plugin install <org/image>**, using the information provided by the plugin developer.

You can list all installed plugins using docker plugin ls, and you can inspect a specific plugin using **docker inspect**.

Customize log driver output

The tag log option specifies how to format a tag that identifies the container's log messages. By default, the system uses the first 12 characters of the container ID. To override this behavior, specify a tag option:

```
$ docker run --log-driver=fluentd --log-opt fluentd-address=myhost.local:24224 --log-opt tag="mailer"
```

For example, specifying a --log-opt tag="{{.ImageName}}/{{.Name}}/{{.ID}}" value yields syslog log lines like:

```
Aug  7 18:33:19 HOSTNAME hello-world/foobar/5790672ab6a0[9103]: Hello from Docker.
```

Overview of Docker Compose

Compose is a tool for defining and running multi-container Docker applications.

It has commands for managing the whole lifecycle of your application:

- Start, stop, and rebuild services
- View the status of running services
- Stream the log output of running services
- Run a one-off command on a service

Using Docker Compose

Using Compose is basically a three-step process:

- Define your app's environment with a Dockerfile
- Define the services that make up your app in docker-compose.yml
- Run **docker compose up** and the Docker compose command starts and runs your entire app.

A docker-compose.yml looks like this:

```yaml
version: "3.9"  # optional since v1.27.0
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
      - logvolume01:/var/log
    links:
      - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```

Features of Docker Compose

The features of **Compose** that make it effective are:

- Multiple isolated environments on a single host
- Preserve volume data when containers are created
- Only recreate containers that have changed
- Variables and moving a composition between environments

Install Docker Compose

You can run **Compose** on macOS, Windows, and 64-bit Linux.

Docker Compose relies on Docker Engine for any meaningful work, so make sure you have **Docker Engine** installed either locally or remote, depending on your setup.

- On desktop systems like Docker Desktop for Mac and Windows, Docker Compose is included as part of those desktop installs.
- On Linux systems, first install the Docker Engine for your OS as described on the Get Docker page, then installing Compose on Linux systems.
- You can run Compose as a non-root user

Environment variables in Compose

It's possible to use environment variables in your shell to populate values inside a Compose file:

```
web:
  image: "webapp:${TAG}"
```

You can set default values for any environment variables referenced in the Compose file, or used to configure Compose, in an environment file named .env

```
$ cat .env
TAG=v1.5

$ cat docker-compose.yml
version: '3'
services:
  web:
    image: "webapp:${TAG}"
```

Using the “--env-file” option

By passing the file as an argument, you can store it anywhere and name it appropriately, for example, .env.ci, .env.dev, .env.prod. Passing the file path is done using the --env-file option:

```
$ docker-compose --env-file ./config/.env.dev up
```

The .env file is loaded by default. Passing the --env-file argument overrides the default file path

Set environment variables in containers

You can set environment variables in a service's containers with the 'environment' key, just like with docker run -e VARIABLE=VALUE ...:

```
web:
  environment:
    - DEBUG=1
```

Pass environment variables to containers

You can pass environment variables from your shell straight through to a service's containers with the 'environment' key by not giving them a value, just like with docker run -e VARIABLE ...

```
web:
  environment:
    - DEBUG
```

The “env_file” configuration option

You can pass multiple environment variables from an external file through to a service’s containers with the ‘env_file’ option, just like with docker run --env-file=FILE …:

```
web:
  env_file:
    - web-variables.env
```

Set environment variables with 'docker-compose run'

Similar to docker run -e, you can set environment variables on a one-off container with docker-compose run -e:

```
$ docker-compose run -e DEBUG=1 web python console.py
```

You can also pass a variable from the shell by not giving it a value:

```
$ docker-compose run -e DEBUG web python console.py
```

Networking in Compose

By default **Compose** sets up a single network for your app. Each container for a service joins the default network and is both reachable by other containers on that network, and discoverable by them at a hostname identical to the container name.

Your app's network is given a name based on the "project name", which is based on the name of the directory it lives in. You can override the project name with either the --project-name flag or the COMPOSE_PROJECT_NAME environment variable.

Networking in Compose

When you run docker-compose up, the following happens:

- A network called myapp_default is created.
- A container is created using web's configuration. It joins the network myapp_default under the name web.
- A container is created using db's configuration. It joins the network myapp_default under the name db.

```
version: "3.9"
services:
  web:
    build: .
    ports:
      - "8000:8000"
  db:
    image: postgres
    ports:
      - "8001:5432"
```

Links

Links allow you to define extra aliases by which a service is reachable from another service. They are not required to enable services to communicate - by default, any service can reach any other service at that service's name

```
version: "3.9"
services:

  web:
    build: .
    links:
      - "db:database"
  db:
    image: postgres
```

docker-compose CLI

You can also see this information by running **docker-compose --help** from the command line.

Use the **-f** flag to specify the location of a Compose configuration file.

You can supply multiple **-f** configuration files. When you supply multiple files, Compose combines them into a single configuration. Compose builds the configuration in the order you supply the files. Subsequent files override and add to their predecessors.

DO A LAB: COMPOSE AND WORDPRESS

DO A PROJECT: COMPOSE WITH NODEJS AND MYSQL

Q&A

Session 6

Docker Swarm

Agenda

- Introduction to Docker Swarm
- Set up docker swarm
- Deploy a stack to a Swarm

Swarm mode overview

Feature highlights

- Cluster management integrated with Docker Engine
- Decentralized design
- Declarative service model
- Scaling
- Desired state reconciliation
- Multi-host networking
- Service discovery
- Load balancing

What is Swarm

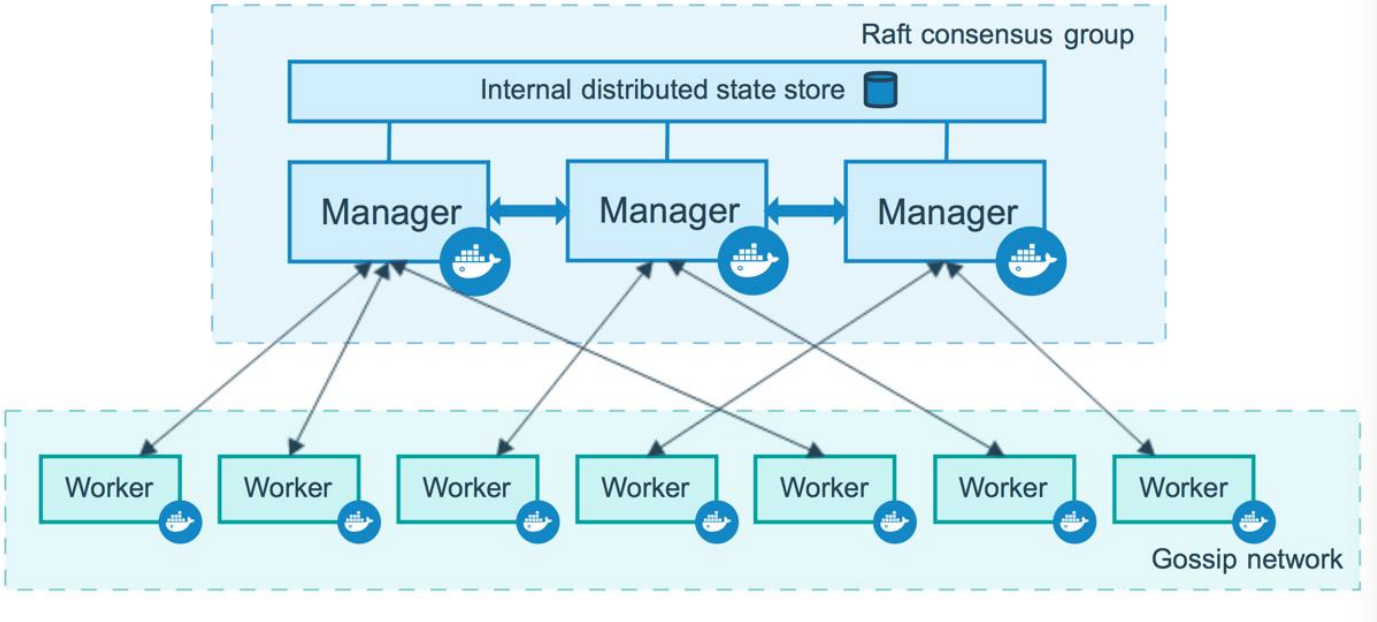
A **swarm** consists of multiple Docker hosts which run in swarm mode and act as managers (to manage membership and delegation) and workers (which run swarm services). A given Docker host can be a manager, a worker, or perform both roles

One of the key advantages of **swarm** services over standalone containers is that you can modify a service's configuration, including the networks and volumes it is connected to, without the need to manually restart the service. Docker will update the configuration, stop the service tasks with the out of date configuration, and create new ones matching the desired configuration.

Nodes

A **node** is an instance of the Docker engine participating in the swarm. You can also think of this as a Docker node. You can run one or more nodes on a single physical computer or cloud server, but production swarm deployments typically include Docker nodes distributed across multiple physical and cloud machines.

There are two types of nodes: **managers** and **workers**



Manager nodes

Manager nodes handle cluster management tasks:

- maintaining cluster state
- scheduling services
- serving swarm mode HTTP API endpoints

To take advantage of swarm mode's fault-tolerance features, Docker recommends you implement an **odd number** of nodes according to your organization's high-availability requirements. When you have multiple managers you can recover from the failure of a manager node without downtime.

Worker nodes

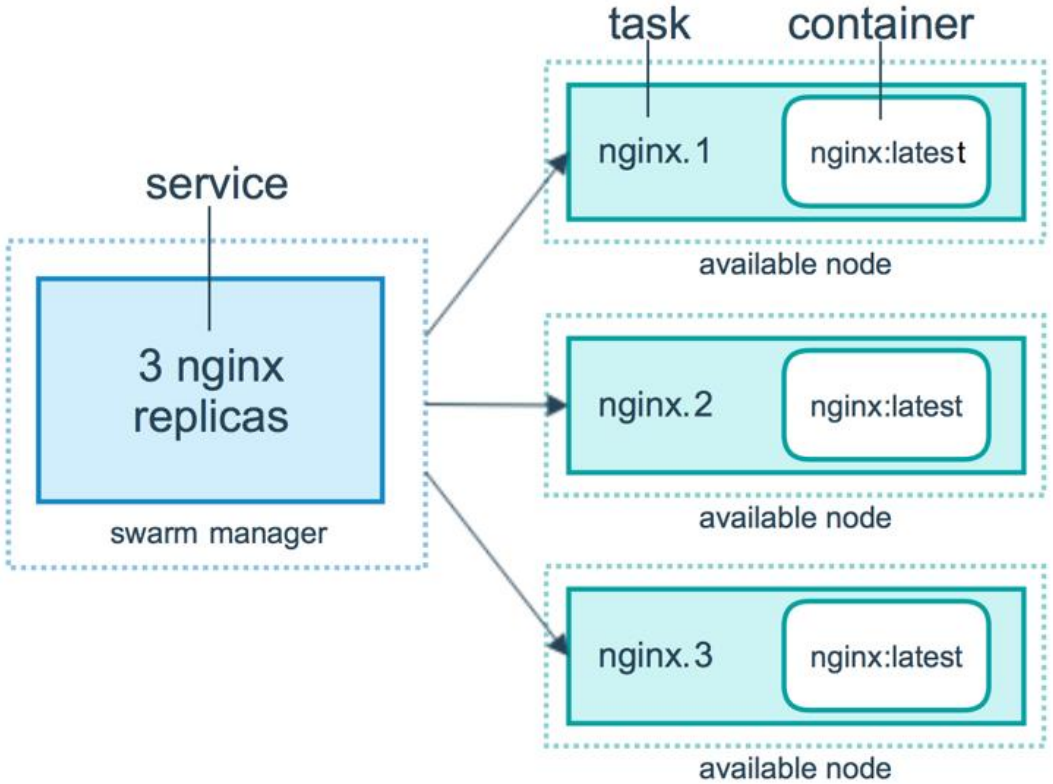
Worker nodes are also instances of Docker Engine whose sole purpose is to execute containers. Worker nodes don't participate in the Raft distributed state, make scheduling decisions, or serve the swarm mode HTTP API.

You can promote a worker node to be a manager by running **docker node promote**. For example, you may want to promote a worker node when you take a manager node offline for maintenance

Services, tasks, and containers

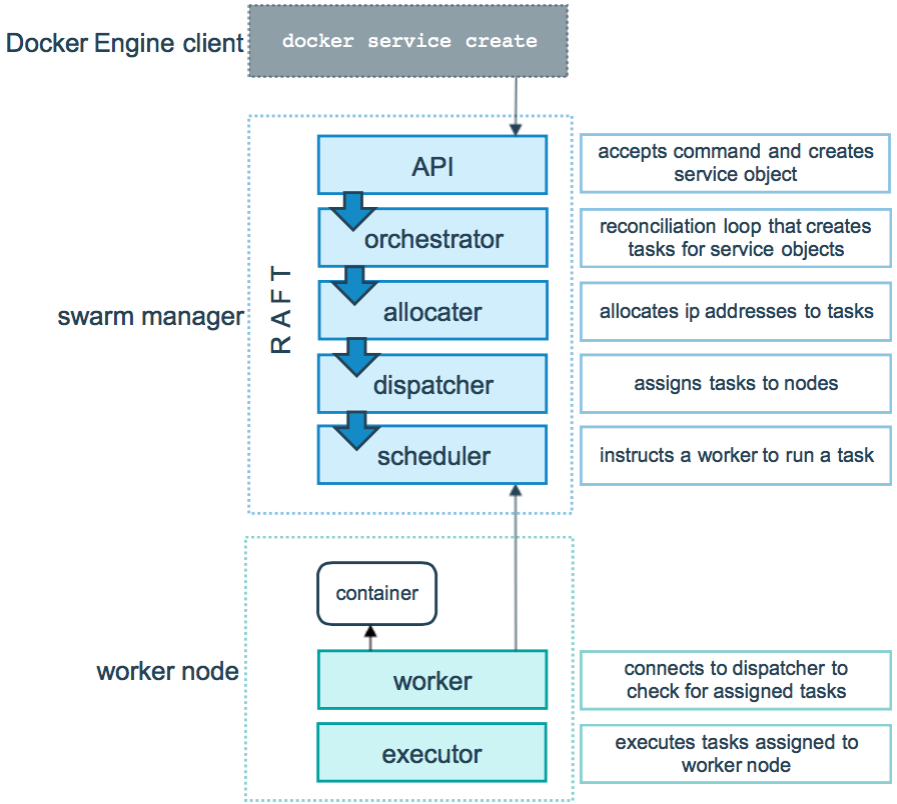
When you deploy the **service** to the swarm, the swarm manager accepts your service definition as the desired state for the service.

Then it schedules the service on nodes in the swarm as one or more replica tasks. The tasks run independently of each other on nodes in the swarm.



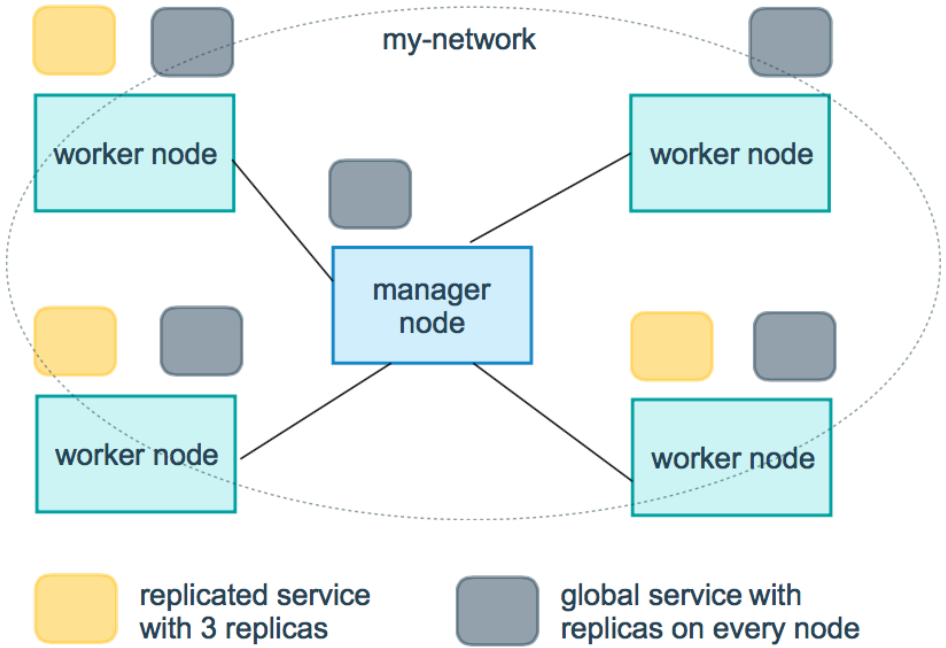
Tasks and scheduling

A **task** is the atomic unit of scheduling within a swarm. When you declare a desired service state by creating or updating a service, the orchestrator realizes the desired state by scheduling tasks



Replicated and global services

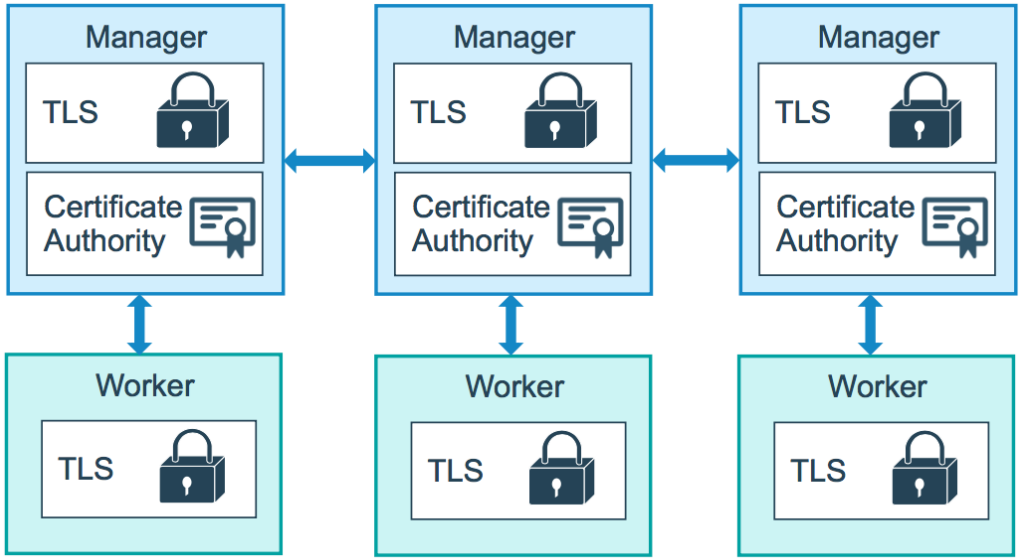
There are two types of service deployments, **replicated** and **global**.



Manage swarm security with public key infrastructure (PKI)

The swarm mode **public key infrastructure (PKI)** system built into Docker makes it simple to securely deploy a container orchestration system.

The nodes in a swarm use **mutual Transport Layer Security (TLS)** to authenticate, authorize, and encrypt the communications with other nodes in the swarm.



Run Docker Engine in swarm mode

When you first install and start working with Docker Engine, swarm mode is disabled by default. When you enable swarm mode, you work with the concept of services managed through the docker service command.

There are two ways to run the Engine in swarm mode:

- Create a new swarm, covered in this article.
- Join an existing swarm.

Create a new swarm

When you run the command to create a swarm, the Docker Engine starts running in swarm mode.

Run docker **swarm init** to create a single-node swarm on the current node

The output for **docker swarm init** provides the connection command to use when you join new worker nodes to the swarm:

```
$ docker swarm init
Swarm initialized: current node (dxn1zf6l61qsb1josjja83ngz) is now a manager.

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

    docker swarm join \
    --token SWMTKN-1-49nj1cmql0jkz5s954yi3oex3nedyz0fb0xx14ie39trti4wxv-8vxv8rssmk743ojnwacrr2e7c \
    192.168.99.100:2377

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

Join node to swarm as worker node

To retrieve the join command including the join token for worker nodes, run the following command on a manager node:

```
$ docker swarm join-token worker

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

    docker swarm join \
    --token SWMTKN-1-49nj1cmql0jkz5s954yi3oex3nedyz0fb0xx14ie39trti4wxv-8vxv8rssmk743ojnwacrr2e7c \
    192.168.99.100:2377
```

Run the command from the output on the worker to join the swarm:

```
$ docker swarm join \
  --token SWMTKN-1-49nj1cmql0jkz5s954yi3oex3nedyz0fb0xx14ie39trti4wxv-8vxv8rssmk743ojnwacrr2e7c \
  192.168.99.100:2377

This node joined a swarm as a worker.
```

Join node to swarm as master node

To retrieve the join command including the join token for manager nodes, run the following command on a manager node:

```
$ docker swarm join-token manager

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

    docker swarm join \
    --token SWMTKN-1-61ztec5kyafptydic6jfc1i33t37flcl4nuipzcusor96k7kby-5vy9t8u35tuqm7vh67lrz9xp6 \
    192.168.99.100:2377
```

Run the command from the output on the new manager node to join it to the swarm:

```
$ docker swarm join \
  --token SWMTKN-1-61ztec5kyafptydic6jfc1i33t37flcl4nuipzcusor96k7kby-5vy9t8u35tuqm7vh67lrz9xp6 \
  192.168.99.100:2377

This node joined a swarm as a manager.
```

Manage nodes in a swarm

As part of the swarm management lifecycle, you may need to view or update a node as follows:

- list nodes in the swarm
- inspect an individual node
- update a node
- leave the swarm

List nodes in the swarm

To view a list of nodes in the swarm run **docker node ls** from a manager node:

```
$ docker node ls

ID                            HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
46aqrk4e473hjbt745z53cr3t     node-5    Ready   Active        Reachable
61pi3d91s0w3b90ijw3deeb2q     node-4    Ready   Active        Reachable
a5b2m3oghd48m8eu391pefq5u     node-3    Ready   Active
e7p8btxeu3ioshyuj6lxiv6g0     node-2    Ready   Active
ehkv3bcimagdese79dn78otj5 *   node-1    Ready   Active        Leader
```

Inspect an individual node

You can run **docker node inspect <NODE-ID>** on a manager node to view the details for an individual node. The output defaults to JSON format, but you can pass the --pretty flag to print the results in human-readable format

```
$ docker node inspect self --pretty

ID:                     ehkv3bcimagdese79dn78otj5
Hostname:               node-1
Joined at:              2016-06-16 22:52:44.9910662 +0000 utc
Status:
 State:                 Ready
 Availability:          Active
Manager Status:
 Address:               172.17.0.2:2377
 Raft Status:           Reachable
 Leader:                Yes
Platform:
 Operating System:      linux
 Architecture:          x86_64
Resources:
 CPUs:                  2
 Memory:                1.954 GiB
Plugins:
  Network:              overlay, host, bridge, overlay, null
  Volume:               local
Engine Version:         1.12.0-dev
```

Update a node

You can modify node attributes as follows:

- change node availability
- add or remove label metadata
- change a node role

Leave the swarm

Run the **docker swarm leave** command on a node to remove it from the swarm.

When a node leaves the swarm, the Docker Engine stops running in swarm mode. The orchestrator no longer schedules tasks to the node.

If the node is a manager node, you receive a warning about maintaining the quorum. To override the warning, pass the --force flag. If the last manager node leaves the swarm, the swarm becomes unavailable requiring you to take disaster recovery measures.

Deploy services to a swarm

To create a single-replica service with no extra configuration, you only need to supply the image name:

```
$ docker service create nginx
```

The service is scheduled on an available node. To confirm that the service was created and started successfully, use the docker service ls command:

```
$ docker service ls
```

To provide a name for your service, use the --name flag

```
$ docker service create --name my_web nginx
```

Update a service

You can change almost everything about an existing service using the docker service update command. When you update a service, Docker stops its containers and restarts them with the new configuration.

```
$ docker service update --publish-add 80 my_web
```

Remove a service

To remove a service, use the **docker service remove** command. You can remove a service by its ID or name, as shown in the output of the **docker service ls** command

```
$ docker service remove my_web
```

PRACTICE DOCKER NETWORKING

Q&A

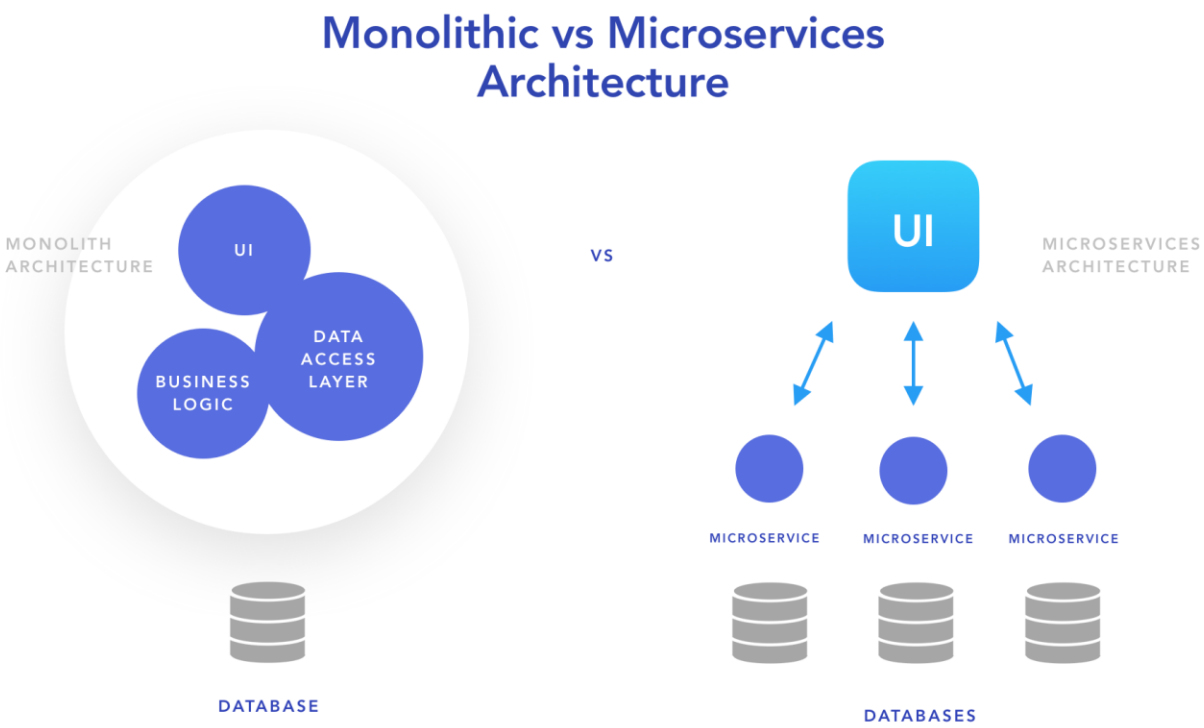
Session 7

Kubernetes

Agenda

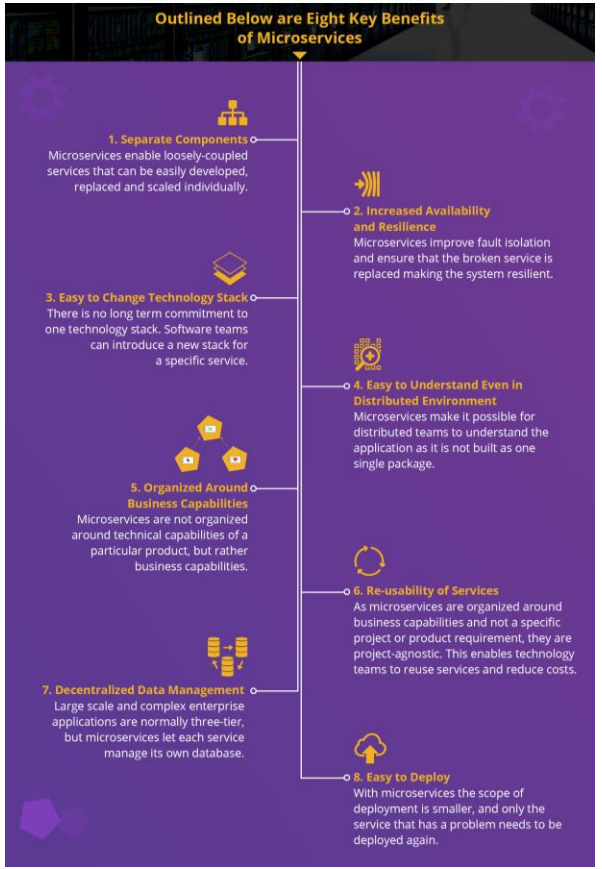
- Assignment Review & Guides
- From Monolithic to Microservices
- What is Kubernetes?
- Kubernetes architecture

From Monolithic to Microservices



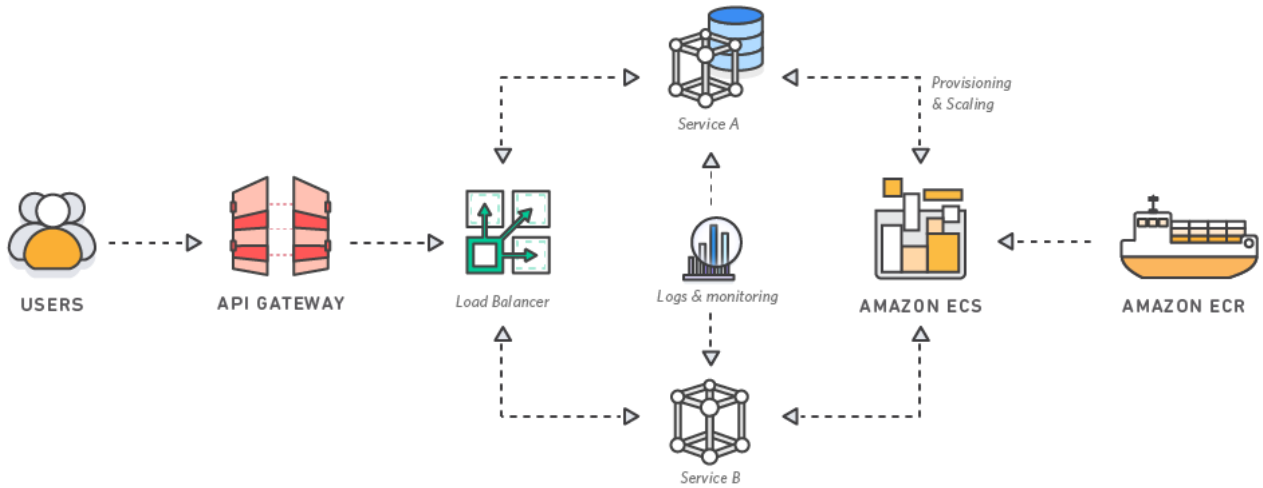
Benefits of microservices

Microservice architecture divides a single application into smaller sets of services, each running on its own but communicating with each other through APIs



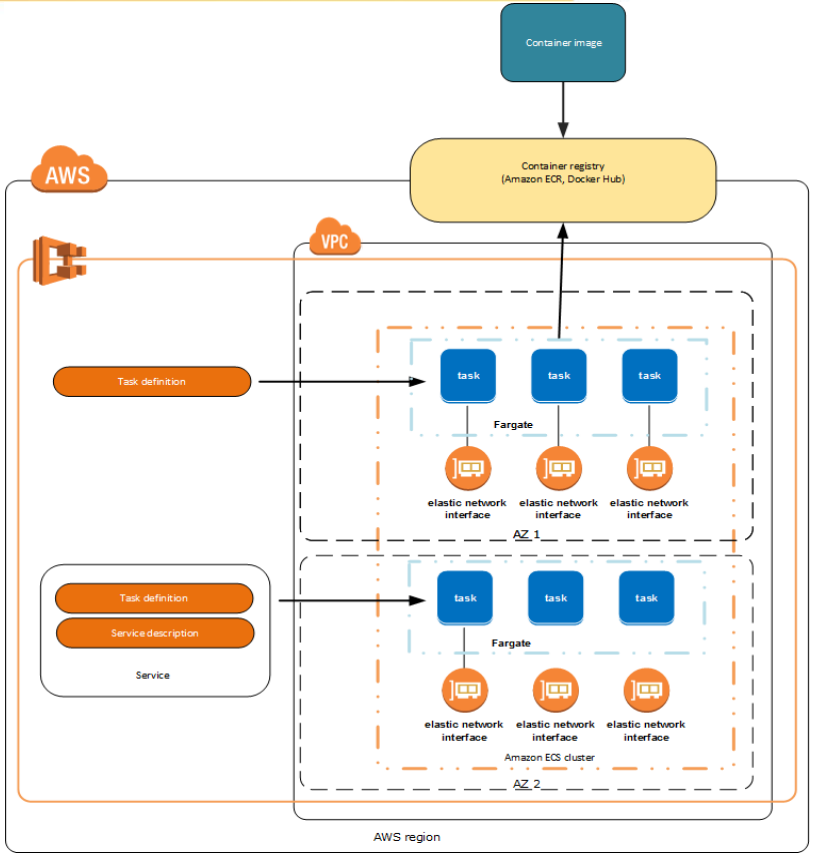
AWS ECS

Amazon Elastic Container Service (Amazon ECS) is a highly scalable, fast, container management service that makes it easy to run, stop, and manage Docker containers on a cluster of Amazon EC2 instances.



ECS Architecture

- Elements of the Amazon ECS:
- Cluster
- Container agent
- Containers and images
- Task definitions
- Task and scheduling

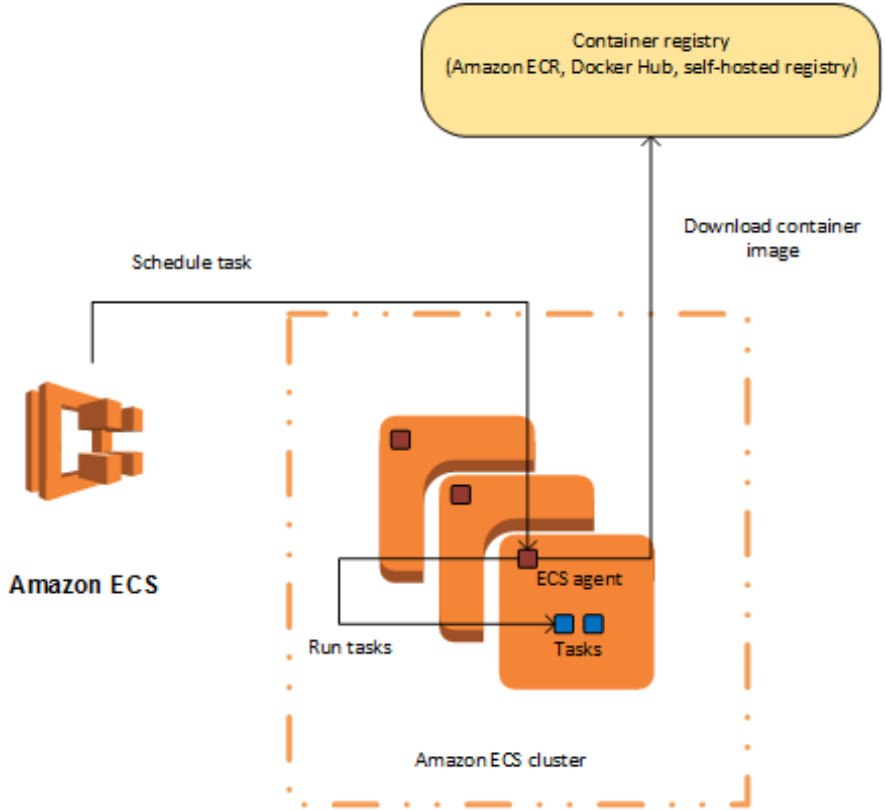


ECS - Cluster

- An Amazon ECS cluster is a logical grouping of tasks or services.
- You can register one or more Amazon EC2 instances, also referred to as container instances with your cluster to run tasks on, or you can use the serverless infrastructure that Fargate provides. When your tasks are run on Fargate, your cluster resources are managed by Fargate.

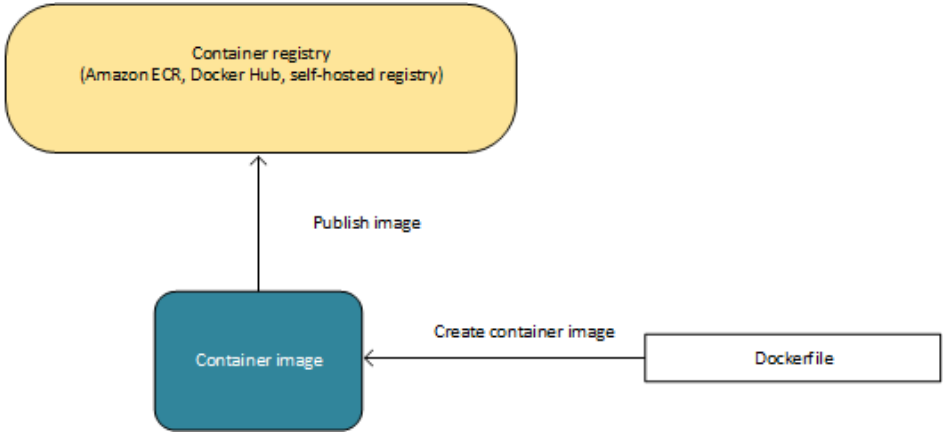
ECS – Container Agent

- The container agent runs on each container instance within an Amazon ECS cluster. It sends information about the resource's current running tasks and resource utilization to Amazon ECS, and starts and stops tasks whenever it receives a request from Amazon ECS



ECS – Container & Images

- To deploy applications on Amazon ECS, your application components must be architected to run in containers. A container is a standardized unit of software development, containing everything that your software application needs to run: code, runtime, system tools, system libraries, etc. Containers are created from a read-only template called an image.



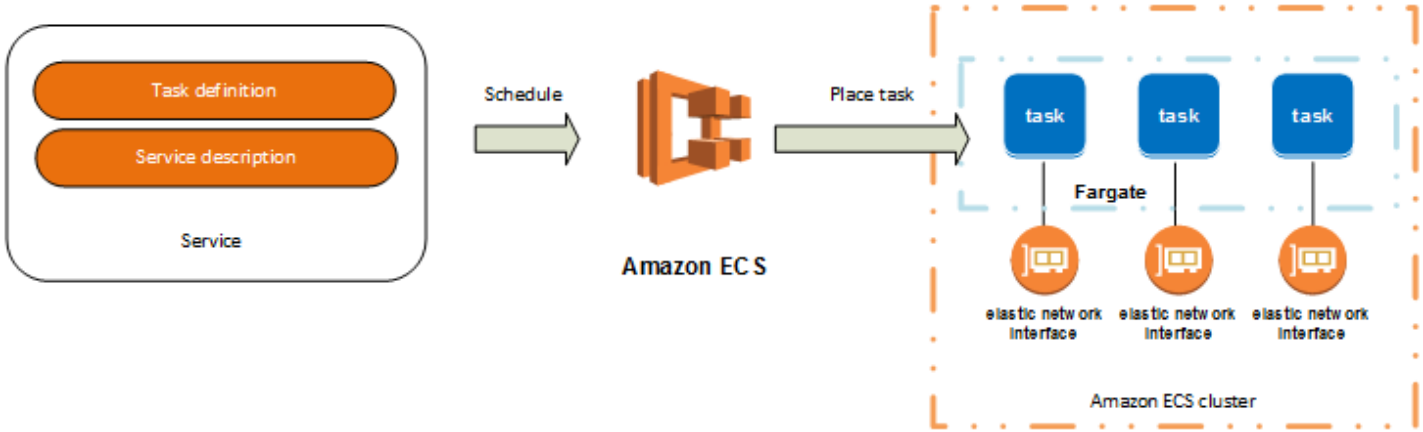
ECS – Task definitions

- To prepare your application to run on Amazon ECS, you create a task definition. The task definition is a text file, in JSON format, that describes one or more containers, up to a maximum of ten, that form your application

```
{
    "family": "webserver",
    "containerDefinitions": [
        {
            "name": "web",
            "image": "nginx",
            "memory": "100",
            "cpu": "99"
        },
    ],
    "requiresCompatibilities": [
        "FARGATE"
    ],
    "networkMode": "awsvpc",
    "memory": "512",
    "cpu": "256",
}
```

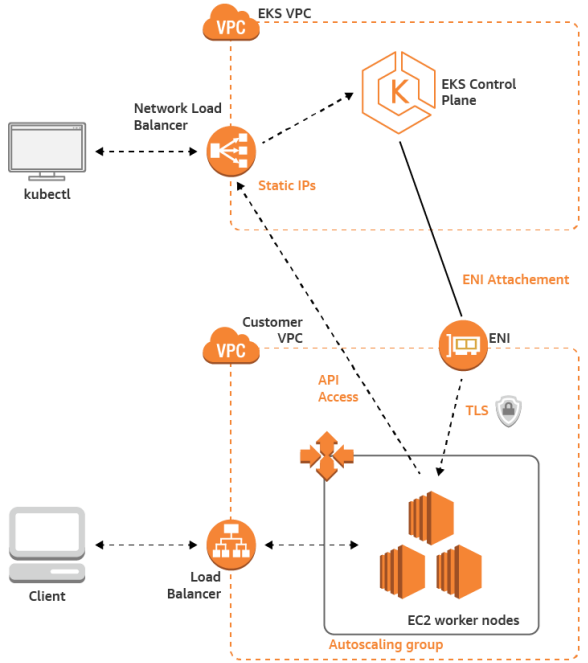
ECS – Tasks & Scheduling

- A task is the instantiation of a task definition within a cluster. After you have created a task definition for your application within Amazon ECS, you can specify the number of tasks that will run on your cluster.
- The Amazon ECS task scheduler is responsible for placing tasks within your cluster.



AWS EKS

- Amazon EKS is a managed service that makes it easy for you to run Kubernetes on AWS without needing to install, operate, and maintain your own Kubernetes control plane or nodes. Kubernetes is an open-source system for automating the deployment, scaling, and management of containerized applications

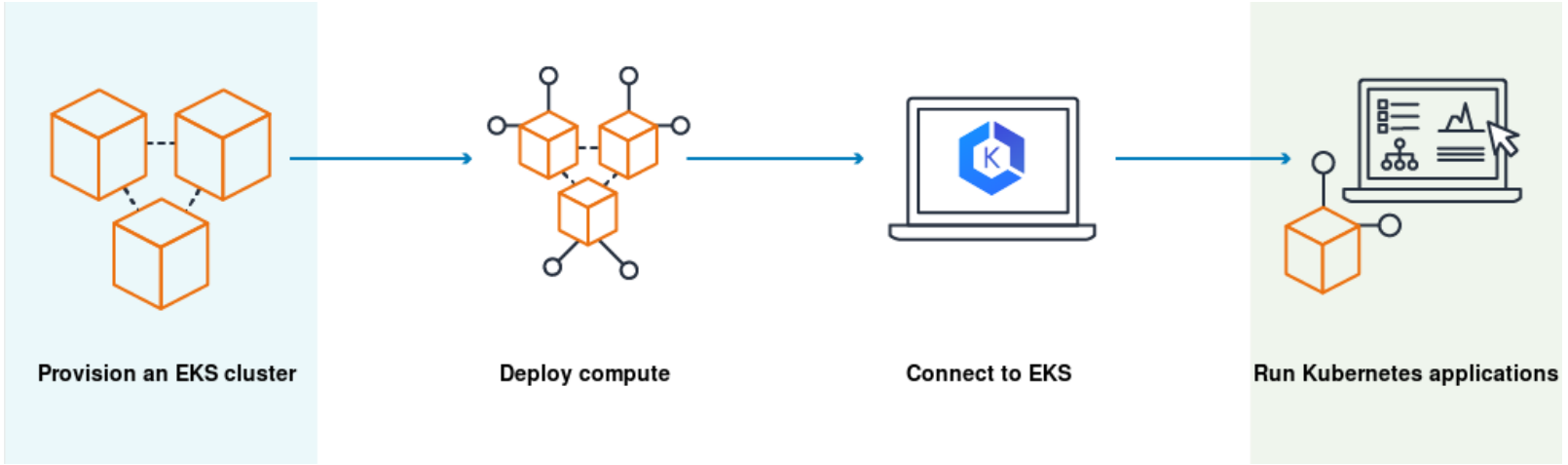


AWS EKS CONTROL PLANE

- Amazon EKS runs Kubernetes control plane instances across multiple Availability Zones to ensure high availability. Amazon EKS automatically detects and replaces unhealthy control plane instances, and it provides automated version upgrades and patching for them.

- Amazon EKS runs a single tenant Kubernetes control plane for each cluster, and control plane infrastructure is not shared across clusters or AWS accounts.

How does Amazon EKS work?



KUBERNETES

WHY CONTAINERS

- Build, Ship, and Run any App, Anywhere.
- No more dependency problems; e.x. no RPM/DEB or dynamic libraries incompatibility issues.
- No more: "but it works on my machine", as the same image used in dev is deployed to prod.
- Serverless implementations are mainly backed by containers.

WHY KUBERNETES

- Be able to manage hundreds or thousands of containers, on a fleet of hundred or thousands of nodes.
- Be able to deploy an application without worrying about the hardware infrastructure.
- Be able to guarantee that applications will stay running according with the specifications.
- Be able to facilitate scaling and upgrades not only for the workloads but for the k8s cluster itself.

WHAT IS KUBERNETES

- Kubernetes (K8s) is an open-source system for automating deployment, scaling, and management of containerized applications.
- Designed from the ground-up as a loosely coupled collection of components centered around deploying, maintaining and scaling workloads.
- Abstracts away the underlying hardware of the nodes and provides a uniform interface for workloads to be both deployed and consume the shared pool ofresources.
- A distributed cluster technology that manages container-based systems in a declarative manner using an API (a container orchestrator).

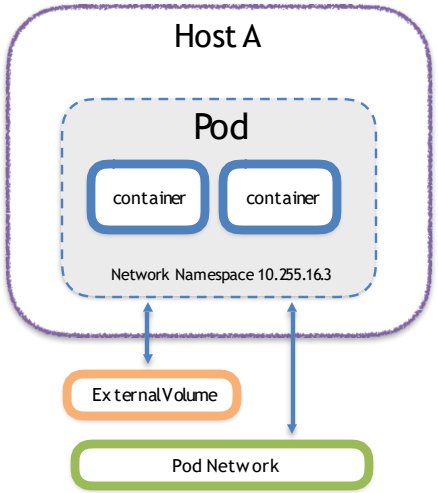
A COUPLE OF KEY CONCEPTS…

PODS

Ephemeral

Atomic unit or smallest "unit of work" of Kubernetes.

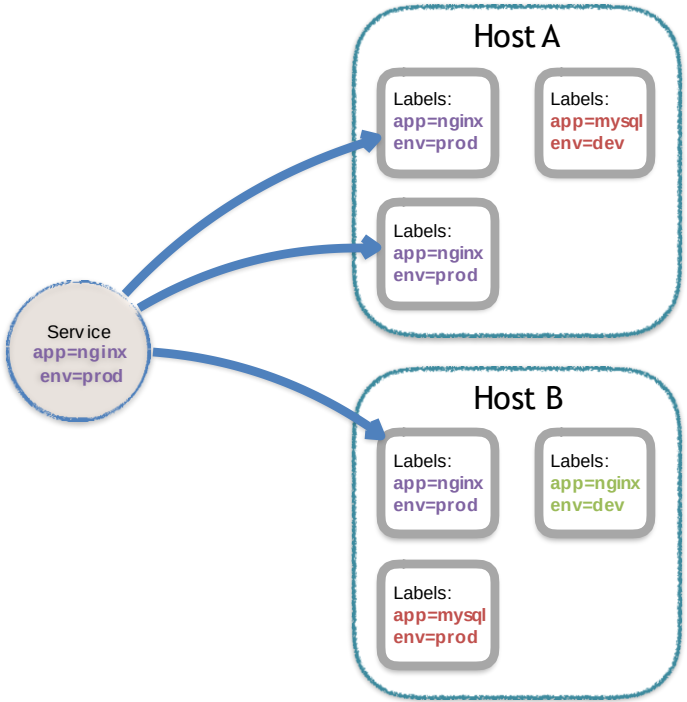
- Pods are **one or MORE containers** that share volumes, a network namespace, and are a part of a **single context.**



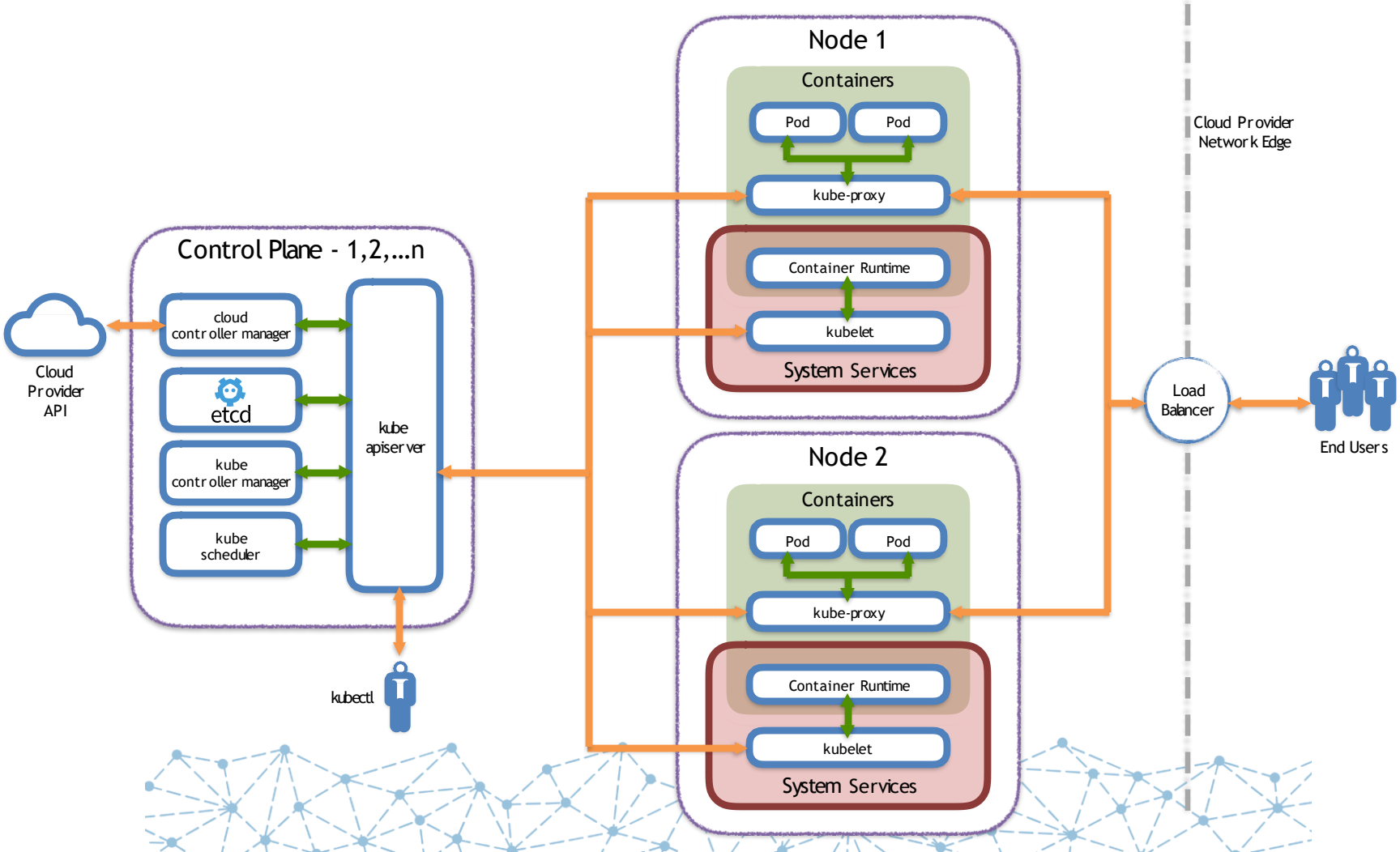
NOT Ephemeral

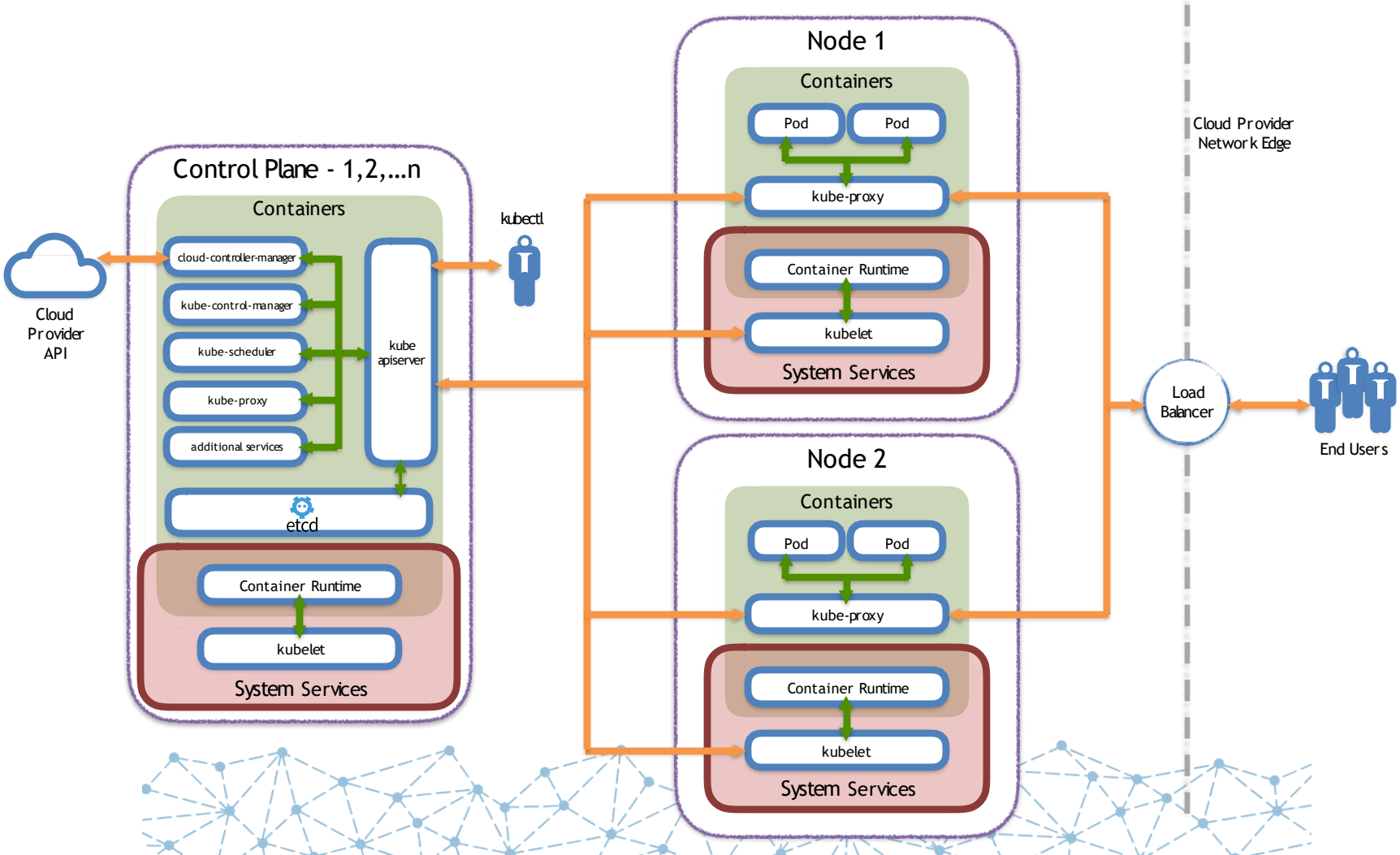
Unified method of accessing the exposed workloads of Pods.

- **Durable resource**
  - static cluster IP
  - static namespaced DNS name



ARCHITECTURE OVERVIEW


Node 1
Containers
Pod
Pod
kube-proxy
Container Runtime
kubelet
System Services
Cloud Provider
Network Edge
Control Plane - 1,2,...n
cloud
contr oller manager
etcd
kube
contr oller manager
kube
scheduler
kube
apiser ver
Cloud
Provider
API
kubectl
Load
Balancer
End Users
Node 2
Containers
Pod
Pod
kube-proxy
Container Runtime
kubelet
System Services


Control Plane - 1,2,...n
Containers
cloud-controller-manager
kube-control-manager
kube-scheduler
kube-proxy
additional services
kube apiserver
etcd
Container Runtime
kubelet
System Services
Cloud Provider API
kubectl
Node 1
Containers
Pod
Pod
kube-proxy
Container Runtime
kubelet
System Services
Node 2
Containers
Pod
Pod
kube-proxy
Container Runtime
kubelet
System Services
Cloud Provider Network Edge
Load Balancer
End Users

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.
- Acts as the gatekeeper to the cluster by handling authentication and authorization, request validation, mutation, and admission control in addition to being the front-end to the backing datastore.

ETCD

master

- 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.
- Stores objects and config information.
- Uses “Raft Consensus” among a quorum of systems to create a fault-tolerant consistent “view” of the cluster.

KUBE-CONTROLLER-MANAGER

- Serves as the primary daemon that manages all core component control loops.
- Monitors the cluster state via the apiserver and steers the cluster towards the desired state.

KUBE-SCHEDULER

master

- Component on the master that watches newly created pods that have no node assigned, and selects a node for them to run on.

- Factors taken into account for scheduling decisions include individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference and deadlines.

CLOUD-CONTROLLER-MANAGER

- Daemon that provides cloud-provider specific knowledge and integration capability into the core control loop of Kubernetes.

- The controllers include Node, Route, Service, and add an additional controller to handle things such as **PersistentVolume** Labels.

KUBE-PROXY

worker

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

KUBELET

worker

- An agent that runs on each node in the cluster. It makes sure that containers are running in a pod.

- The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy.

CONTAINER RUNTIME ENGINE

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

- **containerd** (Docker)
- **cri-o**
- **rkt**
- **kata** (formerly clear and hyper)
- **virtlet** (VM CRI compatible runtime)

- **Pod Network**
  - Cluster-wide network used for pod-to-pod communication managed by a CNI (Container Network Interface) plugin.

- **Service Network**
  - Cluster-wide range of Virtual IPs managed by **kube-proxy** for service discovery.

FUNDAMENTAL NETWORKING RULES

- All containers within a pod can communicate with each other unimpeded.
- All Pods can communicate with all other Pods without NAT.
- All nodes can communicate with all Pods (and vice-versa) without NAT.
- The IP that a Pod sees itself as is the same IP that others see it as.

What if I want to limit communication within Pods ?

Learn about **Network Policies** (and make sure the chosen CNI supports it)

Q&A

Session 8

Kubernetes (Cont.)

Agenda

- Assignment Review & Guides
- Kubernetes concepts(Namespace, Pod, Deployment, Labels/Selector, ReplicaSet)
- Working with Pod
- Lab: Deployment Rolling Update and Rollback

API

API OVERVIEW

- The **REST API** is the true **keystone** of Kubernetes.
- **Everything** within Kubernetes is as **an API Object**.
- Referenced within an object as the **apiVersion** and **kind**.

Format:
/apis/<group>/<version>/<resource>

Examples:
/apis/apps/v1/deployments
/apis/batch/v1beta1/cronjobs

OBJECT MODEL

- Objects are a "record of intent" or a persistent entity that represent the desired state of the object within the cluster.

- All objects **MUST** have apiVersion, kind, and poses the nested fields metadata.name, metadata.namespace, and metadata.uid.

OBJECT EXPRESSION - YAML

- Files or other representations of Kubernetes Objects are generally represented in YAML.

- Three basic datatypes:
  - **mappings** - hash or dictionary,
  - **sequences** - array or list
  - **scalars** - string, number, boolean etc

```
apiVersion: v1
kind: Pod
metadata:
  name: sample
  namespace: test
spec:
  containers:
  - name: container1
    image: nginx
  - name: container2
    image: alpine
```

YAML VS JSON

```yaml
apiVersion: v1
kind: Pod
metadata:
    name: pod-example
spec:
    containers:
    - name: nginx
      image: nginx:stable-alpine
        ports:
        - containerPort: 80
```

```json
{
 "apiVersion": "v1",
 "kind": "Pod",
 "metadata": {
     "name": "pod-example"
 },
 "spec": {
     "containers": [
         {
             "name":"nginx",
             "image": "nginx:stable-alpine",    [
             "ports":
               { "containerPort": 80 }
             ]
         }
       ]
     }
  }
```

Are you wondering about the YAML schema?
kubectl explain is your friend ;)

What about kinds or versions ?
kubectl api-versions is your friend ;)

CORE CONCEPTS

NAMESPACES

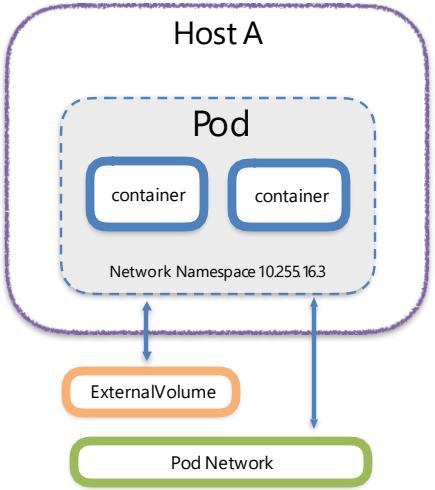
Namespaces are a logical cluster or environment, and are the primary method of partitioning a cluster or scoping access.

```
apiVersion: v1
kind: Namespace
metadata:
  name: prod
  labels:
    app: MyBigWebApp
```

```
$ kubectl get ns --show-labels
NAME              STATUS     AGE          LABELS
default           Active     11h          <none>
kube-public       Active     11h          <none>
kube-system       Active     11h          <none>
prod              Active     6s           app=MyBigW
                                          ebApp
```

PODS

- **Atomic unit** orsmallest "unit ofwork"of Kubernetes.
- Foundational building block of Kubernetes Workloads.
- Pods are one or MORE containers that share volumes, a network namespace, and are a part of a **single context**.




```
apiVersion: v1
kind: Pod
metadata:
  name: sample
spec:
  containers:
  - name: nginx
    image: nginx:stable-alpine
    ports:
    - containerPort: 80
      name: http
      protocol: TCP
    env:
    - name: MYVAR
      value: isAwesome
    command: ["/bin/sh", "-c"]
    args: ["echo ${MYVAR}"]
    readinessProbe:
      tcpSocket:
        port: http
        initialDelaySeconds: 10
        periodSeconds: 10
    livenessProbe:
      httpGet:
        path: /
        port: http
        initialDelaySeconds: 30
        periodSeconds: 60
```

Name of the container

Container Image

Array of environment variables

Array of ports to expose

Entrypoint Array ~ Docker ENTRYPOINT

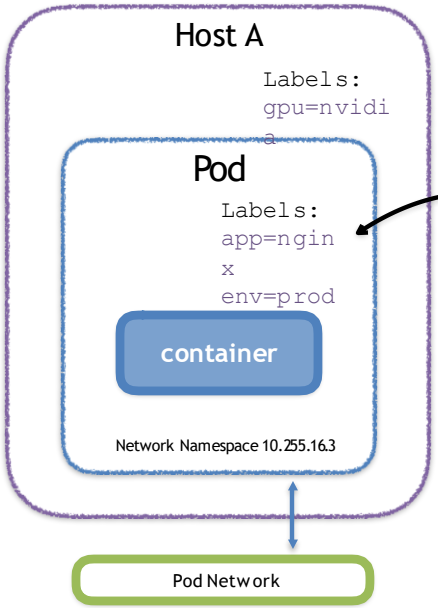
Arguments to pass to the command ~ Docker CMD

Tells when the application is ready to receive requests

Checks if the container is still alive (running)

LABELS

- key-value pairs that are used to identify, describe and group together related sets of objects or resources.
- NOT characteristic of uniqueness.
- Have a strict syntax with a slightly limited character set*.



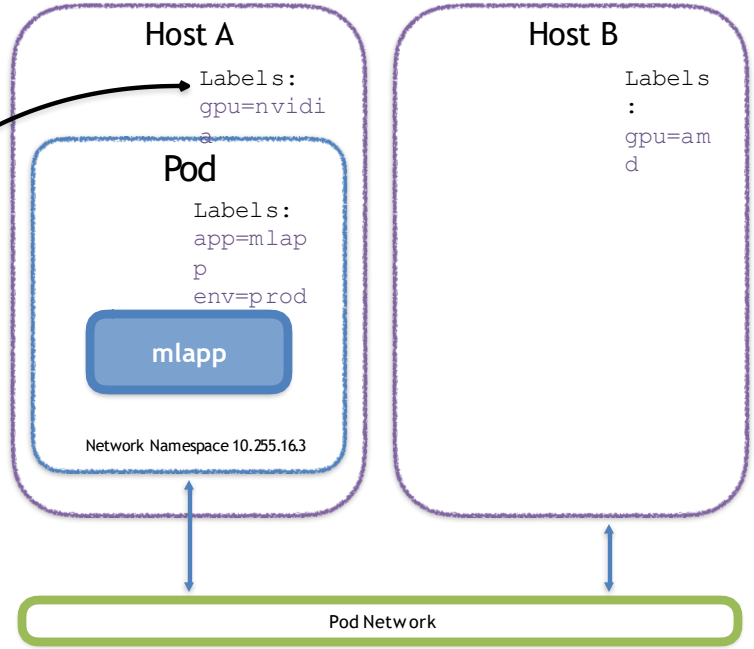

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-label-example
  labels:
    app: nginx
    env: prod
spec:
  containers:
  - name: nginx
    image: nginx:stable-alpine
    ports:
    - containerPort: 80
```

https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#syntax-and-character-set

SELECTORS

Selectors use labels to filter or select objects, and are used throughout Kubernetes.

```
apiVersion: v1
kind: Pod
metadata:
  name: mlapp
  labels:
    app: mlapp
    env: prod
spec:
  nodeSelector:
  - gpu: nvidia
  containers:
  - name: nginx
    image: tensorflow/tensorflow
```



SERVICES

- **Unified method of accessing** the exposed workloads of Pods.
- **Durable resource** (unlike Pods)
  - static cluster-unique IP
  - static namespaced DNS name

There are 4 major service types:

- ClusterIP (default)
- NodePort
- LoadBalancer
- ExternalName

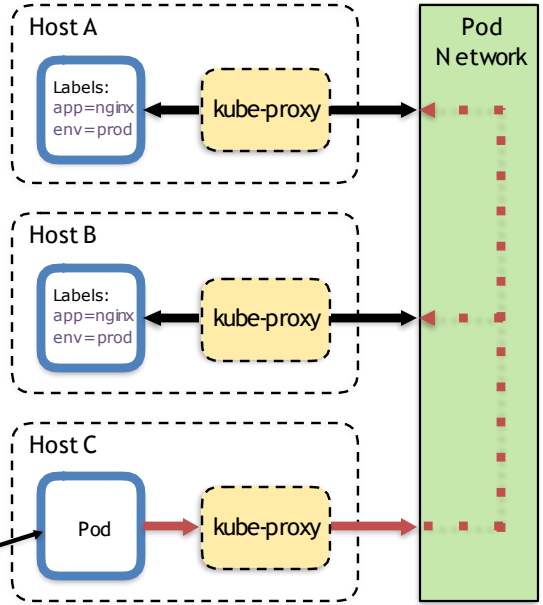
<service name>.<namespace>.svc.cluster.local

CLUSTER IP SERVICE

- The Pod on host C requests the service.
- Hits host iptables and it load-balances the connection between the endpoints residing on Hosts A,B

```
Name:           example-prod
Selector:       app=nginx,env=prod
Type:           ClusterIP
IP:             10.96.28.176
Port:           <unset>  80/TCP
TargetPort:     80/TCP
Endpoints:      10.255.16.3:80,
                10.255.16.4:80
```

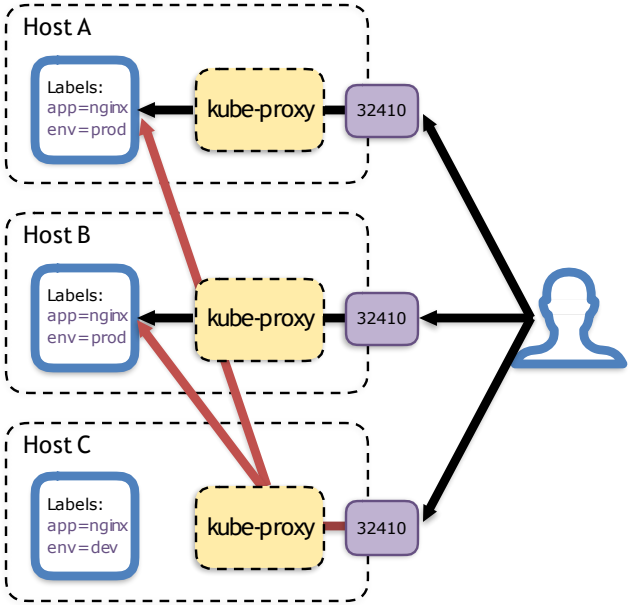
```
/ # nslookup example-prod.default.svc.cluster.local

Name:      example-prod.default.svc.cluster.local
Address 1: 10.96.28.176 example-prod.default.svc.cluster.local
```



NODE PORT SERVICE

- User can hit any host in cluster on **NodePort** IP and get to service.
- Does introduce extra hop if hitting a host without instance of the pod.

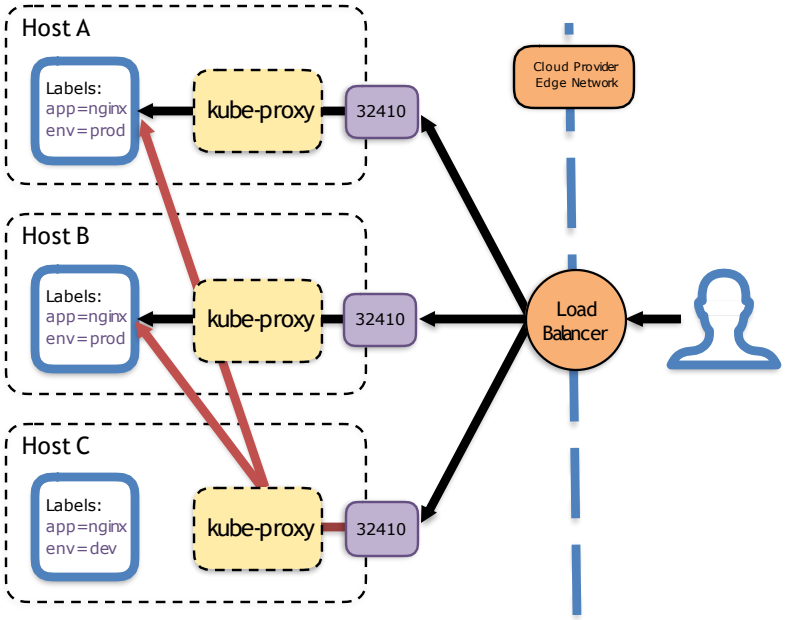
```
Name:           example-prod
Selector:       app=nginx,env=prod
Type:           NodePort
IP:             10.96.28.176
Port:           <unset>  80/TCP
TargetPort:     80/TCP
NodePort:       <unset>  32410/TCP
Endpoints:      10.255.16.3:80,
                10.255.16.4:80
```



LOAD BALANCER SERVICE

- **LoadBalancer** services extend **NodePort**.
- Works in conjunction with an external system to map acluster external IP to the exposed service.

```
Name:           example-prod
Selector:       app=nginx,env=prod
Type:           LoadBalancer
IP:             10.96.28.176
LoadBalancer
Ingress:        172.17.18.43
Port:           <unset>  80/TCP
TargetPort:     80/TCP
NodePort:       <unset>  32410/TCP
Endpoints:      10.255.16.3:80,
                10.255.16.4:80
```



EXTERNAL NAME SERVICE

- **ExternalName** is used to reference endpoints **OUTSIDE** the cluster.
- Creates an internal **CNAME** DNS entry that aliases another.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: example-prod
spec:
  type: ExternalName
spec:
  externalName: example.com
```

WORKLOADS

WORKLOADS

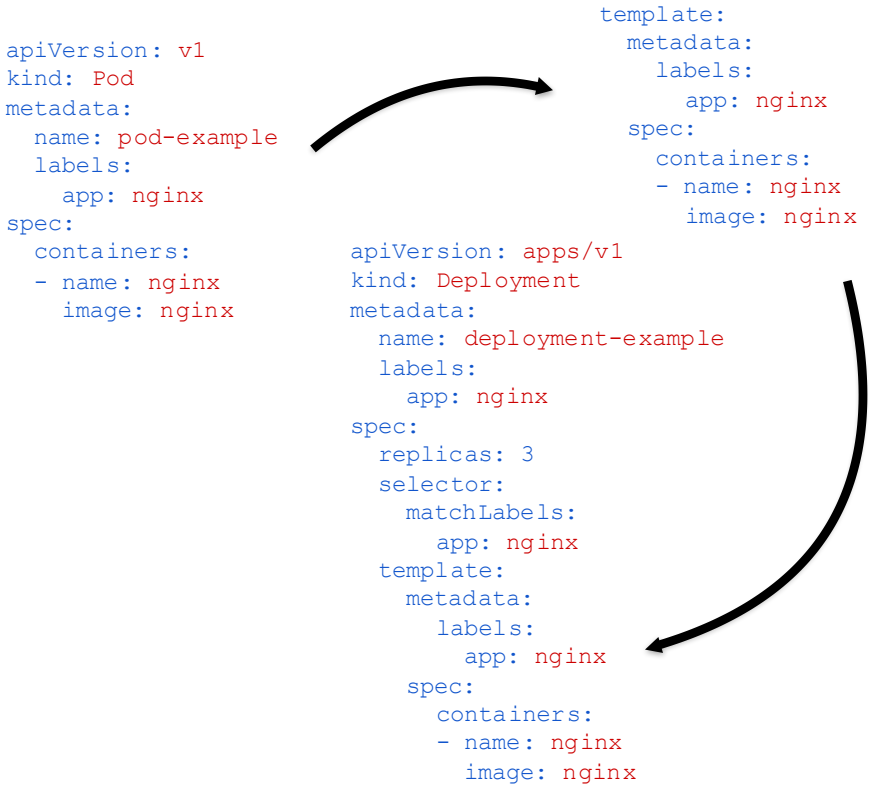
- Workloads within Kubernetes are higher level objects that manage Pods or other higher level objects.

- In **ALL CASES** a Pod Template is included, and acts the base tier of management.

- ReplicaSet
- Deployment
- DaemonSet
- StatefulSet
- Job
- CronJob

POD TEMPLATE

- Workload Controllers manage instances of Pods based off a provided template.
- Pod Templates are Pod specs with limited metadata.
- Controllers use Pod Templates to make actual pods



RESOURCE MODEL

- **Request**: amount of a resource allowed to be used, with a strong guarantee of availability.
  - CPU (seconds/second), RAM (bytes)
  - Scheduler will not over-commit requests
- **Limit**: max amount of a resource that can be used, regardless of guarantees
  - Scheduler ignores limits

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-example
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

INIT CONTAINERS

If we compare a container template with a Class definition in Java, an initContainer would be the constructor of a class; while a running Pod, would be an instance of thatclass.

Can be used for everything that should happen before the containers within a Pod start running. For example: wait for dependencies, initialize volumes or databases, verify requirements, etc.

REPLICA SET

- Primary method of managing pod replicas and their lifecycle.
- Includes their scheduling, scaling, and deletion.
- Their job is simple: **Always ensure the desired number of pods are running.**

- **replicas**: The desired number of instances of the Pod.
- **selector**: The label selector for the **ReplicaSet** will manage **ALL** Pod instances that it targets; whether it's desired or not.

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: rs-example
spec:
  replicas: 3
  selector:
    matchLabels:
      app:  nginx
      env: prod
  template:
    <pod template>
```

DEPLOYMENT

- Declarative method of managing Pods via ReplicaSets.
- Provide rollback functionality and update control.
- Updates are managed through the **pod-template-hash label**.
- Each iteration creates a unique label that is assigned to both the **ReplicaSet** and subsequent Pods.

- **revisionHistoryLimit**: The number of previous iterations of the Deployment to retain.
- **strategy**: Describes the method of updating the Pods based on the **type**. Valid options are:
  - **Recreate**: All existing Pods are killed before the new ones are created.
  - **RollingUpdate**: Cycles through updating the Pods according to the parameters: **maxSurge** and **maxUnavailable**.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-example
spec:
  replicas: 3
  revisionHistoryLimit: 3
  selector:
    matchLabels:
      app: nginx
      env: prod
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
  template:
    <pod template>
```

DAEMON SET

- Ensure that all nodes matching certain criteria will run an instance of the supplied Pod.
- They **bypass** default scheduling mechanisms.
- Are ideal for cluster wide services such as log forwarding, or health monitoring.
- Revisions are managed via a **controller-revision-hash** label.

- **spec.template.spec.nodeSelector**: The primary selector used to target nodes.
- **Default Host Labels**:
  - kubernetes.io/hostname
  - beta.kubernetes.io/os
  - beta.kubernetes.io/arch
- **Cloud Host Labels**:
  - failure-domain.beta.kubernetes.io/zone
  - failure-domain.beta.kubernetes.io/region
  - beta.kubernetes.io/instance-type

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: ds-example
spec:
  revisionHistoryLimit: 3
  selector:
    matchLabels:
      app: nginx
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
   template:
     spec:
      nodeSelector:
        nodeType: edge
      <pod template>
```

STATEFULSET

- Tailored to managing Pods that must persist or maintain state.
- Pod identity including **hostname**, **network**, and **storage** **WILL** be persisted.
- Assigned a unique ordinal name following the convention of '<statefulset name>-<ordinal index>'.
- Naming convention is also used in Pod's network Identity and Volumes.
- Pod lifecycle will be ordered and follow consistent patterns.
- Revisions are managed via a **controller-revision-hash** label.


```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: sts-cassandra
spec:
  replicas: 3
  selector:
    matchLabels:
      app: cassandra
  serviceName: cassandra
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      partition: 0
  template:
    metadata:
      labels:
        app: cassandra
    spec:
      containers:
      - name: cassandra-node
        image: cassandra:3.11.4
        env:
        - name: CASSANDRA_SEEDS
          value: sts-cassandra-0.cassandra
        - name: CASSANDRA_CLUSTER_NAME
          value: my-cluster
      ports:
        - containerPort: 7000

        - containerPort: 7199

        - containerPort: 9042

      volumeMounts:
      - name: data
        mountPath: /cassandra_data
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes: [ "ReadWriteOnce"
      ]
      storageClassName:
      standard  resources:
        requests:
          storage: 100Gi
```

The name of the associated headless service; or a service without a ClusterIP

Pods with an ordinal greater than the partition value will be updated one-by-one in reverse order

Template of the persistent volume(s) request to use for each instance of the StatefulSet.

HEADLESS SERVICE

<StatefulSet Name>-<ordinal>.<service name>.<namespace>.svc.cluster.local

```
apiVersion: v1
kind: Service
metadata:
  name: cassandra
spec:
  clusterIP: None
  selector:
    app: cassandra
  ports:
  - name: intra-node
    port: 80
  - name: jmx
    port: 7199
  - name: cql
    port: 9042
```

```
$ kubectl get pods
NAME              READY     STATUS    RESTARTS   AGE
sts-cassandra-0   1/1       Running   0          11m
sts-cassandra-1   1/1       Running   0          11m
sts-cassandra-2   1/1       Running   0          11m
```

```
/ # dig cassandra.default.svc.cluster.local +noall +answer

; <<>> DiG 9.9.4-RedHat-9.9.4-74.el7_6.1 <<>> cassandra.default.svc.cluster.local +noall +answer
;; global options: +cmd
  cassandra.default.svc.cluster.local.  5 IN A   172.17.0.5

  cassandra.default.svc.cluster.local.  5 IN A   172.17.0.4

  cassandra.default.svc.cluster.local.  5 IN A   172.17.0.6

/ # dig sts-cassandra-0.cassandra.default.svc.cluster.local +noall +answer

; <<>> DiG 9.9.4-RedHat-9.9.4-74.el7_6.1 <<>> sts-cassandra-0.cassandra.default.svc.cluster.local +noall +answer

/ # dig sts-cassandra-1.cassandra.default.svc.cluster.local +noall +answer

; <<>> DiG 9.9.4-RedHat-9.9.4-74.el7_6.1 <<>> sts-cassandra-1.cassandra.default.svc.cluster.local +noall +answer

/ # dig sts-cassandra-2.cassandra.default.svc.cluster.local +noall +answer

; <<>> DiG 9.9.4-RedHat-9.9.4-74.el7_6.1 <<>> sts-cassandra-2.cassandra.default.svc.cluster.local +noall +answer
```

VOLUME CLAIM TEMPLATE

<Volume Name>.<StatefulSet Name>-<ordinal>

```
volumeClaimTemplates:
 - metadata:
     name: data
   spec:
     accessModes: [ "ReadWriteOnce" ]
     storageClassName: standard
     resources:
       requests:
         storage: 1Gi
```

Persistent Volumes associated with a StatefulSet will **NOT** be automatically garbage collected when its associated StatefulSet is deleted. They must manually be removed.

```
$ kubectl get pvc
NAME                    STATUS    VOLUME                                     CAPACITY   ACCES MODES   STORAGECLASS   AGE
                                                                                        S
data-cassandra-sts-0    Bound     pvc-8baec2a5-8c77-11e9-a6be-0800275ddaf0   100Gi      RWO           standard       17m
data-cassandra-sts-1    Bound     pvc-95463714-8c77-11e9-a6be-0800275ddaf0   100Gi      RWO           standard       17m
data-cassandra-sts-2    Bound     pvc-9807c42a-8c77-11e9-a6be-0800275ddaf0   100Gi      RWO           standard       17m
```

JOB

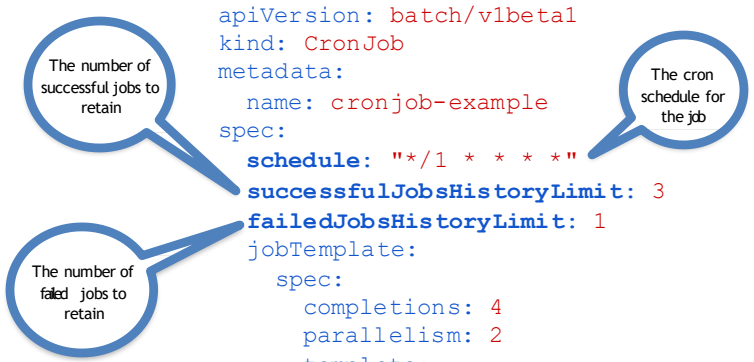
- Job controller ensures one or more pods are executed and successfully terminate.
- Will continue to try and execute the job until it satisfies the completion and/or parallelism condition.
- Pods are NOT cleaned up until the job itself is deleted.

- **backoffLimit**: The number of failures before the job itself is considered failed.
- **completions**: The total number of successful completions desired.
- **parallelism**: How many instances of the pod can be run concurrently.
- **spec.template.spec.restartPolicy**:Jobs only support a restartPolicy of type **Never** or **OnFailure**.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job-example
spec:
  backoffLimit: 4
  completions: 4
  parallelism: 2
  template:
    spec:
      restartPolicy:Never
        template:
        <pod-template>
```

CRONJOB

- An extension of the Job Controller, it provides a method of executing jobs on a cron-like schedule.
- CronJobs within Kubernetes use **UTC ONLY.**



OPERATORS

- Might be the most powerful feature of K8s.
- When none of the workloads fits the needs of your app, you can create your own controllers with your own specifications (CRD).
- Should be implemented in Go, but there are alternatives.
- A given controller usually requires a ServiceAccount with proper RBAC Role to be able to manage workloads (e.x. create and maintain Pods, Services, etc.)

```
apiVersion: acid.zalan.do/v1
kind: postgresql
metadata:
  name: opennms-database
spec:
  teamId: OpenNMS
  volume:
    size: 100Gi
  numberOfInstances: 3
  enableMasterLoadBalancer: false
  enableReplicaLoadBalancer: false
  users:
    opennms:
    - superuser
  - createdb
  databases:
    opennms: opennms
  postgresql:
    version: "10"
```

LAB

Q&A

Session 9

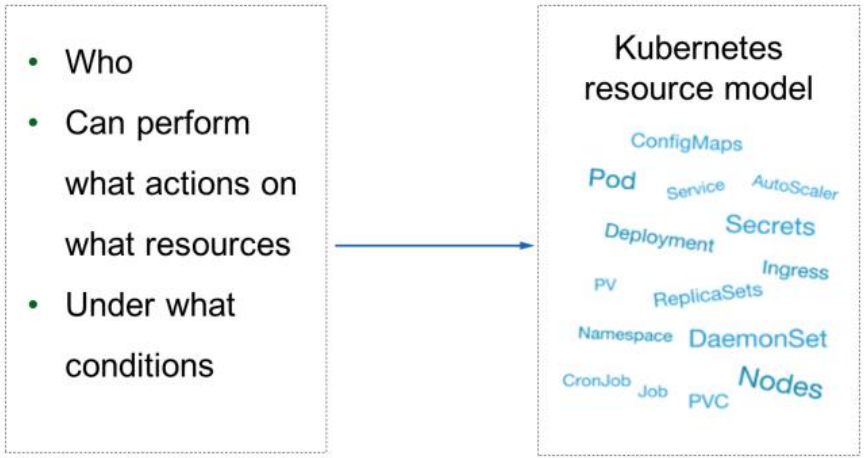
Kubernetes (Cont.)

Agenda

- Assignment Review & Guides
- Kubernetes authentication, authorization and admission control
- Ingress and ingress controller

Access Control

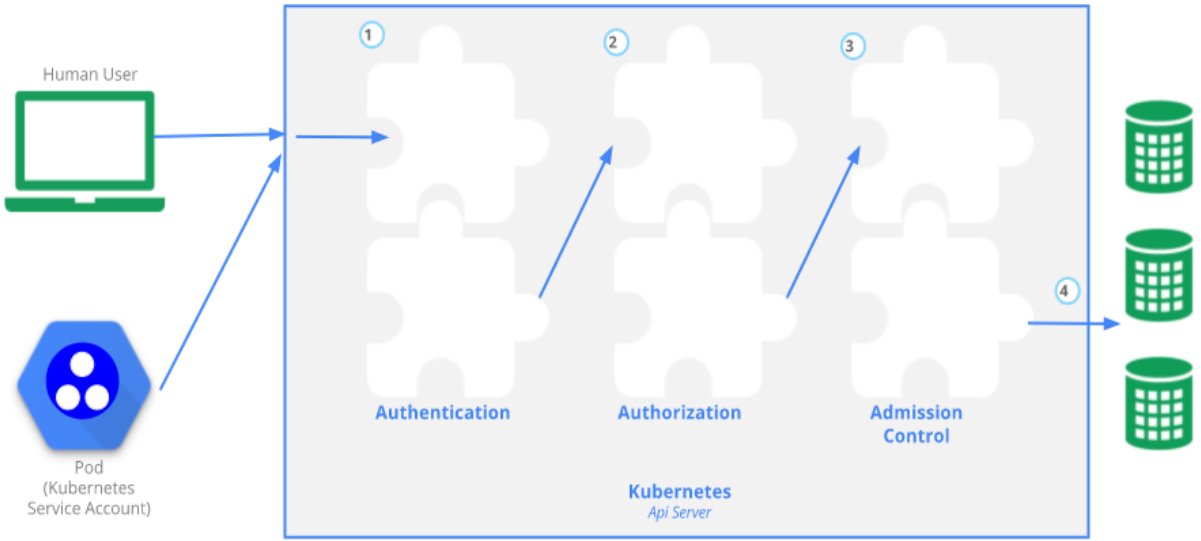
Access control is an important part of cloud-native security. It is also a basic security measure that a Kubernetes cluster must take in a multi-tenant environment



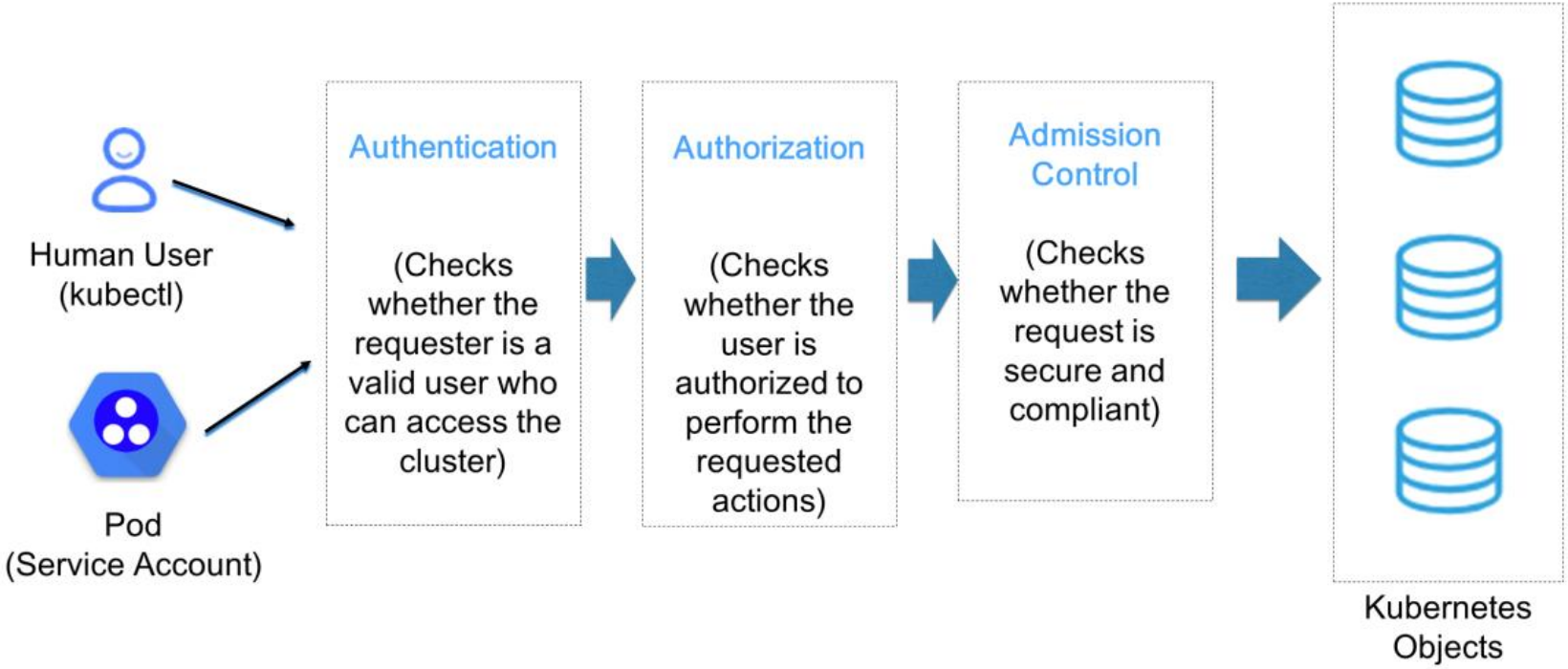
Controlling Access to the Kubernetes API

Users access the Kubernetes API using kubectl, client libraries, or by making REST requests.

Both human users and Kubernetes service accounts can be authorized for API access

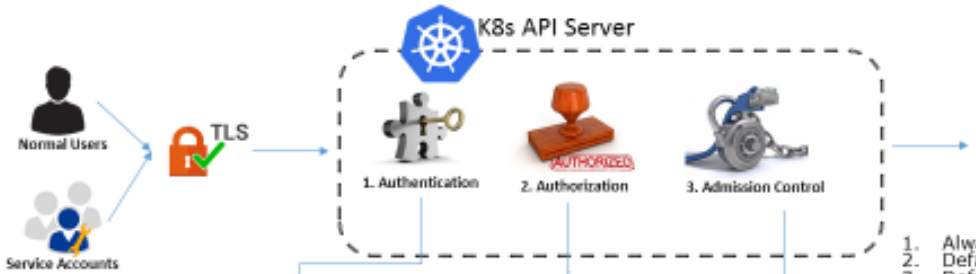


Controlling Access to the Kubernetes API



Controlling Access to the Kubernetes API

Accessing the Kubernetes API



1. X509 Client Certs
2. Static Token File
3. Bootstrap Tokens
4. Static Password File
5. Service Account Tokens
6. OpenID Connect Tokens
7. Webhook Token authentication
8. Authenticating Proxy
9. Anonymous
10. User Impersonation

- Authorization Modules:

1. Node
2. ABAC
3. RBAC
4. RBAC
5. Webhook

- Using Flags for your authorization Module
- Privilege escalation visa pod creation.

1. AlwaysPullImages
2. DefaultStorageClass
3. DefaultTolerationSeconds
4. DenyExecOnPrivileges (Depr)
5. DenyEscalatingExec
6. EventRateLimit
7. ExtendedResourceToleraton
8. ImagePolicyWebhook
9. Initializers
10. LimitPodHardAntiAffinityTopology
11. LimitRanger
12. MutatingAdmissionWebhook
13. NamespaceAutoProvision
14. NamespaceExists
15. NamespaceLifecycle
16. NodeRestriction
17. OwnerReferencesPermissionEnforcement
18. PersistentVolumeLabel (Depr)
19. PodNodeSelector
20. PersistentVolumeClaimResize
21. PodPreset
22. PodSecurityPolicy
23. PodTolerationRestriction
24. Priority
25. ResourceQuota
26. SecurityContextDeny
27. ValidatingAdmissionWebhook

Kubernetes authentication

Users in Kubernetes

All Kubernetes clusters have two categories of users:

- Service accounts managed by Kubernetes
- Normal users

Kubernetes does not have objects which represent normal user accounts. Normal users cannot be added to a cluster through an API call.

Service accounts are users managed by the Kubernetes API. They are bound to specific namespaces, and created automatically by the API server or manually through API calls

Authentication strategies

Kubernetes uses client certificates, bearer tokens, an authenticating proxy, or HTTP basic auth to authenticate API requests through authentication plugins

You can enable multiple authentication methods at once. You should usually use at least two methods:

- Service account tokens for service accounts
- At least one other method for user authentication.

X509 Client Certs Authentication

This authentication mode is often used by the API server. The API server starts the Transport Layer Security (TLS)-based handshake process when receiving an access request that is initiated through the client certificate which is signed by the cluster-dedicated Certificate Authority (CA) or by the trusted CA in the API server's client CA.

The API server checks the client certificate validity and verifies the information in the certificate, such as the request source address. X.509 authentication supports two-way authentication and provides robust security.

By default, it is used by Kubernetes components to authenticate each other and provides access credentials that are often used by kube-config for the kubectl client

Service Account Tokens

A service account is an automatically enabled authenticator that uses signed bearer tokens to verify requests

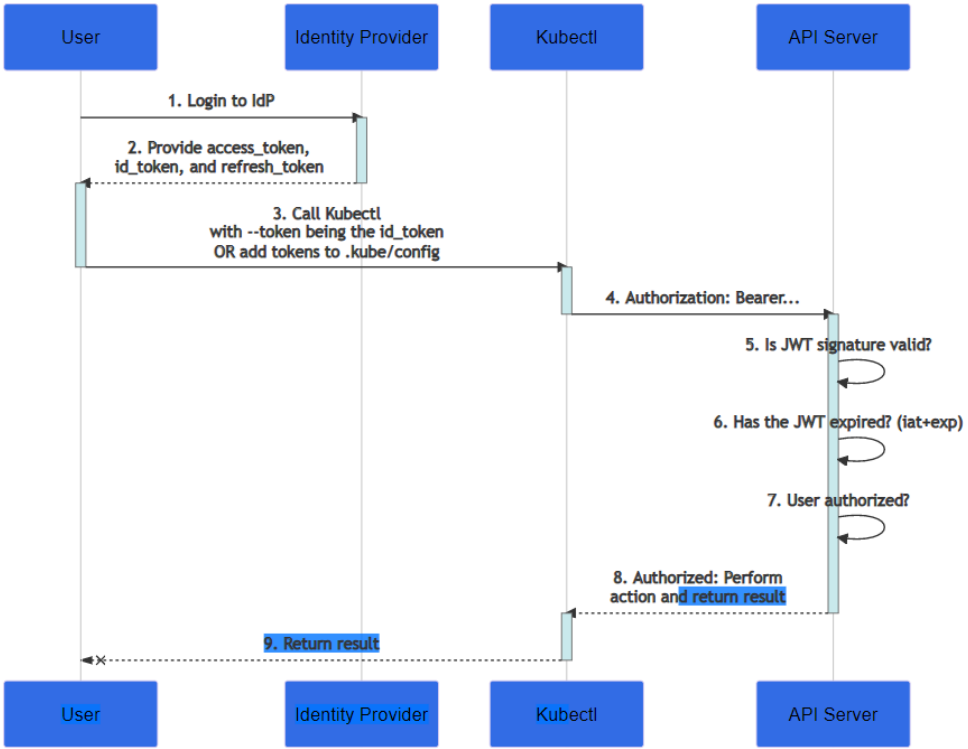
Service accounts are usually created automatically by the API server and associated with pods running in the cluster. Bearer tokens are mounted into pods at well-known locations, and allow in-cluster processes to talk to the API server

```yaml
apiVersion: apps/v1 # this apiVersion is relevant as of Kubernetes 1.9
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: default
spec:
  replicas: 3
  template:
    metadata:
    # ...
    spec:
      serviceAccountName: bob-the-bot
      containers:
      - name: nginx
        image: nginx:1.14.2
```

OpenID Connect Tokens

OpenID Connect is a flavor of OAuth2 supported by some OAuth2 providers, notably Azure Active Directory, Salesforce, and Google

To identify the user, the authenticator uses the id_token (not the access_token) from the OAuth2 token response as a bearer token



Authorization Overview

In Kubernetes, you must be authenticated (logged in) before your request can be authorized (granted permission to access)

Kubernetes expects attributes that are common to REST API requests. This means that Kubernetes authorization works with existing organization-wide or cloud-provider-wide access control systems which may handle other APIs besides the Kubernetes API.

Determine Whether a Request is Allowed or Denied

Kubernetes authorizes API requests using the API server. It evaluates all of the request attributes against all policies and allows or denies the request. All parts of an API request must be allowed by some policy in order to proceed. This means that permissions are denied by default.

When multiple authorization modules are configured, each is checked in sequence. If any authorizer approves or denies a request, that decision is immediately returned and no other authorizer is consulted. If all modules have no opinion on the request, then the request is denied. A deny returns an HTTP status code 403

Authorization Modes

The Kubernetes API server may authorize a request using one of several authorization modes:

- Node
- ABAC (Attribute-based access control)
- RBAC (Role-based access control)
- Webhook

Checking API Access

kubectl provides the **auth can-i** subcommand for quickly querying the API authorization layer. The command uses the **SelfSubjectAccessReview** API to determine if the current user can perform a given action, and works regardless of the authorization mode used.

```
kubectl auth can-i create deployments --namespace dev
```

Using RBAC Authorization

From a practical standpoint, the most useful authorization method is RBAC. It is based on declarative permissions definitions and cluster API objects. The main objects are roles and cluster roles, both representing a set of permissions on certain objects in the API

A **role** can only exist in a certain namespace and can only refer to namespaced resources. It also only defines permission to objects within that namespace.

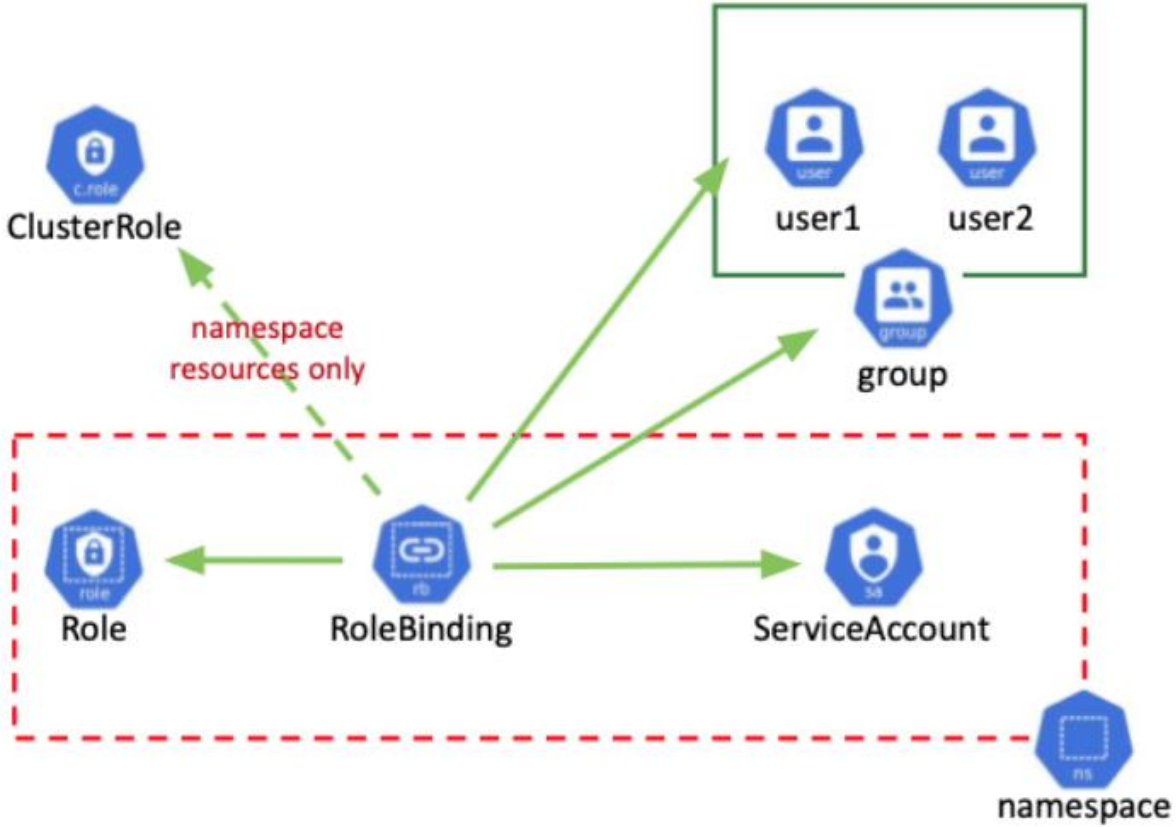
A **cluster role**, on the other hand, is a global object that will provide access to global objects as well as non-resource URLs, such as API version and healthz endpoints on the API server.

RoleBinding and ClusterRoleBinding

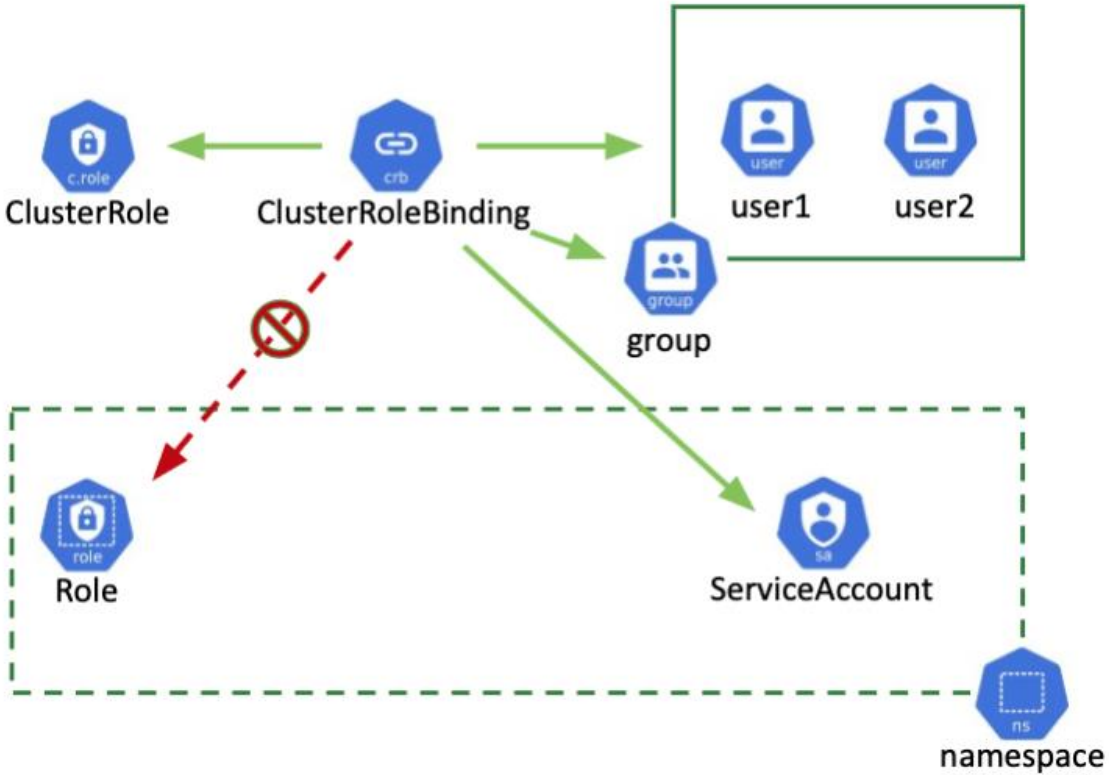
A role binding grants the permissions defined in a role to a user or set of users. It holds a list of subjects (users, groups, or service accounts), and a reference to the role being granted. A RoleBinding grants permissions within a specific namespace whereas a ClusterRoleBinding grants that access cluster-wide.

A RoleBinding may reference any Role in the same namespace. Alternatively, a RoleBinding can reference a ClusterRole and bind that ClusterRole to the namespace of the RoleBinding. If you want to bind a ClusterRole to all the namespaces in your cluster, you use a ClusterRoleBinding

RoleBinding and ClusterRoleBinding



RoleBinding and ClusterRoleBinding



Using ABAC Authorization

Attribute-based access control (ABAC) defines an access control paradigm whereby access rights are granted to users through the use of policies which combine attributes together

To enable ABAC mode, specify --authorization-policy-file=SOME_FILENAME and --authorization-mode=ABAC on startup.

The file format is one JSON object per line. There should be no enclosing list or map, only one map per line. Each line is a "policy object", where each such object is a map with the properties

```
{"apiVersion": "abac.authorization.kubernetes.io/v1beta1", "kind": "Policy", "spec": {"user": "alice", "namespace": "*", "resource": "*", "apiGroup": "*"}}
```

Using Node Authorization

Node authorization is a special-purpose authorization mode that specifically authorizes API requests made by kubelets

The Node authorizer allows a kubelet to perform API operations including:

- Read operations
- Write operations
- Auth-related operations

Webhook Mode

A WebHook is an HTTP callback: an HTTP POST that occurs when something happens; a simple event-notification via HTTP POST. A web application implementing WebHooks will POST a message to a URL when certain things happen.

When specified, mode Webhook causes Kubernetes to query an outside REST service when determining user privileges

Using Admission Controllers

An admission controller is a piece of code that intercepts requests to the Kubernetes API server prior to persistence of the object, but after the request is authenticated and authorized

There are two special controllers: MutatingAdmissionWebhook and ValidatingAdmissionWebhook. These execute the mutating and validating (respectively) admission control webhooks which are configured in the API

Turn on an admission controller

The Kubernetes API server flag enable-admission-plugins takes a comma-delimited list of admission control plugins to invoke prior to modifying objects in the cluster.

For example, the following command line enables the NamespaceLifecycle and the LimitRanger admission control plugins:

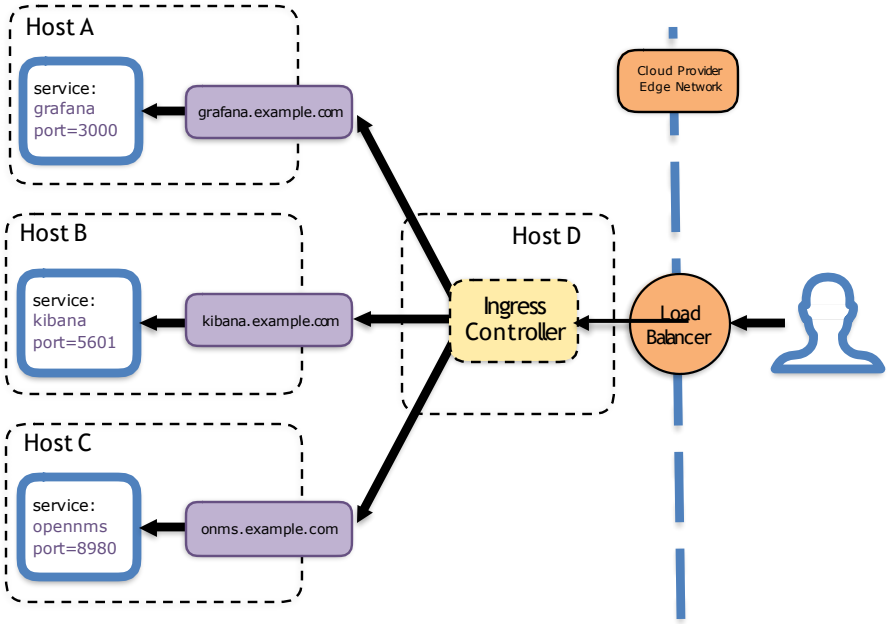
```
kube-apiserver --enable-admission-plugins=NamespaceLifecycle,LimitRanger ...
```

Kubernetes ingress

Known as Routes in OpenShift

- An API object that manages external access to the services in a cluster, through a single LoadBalancer.
- Provides load balancing, SSL termination and name/path-based virtual hosting
- Gives services externally-reachable URLs.
- Multiple implementations to choose from.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-rules
  annotations:
    kubernetes.io/ingress.class: nginx
spec:
 rules:
    - host: onms.example.com
      http:
        paths:
          - path: /
            backend:
              serviceName: opennms
              servicePort: 8980
…
```



The Ingress resource

A minimal Ingress resource example:

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: minimal-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
      paths:
      - path: /testpath
        pathType: Prefix
        backend:
          service:
            name: test
            port:
              number: 80
```

Types of Ingress

Ingress backed by a single Service

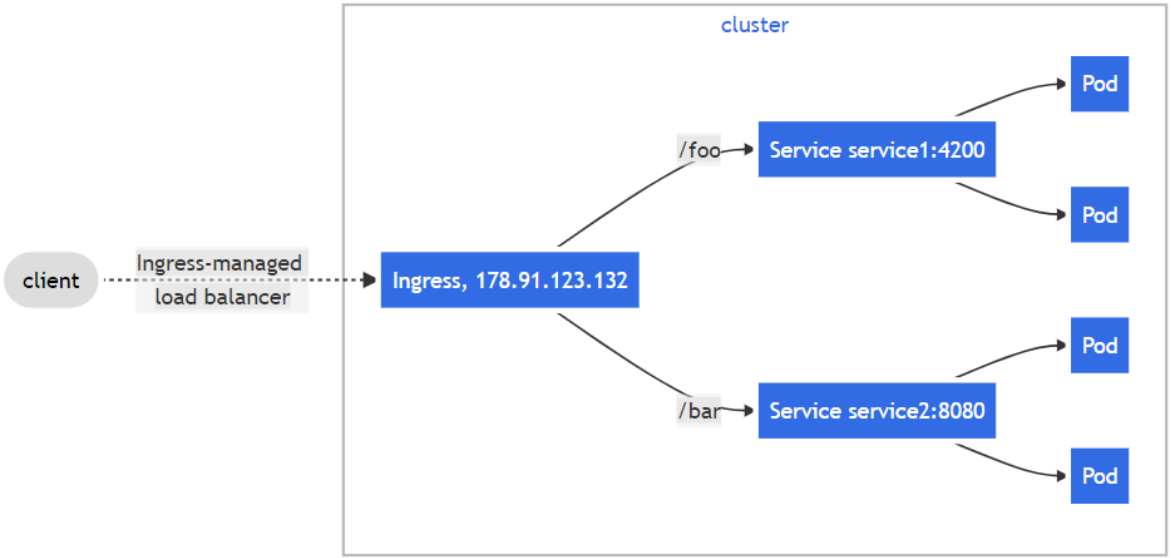
There are existing Kubernetes concepts that allow you to expose a single Service
You can also do this with an Ingress by specifying a default backend with no rules

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: test-ingress
spec:
  defaultBackend:
    service:
      name: test
      port:
        number: 80
```

Types of Ingress

Simple fanout

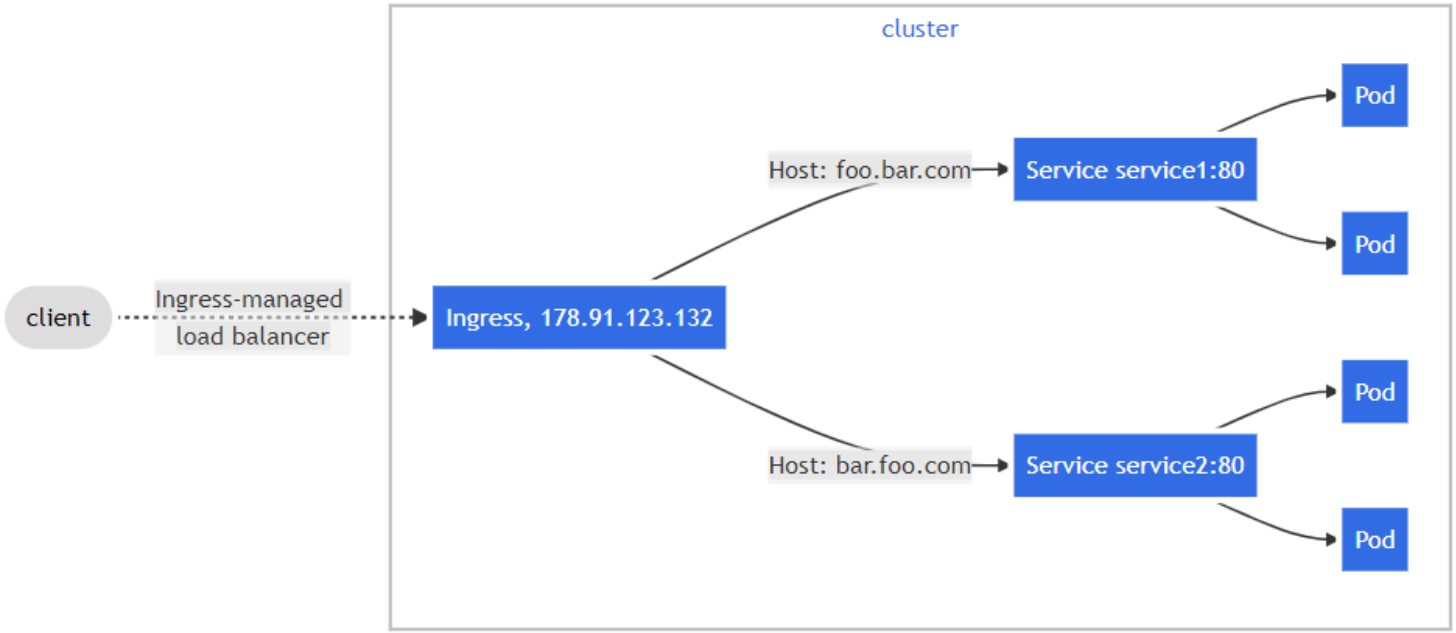
A fanout configuration routes traffic from a single IP address to more than one Service, based on the HTTP URI being requested. An Ingress allows you to keep the number of load balancers down to a minimum



Types of Ingress

Name based virtual hosting

Name-based virtual hosts support routing HTTP traffic to multiple host names at the same IP address.



Ingress Controllers

In order for the Ingress resource to work, the cluster must have an ingress controller running.

Unlike other types of controllers which run as part of the kube-controller-manager binary, Ingress controllers are not started automatically with a cluster

Kubernetes as a project supports and maintains AWS, GCE, and nginx ingress controllers

LAB

Q&A

Session 10

Kubernetes (Cont.)

Agenda

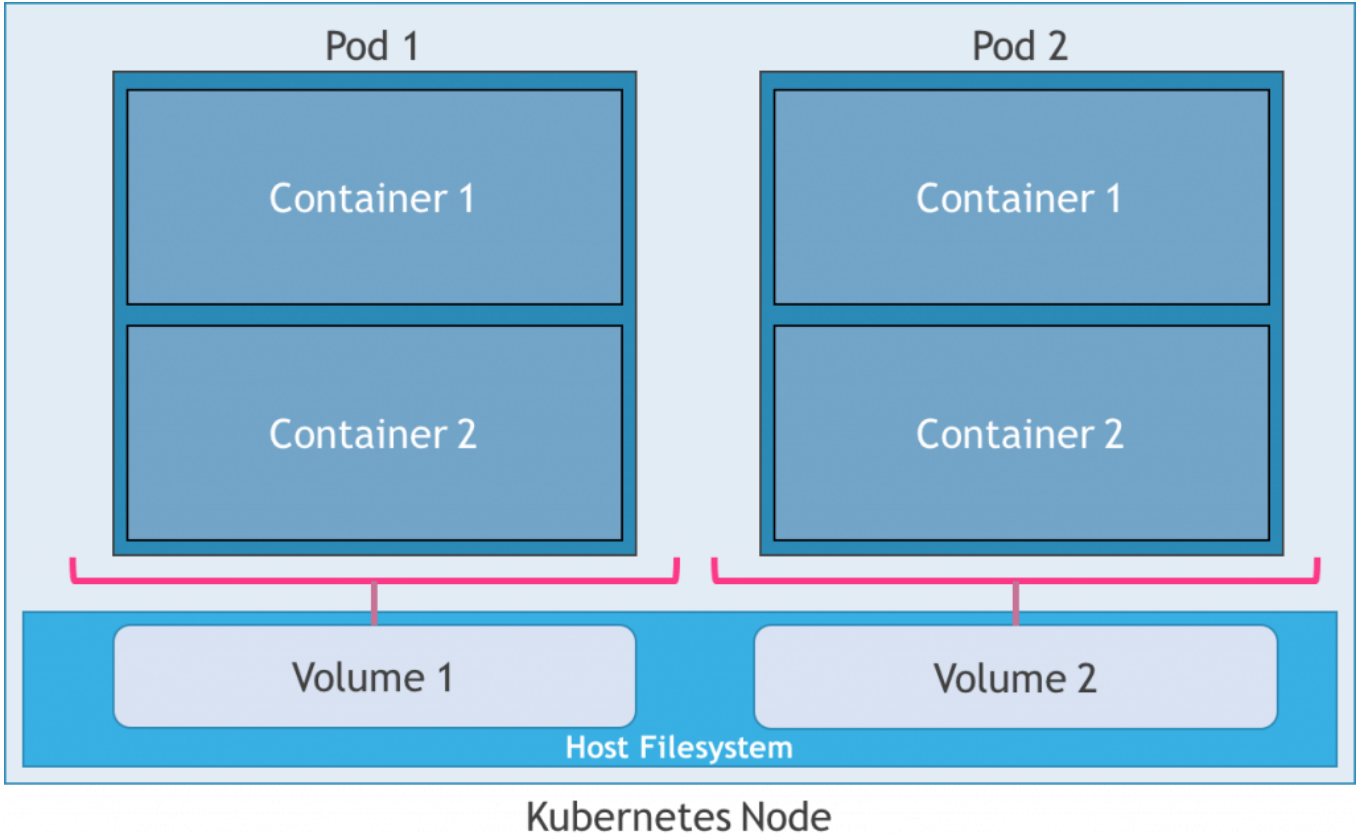
- Assignment Review & Guides
- Kubernetes volume management
- Configmap and secret

STORAGE

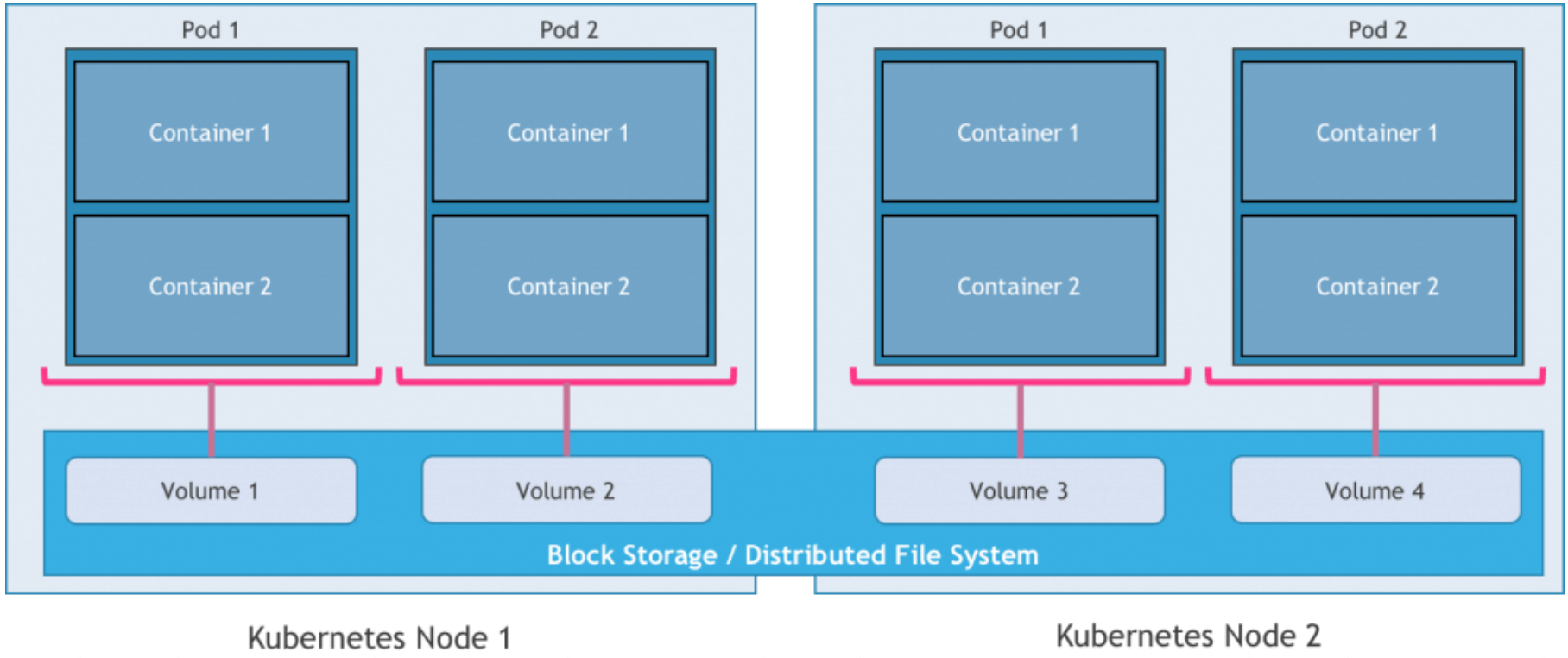
Persistent in Pods

- Pod are ephemeral and stateless
- Volumes bring persistence to Pods
- Kubernetes volumes a similar to Docker volumes, but managed differently
- All container in a Pod can access the volume
- Volumes are associated with the lifecycle of Pod
- Directory in the host are exposed as volumes
- Volumes may be based on a variety of storage backend
- A pod can have one or more types of volumes attached to it

Pod and Volumes



Pod and Volumes



Type of Volumes

- **Host-based**
  - emptyDir
  - hostPath
- **Block Storage**
  - awsElasticBlockStore
  - azureDisk
  - gcePersistentDisk
  - vsphereVolume
  - ...
- **Distributed File System**
  - NFS
  - Ceph
  - Gluster
  - Amazon EFS
  - Azure File System
  - ...
- **Other**
  - Flocker
  - iScsi

AWS EBS configuration example

If the EBS volume is partitioned, you can supply the optional field **partition: "<partition number>"** to specify which parition to mount on

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-ebs
spec:
  containers:
  - image: k8s.gcr.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-ebs
      name: test-volume
  volumes:
  - name: test-volume
    # This AWS EBS volume must already exist.
    awsElasticBlockStore:
      volumeID: "<volume id>"
      fsType: ext4
```

Azure Disk configuration example

```yaml
kind: Pod
apiVersion: v1
metadata:
  name: pod-uses-managed-ssd-5g
  labels:
    name: storage
spec:
  containers:
  - image: nginx
    name: az-c-01
    command:
    - /bin/sh
    - -c
    - while true; do echo $(date) >> /mnt/managed/outfile; sleep 1; done
    volumeMounts:
    - name: managed01
      mountPath: /mnt/managed
  volumes:
  - name: managed01
    azureDisk:
      kind: Managed
      diskName: myDisk
      diskURI: /subscriptions/<subscriptionID>/resourceGroups/<resourceGroup>/providers/Microsoft.Compute/disks/<diskName>
```

emptyDir configuration example

An emptyDir volume is first created when a Pod is assigned to a node, and exists as long as that Pod is running on that node. As the name says, the emptyDir volume is initially empty

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: k8s.gcr.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir: {}
```

hostPath configuration example

A hostPath volume mounts a file or directory from the host node's filesystem into your Pod

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: k8s.gcr.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-pd
      name: test-volume
  volumes:
  - name: test-volume
    hostPath:
      # directory location on host
      path: /data
      # this field is optional
      type: Directory
```

Persistent volumes

A PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes.

It is a resource in the cluster just like a node is a cluster resource. PVs are volume plugins like Volumes, but have a lifecycle independent of any individual Pod that uses the PV

Persistent volumes

- A **PersistentVolume** (PV) represents a storage resource.
- PVs are a **cluster wide resource** linked to a backing storage provider: NFS, EBS, GCEPersistentDisk, etc.
- Generally provisioned by an administrator.
- Their lifecycle is handled independently from a pod
- **CANNOT** be attached to a Pod directly. Relies on a **PersistentVolumeClaim** (PVC).
- A PVC is a **namespaced** request for storage.

Persistent volumes claim

A PersistentVolumeClaim (PVC) is a request for storage by a user.

It is similar to a Pod. Pods consume node resources and PVCs consume PV resources. Pods can request specific levels of resources (CPU and Memory).

Claims can request specific size and access modes (e.g., they can be mounted ReadWriteOnce, ReadOnlyMany or ReadWriteMany).

Provision persistent volume and claim

There are two ways PVs may be provisioned: statically or dynamically

- **Static**

  A cluster administrator creates a number of PVs. They carry the details of the real storage, which is available for use by cluster users. They exist in the Kubernetes API and are available for consumption.

- **Dynamic**

  When none of the static PVs the administrator created match a user's PersistentVolumeClaim, the cluster may try to dynamically provision a volume specially for the PVC. This provisioning is based on StorageClasses

Provision persistent example

Each PV contains a spec and status, which is the specification and status of the volume

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: slow
  mountOptions:
    - hard
    - nfsvers=4.1
  nfs:
    path: /tmp
    server: 172.17.0.2
```

Volume Mode

Kubernetes supports two volumeModes of PersistentVolumes: **Filesystem and Block**

A volume with volumeMode: **Filesystem** is mounted into Pods into a directory. If the volume is backed by a block device and the device is empty, Kubernetes creates a filesystem on the device before mounting it for the first time.

You can set the value of volumeMode to **Block** to use a volume as a raw block device. Such volume is presented into a Pod as a block device, without any filesystem on it. This mode is useful to provide a Pod the fastest possible way to access a volume, without any filesystem layer between the Pod and the volume

Access Mode

The access modes are:

- ReadWriteOnce -- the volume can be mounted as read-write by a single node
- ReadOnlyMany -- the volume can be mounted read-only by many nodes
- ReadWriteMany -- the volume can be mounted as read-write by many nodes
- ReadWriteOncePod -- the volume can be mounted as read-write by a single Pod. This is only supported for CSI volumes and Kubernetes version 1.22+.

Volume Snapshots

In Kubernetes, a VolumeSnapshot represents a snapshot of a volume on a storage system

Similar to how API resources PersistentVolume and PersistentVolumeClaim are used to provision volumes for users and administrators, VolumeSnapshotContent and VolumeSnapshot API resources are provided to create volume snapshots for users and administrators

Volume Snapshots configuration example

Each VolumeSnapshot contains a spec and a status

persistentVolumeClaimName is the name of the PersistentVolumeClaim data source for the snapshot. This field is required for dynamically provisioning a snapshot

```yaml
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshot
metadata:
  name: new-snapshot-test
spec:
  volumeSnapshotClassName: csi-hostpath-snapclass
  source:
    persistentVolumeClaimName: pvc-test
```

Volume Snapshots Contents configuration example

volumeHandle is the unique identifier of the volume created on the storage backend and returned by the CSI driver during the volume creation

```yaml
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotContent
metadata:
  name: snapcontent-72d9a349-aacd-42d2-a240-d775650d2455
spec:
  deletionPolicy: Delete
  driver: hostpath.csi.k8s.io
  source:
    volumeHandle: ee0cfb94-f8d4-11e9-b2d8-0242ac110002
  volumeSnapshotClassName: csi-hostpath-snapclass
  volumeSnapshotRef:
    name: new-snapshot-test
    namespace: default
    uid: 72d9a349-aacd-42d2-a240-d775650d2455
```

Storage class

- A StorageClass provides a way for administrators to describe the "classes" of storage they offer.
- Satisfies a set of requirements instead of mapping to a storage resource directly.
- Ensures that an application's claim for storage is portable across numerous backends or providers.
- Different classes might map to quality-of-service levels, or to backup policies, or to arbitrary policies determined by the cluster administrators

The StorageClass Resource

- Each StorageClass contains the fields provisioner, parameters, and reclaimPolicy, which are used when a PersistentVolume belonging to the class needs to be dynamically provisioned.

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: standard
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
reclaimPolicy: Retain
allowVolumeExpansion: true
mountOptions:
  - debug
volumeBindingMode: Immediate
```

The StorageClass Provisioner

Volume Plugin	Internal Provisioner	Config Example
AWSElasticBlockStore	✓	AWS EBS
AzureFile	✓	Azure File
AzureDisk	✓	Azure Disk
CephFS	-	-
Cinder	✓	OpenStack Cinder
FC	-	-
FlexVolume	-	-
Flocker	✓	-

The StorageClass Provisioner

GCEPersistentDisk	✓	GCE PD
Glusterfs	✓	Glusterfs
iSCSI	-	-
Quobyte	✓	Quobyte
NFS	-	NFS
RBD	✓	Ceph RBD
VsphereVolume	✓	vSphere
PortworxVolume	✓	Portworx Volume
ScaleIO	✓	ScaleIO
StorageOS	✓	StorageOS
Local	-	Local

CONFIGURATION

ConfigMaps

- An API object used to store non-confidential data in key-value pairs
- Externalized data stored within kubernetes
- Can be referenced through several different means:
  - ❑ environment variable
  - ❑ a command line argument (via environment variable)
  - ❑ injected as a configuration file into a volume mount
- Can be created from a manifest, literals, directories, or files directly.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: manifest-example
data:
  state: Minnesota
  city: Minneapolis
  content: |
    Look at this,
    its multiline!
```

Secrets

- Functionally identical to a ConfigMap.
- Stored as base64 encoded content.
- Encrypted at rest within etcd (if configured!).
- Ideal for username/passwords, certificates or other sensitive information that should not be stored in a container.
- Can be created from a manifest, literals, directories, or from files directly.

- **type**: There are three different types of secrets within Kubernetes:
  - **docker-registry** - credentials used to authenticate to a container registry
  - **generic/Opaque** - literal values from different sources
  - **tls** - a certificate based secret
- **data**: Contains key-value pairs of base64 encoded content.

```
apiVersion: v1
kind: Secret
metadata:
  name: manifest-secret
type: Opaque
data:
  username: ZXhhbXBsZQ==
  password: bXlwYXNzd29yZA==
```

LAB

Q&A

Thank you