

Docker for Developers



DevOps Course

Written by: Yaniv Cohen





Clone Git

`git clone https://github.com/yanivomc/seminars.git`

Questions for you...

- What Do You Know About Docker?
- What Do You Know About K8S?
- Who Used Docker/K8S For Development / QA / STG / PROD?
- Who Tried & Failed Implementing Docker / K8S
 - Tell me about the project



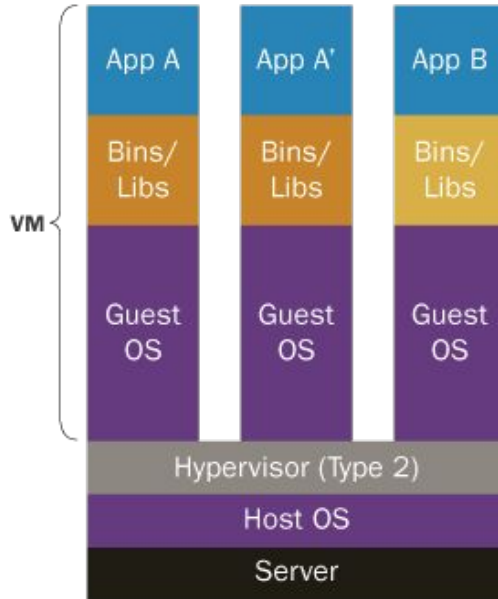
INTRO - DOCKER

Why do we need to know Docker?

What is Docker?

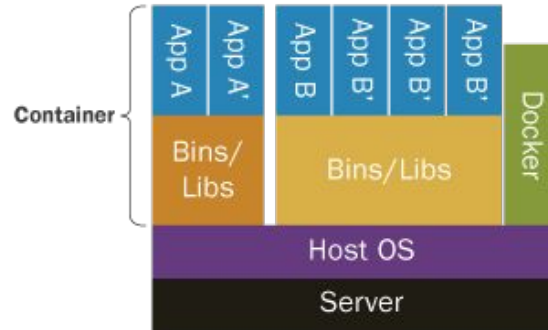
- Docker is an open platform for developing, shipping, and running applications.
- Docker allows you to package an application with all of its dependencies into a standardized unit for software development.

Containers VS. VMs



Virtual Machines

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



Containers

Docker Benefits Upon VMs

- Small to tiny images - Few hundred MB's for OS + Application (5MB for full OS - [Alpine](#)) VS. Gigabytes in VM's
- Very small footprint on the host machine (CPU, RAM Impact) as Docker only use what it required instead of building a complete Operating system per VM.
- Containers use up only as many system resources as they need at a given time. VMs usually require some resources to be permanently allocated before the virtual machine starts.
- Direct hardware access. Applications running inside virtual machines generally cannot access hardware like graphics cards on the host in order to speed processing. Containers Can (ex. [Nvidia](#))
- Microservice in nature and integrations (API's) for whatever task required.
- Portable, Fast (Deployments , Migration , Restarts and Rollbacks) and Secure
- Can run anywhere and everywhere
- Simplify DevOps
- Version controlled
- Open Source

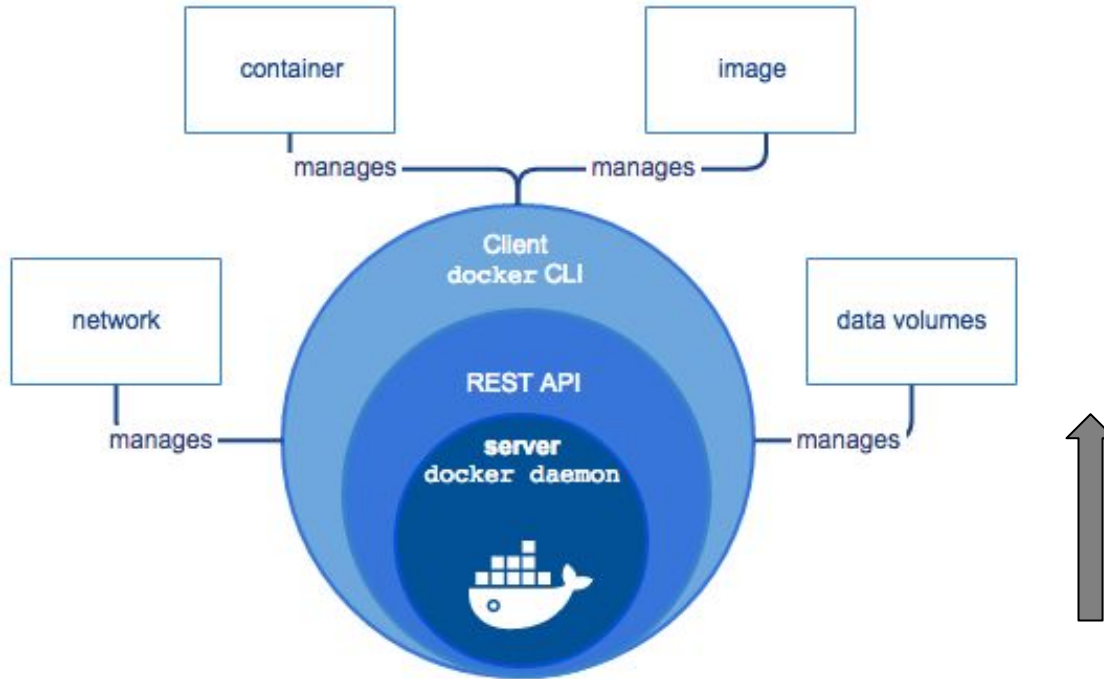
Common Use Cases for Docker

- CI / CD
- Fast Scaling application layers for overcoming application performance limitations.
- For Sandboxed environments (Development, Testing , Debugging)
- Local development environment (no more “ It ran on my laptop...”)
- Infrastructure as a CODE made easy with docker
- Multi-Tier applications (Front End , Mid Tier (Biz Logic) , Data Tier) / Microservices
- Building PaaS , SaaS

- Architecture: Linux X86-64
- **Written in: GoLang** (On March 13, 2014, with the release of version 0.9, Docker dropped LXC as the default execution environment which is an operating system level virtualization and replaced it with its own libcontainer library written in the Go programming language)
- Engine: Client - Server (Daemon) Architecture
- Namespace: Isolation of process in linux where one process cant "See" the other process
- Control Groups: Linux Kernel capability to limit and isolate the resource usage (CPU, RAM, disk I/O, network etc..) of a collection of process
- Container format: libcontainer - Go implementation for creating containers with namespaces, control groups and File system capabilities access control

Docker Architecture

Overview



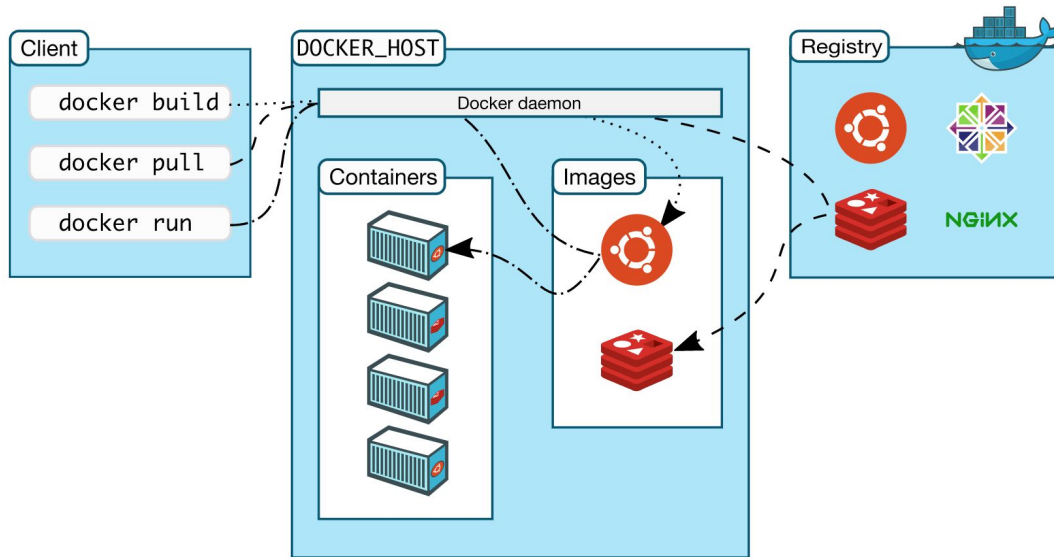
What is Docker - Technical Aspect

Docker Architecture

Docker uses a client-server architecture. The Docker *client* talks to the Docker *daemon*, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon *can* run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface.

Docker Architecture

Docker Architecture



Docker Components

- Engine
- Daemon
- (Docker) Client
- Docker Registries
- Docker Objects
- Machine
- Compose
- Swarm

Docker Components

Engine

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

Docker Components

Daemon

- The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

Docker Components

Docker Client

- The Docker client (docker) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to dockerd, which carries them out. The docker command uses the Docker API. The **Docker client can communicate with more than one daemon.**

Docker Registries

- A Docker registry stores Docker images. Docker Hub and Docker Cloud are public registries that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry.
- When one use “docker pull / push / run” commands, the required images are pulled from the configured registry.

Docker Objects

- Images
 - a. Read Only template with instruction for creating a Docker Container. Often, an Image is based on another image with some additional customization.
 - b. Self own images that are fully created by you using DockerFile with a simple syntax where every instruction control a different Layer in the image. Once a change is made to a specific layer, a rebuild of the image will change only the updated layers. This what makes images small, fast and lightweight in compared to other virtualization solutions

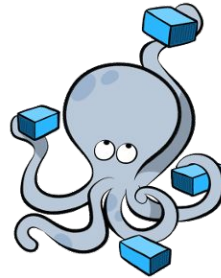
Docker Objects

- Containers
 - a. A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.
 - b. Container is defined by its image as well as any configuration options we provide to it when created or when we start it

Docker Objects

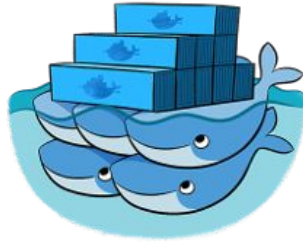
- Services
 - a. Allow you to scale containers across multiple Docker daemons, which all work together as a **swarm** with multiple managers and workers. Each member of a swarm is a Docker daemon, and the daemons all communicate using the Docker API. A service allows you to define the desired state, such as the number of replicas of the service that must be available at any given time. By default, the service is load-balanced across all worker nodes. To the consumer, the Docker service appears to be a single application. Docker Engine supports swarm mode in Docker 1.12 and higher.

Docker Compose



A tool for defining and running complex applications with
Docker (eg multi-container application ex. LAMP)
With a single file

Docker Swarm



A Native Clustering tool for Docker. Swarm pools together several Docker hosts and exposed them as a single virtual Docker host. It scale up to multiple hosts

Docker Components

Good to know:

Docker Machine



A Tool which makes it easy to create Docker Hosts on Operating systems that does not support docker natively, or on cloud providers and inside your datacenter.



INSTALLING DOCKER

Windows 10 Enterprise / Educational



[DOWNLOAD HERE](https://docs.docker.com/docker-for-windows/)

<https://docs.docker.com/docker-for-windows/>

Windows 10 Enterprise / Educational

1. Turn windows features on or off
 - a. Enable HYPER V
 - b. Restart

Windows 10 Check Functionality

1. Open a shell (`cmd.exe` , PowerShell, or other).
2. Run some Docker commands, such as `docker ps` , `docker version` , and `docker info` .

Here is the output of `docker ps` run in a powershell. (In this example, no containers are running yet.)

```
PS C:\Users\jdoe> docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
--------------	-------	---------	---------	--------	-------

Here is an example of command output for `docker version` .

```
PS C:\Users\Docker> docker version
Client:
Version:      17.03.0-ce
API version:  1.26
Go version:   go1.7.5
Git commit:   60ccb22
Built:        Thu Feb 23 10:40:59 2017
OS/Arch:      windows/amd64

Server:
Version:      17.03.0-ce
API version:  1.26 (minimum version 1.12)
Go version:   go1.7.5
Git commit:   3a232c8
Built:        Tue Feb 28 07:52:04 2017
OS/Arch:      linux/amd64
Experimental: true
```



Let's Start

Docker Basics

```
docker run -i -t -d --name dockerlearning -p 8080:80 alpine:latest ash
```

- 'docker run' will run the container
- This will not restart an already running container, just create a new one
- docker run [options] IMAGE [command] [arguments]
 - a. [options] modify the docker process for this container
 - b. IMAGE is the image to use
 - c. [command] is the command to run inside the container (entry point to hold the container running)
 - d. [arguments] are arguments for the command

```
docker run -i -t -d --name dockerlearning -p 8080:80 alpine:latest ash
```

- 'docker run' will run the container
 - a. -i - Interactive mode
 - b. -t - Allocate pseudo TTY - or not Terminal will be available
 - c. -d - Run in the background (Daemon style)
 - d. --name - Give the container a name or let Docker to name it
 - e. -p [local port] : [container port] - Forward local port to the container port

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
98debfd4458	alpine:latest	"sh"	Less than a second ago	Up 1 second	0.0.0.0:8080->80/tcp	dockerlearning

```
docker run -i -t -d --name dockerlearning -p 8080:80 alpine:latest sh
```

- Pulls the alpine:latest image from the registry (if not existed on our station)
 - a. Run “docker images” to see what images already downloaded / in use locally
- Creates new container
- Allocate FS and Mounts a read-write Layer
- Allocates network/bridge interface
- Set up an IP Address
- Executes a process that we specify (in this scenario - “sh” as alpine release doesn't have bash)
- Captures and provides application outputs

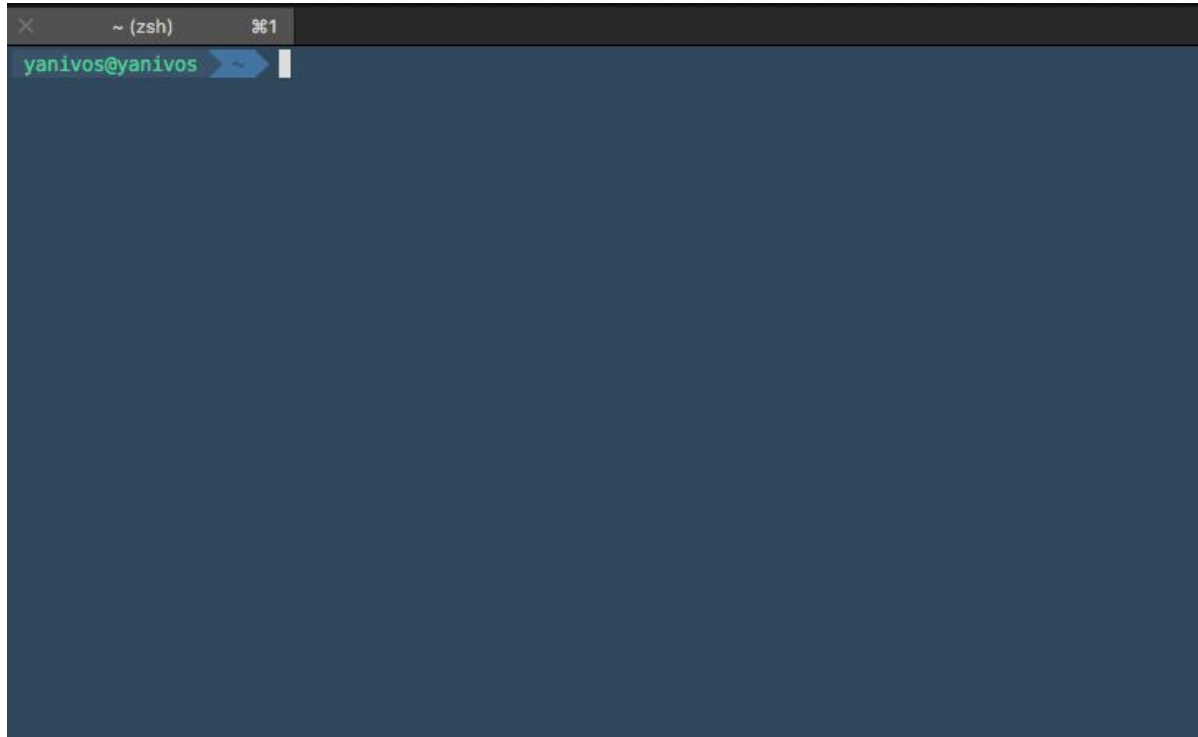
Docker Examples

- Pull / Run an image
- SSH into a container
- View Logs
- Docker Volume
- Using Dockerfile - Building our own Jar
- Package an app and push it to a repo

Common Docker Commands

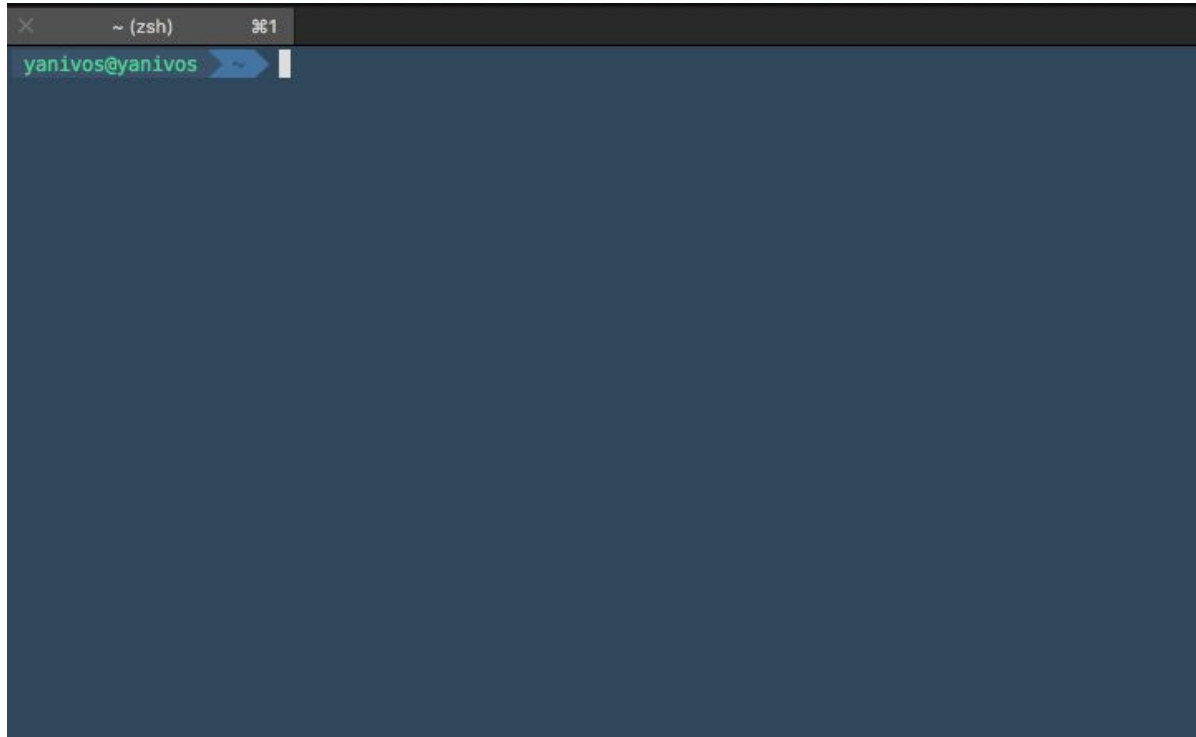
```
// General info
man docker // man docker-run
docker help // docker help run
docker info
docker version
docker network ls
// Images
docker images // docker [IMAGE_NAME]
docker pull [IMAGE] // docker push [IMAGE]
// Containers
docker run
docker ps // docker ps -a, docker ps -l
docker stop/start/restart [CONTAINER]
docker stats [CONTAINER]
docker top [CONTAINER]
docker port [CONTAINER]
docker inspect [CONTAINER]
docker inspect -f "{{ .State.StartedAt }}" [CONTAINER]
docker rm [CONTAINER]
```

Running simple shell



A terminal window with a dark blue background. The title bar at the top shows a close button, the text "~ (zsh)", and a window icon. The prompt "yanivos@yanivos" is displayed in green text, followed by a blue arrow icon and a white cursor. The rest of the terminal area is empty.

Building & Running Mysql On docker



A terminal window with a dark blue background. The title bar shows a close button, a window icon, and the text '~ (zsh) 961'. The prompt 'yanivos@yanivos' is displayed in green, followed by a blue arrow icon and a white cursor.

```
~ (zsh) 961  
yanivos@yanivos ~
```

Why not to run SSH inside a container

- We can...
- Docker is designed for one command per container - Now we run two
- If any update or modification is needed, We need to change our setup and not the docker image..
- If you still want to review something... SSH it.



Docker Advanced

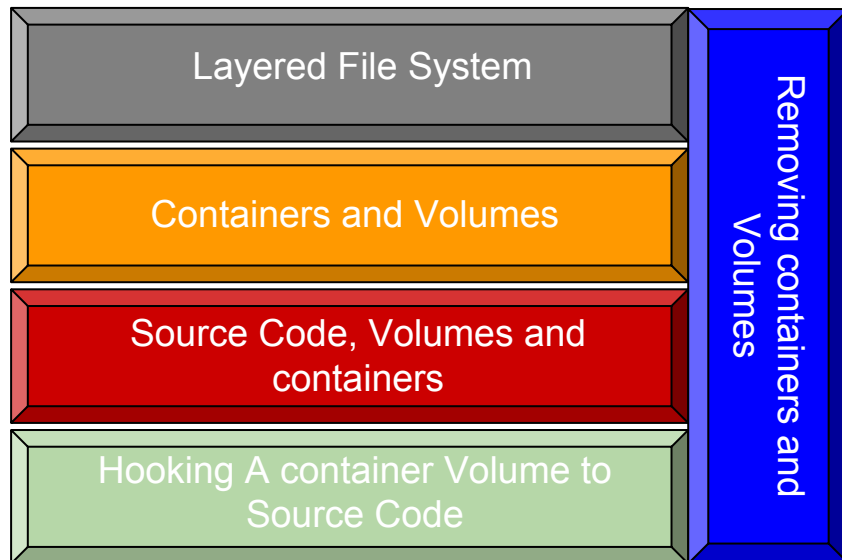
- Volumes - Hooking Source code into a container
- Networking and communications
- Building Custom Images with DockerFile
- Building Custom images with Docker Compose (v3 YAML)
- Working with images
- Building a Microservice Project
- Working with Private Registries



HOOKING SOURCE CODE

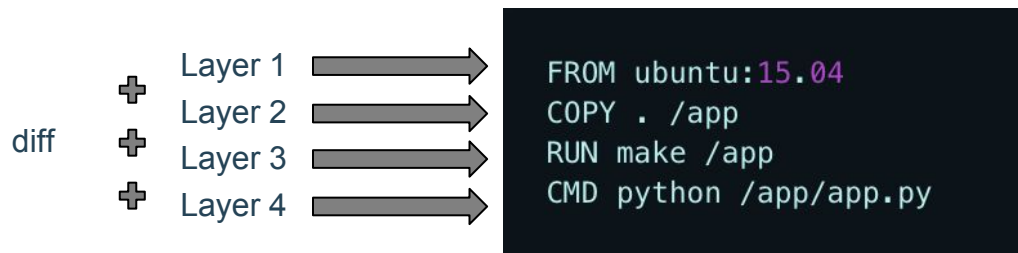
Module Agenda

To understand how we can hook our source code into a container,
We will go over the following:



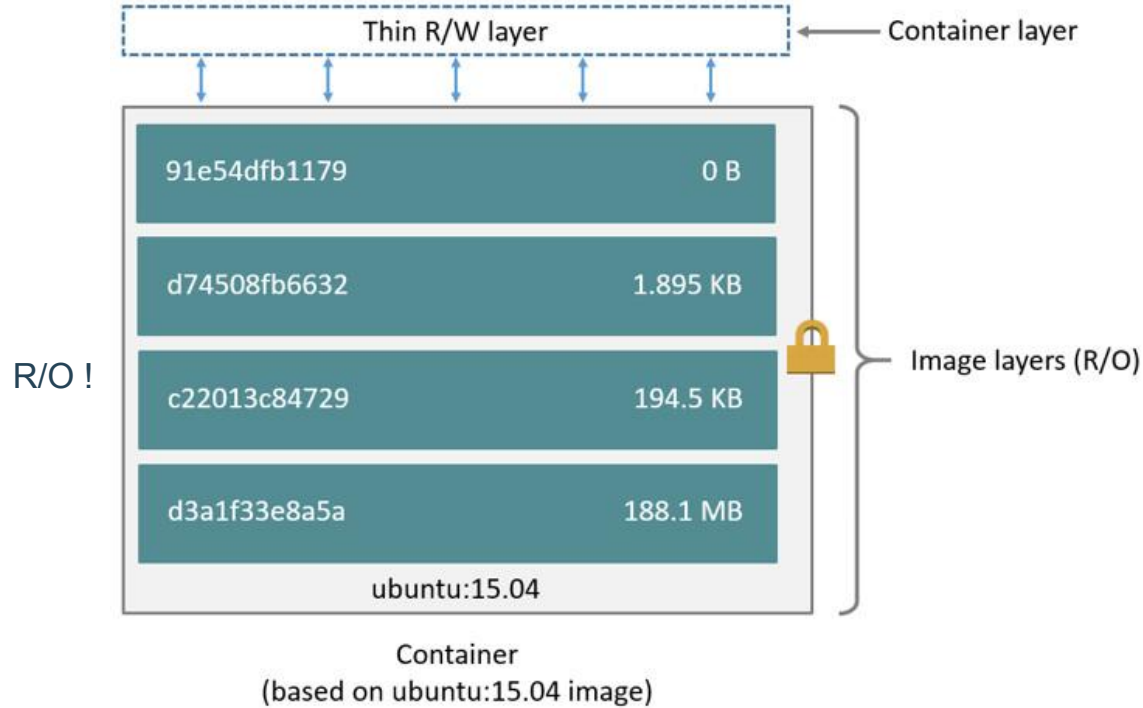
- Images and Layers

A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile



Each layer is only a set of differences from the layer before it. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the “**container layer**”. All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer.

Layered FS

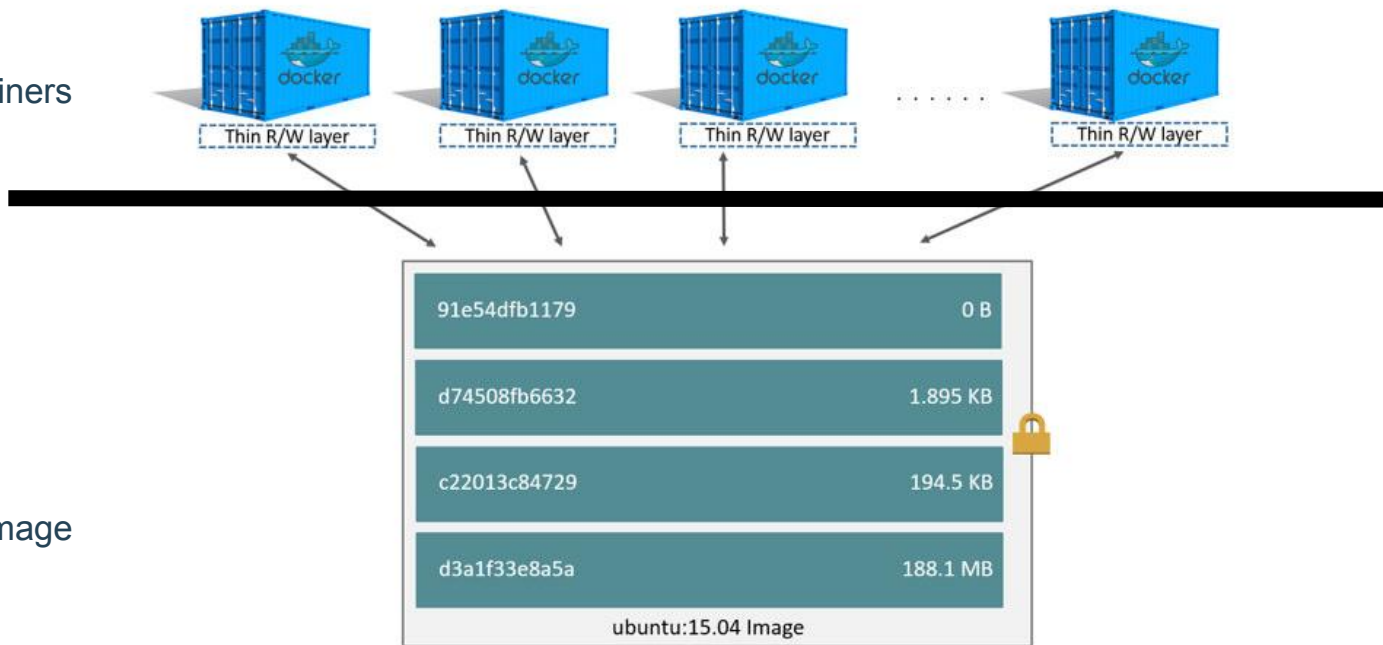


- **Containers and Layers**

The major difference between a container and an image is the top writable layer. All writes to the container that add new or modify existing data are stored in this writable layer. When the container is deleted, the writable layer is also deleted. The underlying image remains unchanged. Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image.

Layered FS

Different Containers



Using same Image

Note: If we need multiple images to have shared access to the exact same data, we store this data in a Docker **volume** and mount it into your containers.



SO HOW DO WE GET OUR SOURCE CODE INTO A CONTAINER?



Containers and Volumes

- What is a Volume

Special type of directory in a container typically referred to as a “data volume”

- Can be shared and reused among one or many containers
- Updates to an image won't affect a data volume
- Data volumes are persisted even after container deletion
- Volumes are OS agnostic. They can run on Linux and windows containers
- Volumes drivers allow us to store volumes on remote hosts or cloud providers.
- Volumes can be encrypted or to add other functionality
- A new volume content can be pre-populated by a container



Containers and Volumes

Follow through

- **Create and manage volumes:**

What will we achieve in the following follow through session:

- Creating new volume
 - Inspecting
 - Removing
- Start a container[s] with a volume

Follow through

RUN

```
docker run -dti --name alpine1 --mount target=/app alpine ash
```

INSPECT

```
docker inspect alpine1
```

```
},  
"Mounts": [  
  {  
    "Type": "volume",  
    "Name": "e2c79e12b3f8b90888688da603cca4c2297ee96f2b914a601378e5d944f17214",  
    "Source": "/var/lib/docker/volumes/e2c79e12b3f8b90888688da603cca4c2297ee96f2b914a601378e5d944f17214/_data",  
    "Destination": "/app",  
    "Driver": "local",  
    "Mode": "z",  
    "RW": true,  
    "Propagation": ""  
  }  
],
```

STOP AND DELETE CONTAINER

```
docker stop alpine1 && docker rm alpine1
```

Follow through 2

Creating a VOLUME managed by docker FS and share it with multiple containers

RUN

```
docker volume create fs_shared
```

LIST VOLUMES

```
docker volume ls
```

```
local          fs_shared
```

RUN AND MOUNT

```
docker run --rm -tdi --name alpine1 --mount source=fs_shared,target=/app alpine ash
```

```
docker run --rm -tdi --name alpine2 --mount source=fs_shared,target=/app alpine ash
```

```
docker run --rm -tdi --name alpine3 --mount source=fs_shared,target=/app alpine ash
```

LAB

Attach to running containers, create files and verify files gets updated on all containers

Disconnect sequence `Ctrl + p + Ctrl + q`



Containers and Volumes

BIND MOUNTS

- Bind Mounts

Bind mounts have been around since the early days of Docker. Bind mounts have limited functionality compared to volumes. When you use a bind mount, a file or directory on the host machine is mounted into a container. The file or directory is referenced by its full or relative path on the host machine. By contrast, when you use a volume, a new directory is created within Docker's storage directory on the host machine, and Docker manages that directory's contents. **TODO: ADD WINDOWS MOUNT**

COMMAND EXAMPLE

-v

```
docker run --rm -tdi -v "$(pwd)"/source:/app [image] [CMD]
```

--mount

```
docker run --rm -tdi --mount type=bind,source="$(pwd)"/source,target=/app [image] [CMD]
```

- **BIND MOUNTS USING -V OR --MOUNT ?**

- Both will provide the same outcome but as -v /--volume exists since day 1 in docker and --mount was introduced since docker 17.06 it became normal and easier to use --mount.



Containers and Volumes

LAB: BIND MOUNTS

- **Create and manage bind mount:**

- Create new host local project folder called “jb_docker” and cd into it
 - Create 2 alpine nodes and share new local folder called source1 using --mount
 - Create 2 alpine nodes and share new local folder called source2 using -v
 - What happened when you tried creating a shared host folder with --mount without first creating the folder manually ? and what happened when you were using -v
- Inspect the new volumes and containers
- Validate shared folder by creating files and make sure the exists on both containers
- Stop all docker containers and Make sure containers got deleted



Containers and Volumes

LAB: Running BootStrap app in a container

- **Hook SpringBoot Jar into a container:**

- Cd into your “jb_docker” folder
 - Copy from your cloned git the demo artifact to ./source
seminars/docker/artifacts/**spring-music.jar**
- Run 1 new container
 - Name: web_api
 - Mount Using -v or --mount
 - source: ./source
 - Target: /app
 - Image: **frolvlad/alpine-oraclejdk8:slim**
 - CMD: **java -jar -Dspring.profiles.active /app/spring-music.jar**

- Validate your work:
 - Run docker ps and expect to see the following

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
c9ca53789a18	frolvlad/alpine-oraclejdk8:slim	"java -jar -Dsprin..."	About a minute ago	Up 2 minutes	0.0.0.0:8080->8080/tcp, 0.0.0.0:8091->8091/tcp	web_api

- Run docker logs OR attach and expect seeing the following (remember ctrl+p+ctrl+q to disconnect)

```
2018-03-01 22:11:36.151 INFO 1 --- [main] s.w.s.m.a.RequestMappingHandlerMapping : Mapped "{[/error]}" onto public java.util.Map<java.lang.String, java.lang.Object> org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter.invoke()
2018-03-01 22:11:36.164 INFO 1 --- [main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/webjars/**] onto handler of type [class org.springframework.web.servlet.resource.ResourceHandlerMapping]
2018-03-01 22:11:36.164 INFO 1 --- [main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**] onto handler of type [class org.springframework.web.servlet.resource.ResourceHandlerMapping]
2018-03-01 22:11:36.199 INFO 1 --- [main] s.w.s.m.a.RequestMappingHandlerAdapter : Looking for @ControllerAdvice: org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext@4aa8f0b4
Mar 01 22:11:34 GMT 2018]; parent: org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext@4aa8f0b4
2018-03-01 22:11:36.645 INFO 1 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8091 (http)
2018-03-01 22:11:36.683 INFO 1 --- [main] o.s.c.support.DefaultLifecycleProcessor : Starting beans in phase 2147483647
2018-03-01 22:11:36.685 INFO 1 --- [main] d.s.w.p.DocumentationPluginsBootstrapper : Context refreshed
2018-03-01 22:11:37.026 INFO 1 --- [main] d.s.w.p.DocumentationPluginsBootstrapper : Found 1 custom documentation plugin(s)
2018-03-01 22:11:37.140 INFO 1 --- [main] s.d.s.w.s.ApiListingReferenceScanner : Scanning for api listing references
2018-03-01 22:11:38.954 INFO 1 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8090 (http)
2018-03-01 22:11:38.985 INFO 1 --- [main] com.khoubyari.example.Application : Started Application in 61.65 seconds (JVM running for 64.564)
2018-03-01 22:11:39.501 INFO 1 --- [nio-8091-exec-1] o.a.c.c.C.[Tomcat-1].[localhost].[/] : Initializing Spring FrameworkServlet 'dispatcherServlet'
2018-03-01 22:11:39.502 INFO 1 --- [nio-8091-exec-1] o.s.web.servlet.DispatcherServlet : FrameworkServlet 'dispatcherServlet': initialization started
2018-03-01 22:11:40.484 INFO 1 --- [nio-8091-exec-1] o.s.web.servlet.DispatcherServlet : FrameworkServlet 'dispatcherServlet': initialization completed in 982 ms
```

Try Browsing from your host browser: <http://localhost:8080/>

Did it worked?

- What do you need to do to forward request to port 8080 and 8080 to your docker web_api ?



FINAL SOLUTION

HOOKING YOUR OWN SOURCE CODE

Volumes

```
docker run -tdi --name web_api -v "$(pwd)"/source:/app -p 8080:8080 frolovlad/alpine-oraclejdk8:slim java -jar  
-Dspring.profiles.active /app/spring-music.jar
```



STOP AND REMOVE

CLEAN UP



Docker Advanced

Dockerfile - Custom images

Module Agenda

“Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. Using docker build users can create an automated build that executes several command-line instructions in succession “

Getting started with Dockerfile

Creating a Custom Dockerfile

Building a Custom image

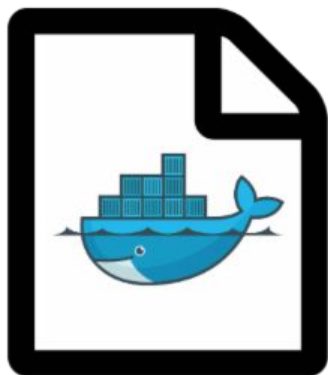
Publishing an Image to Docker
Hub

What will we do in this module?

Get our source code into a **custom built image (vs pre-built images)** to share with others

What is a dockerfile and how it create an Image

Developers use .java or pom file to describe / develop - we use Dockerfile



Dockerfile

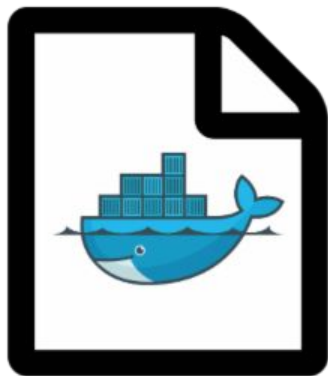


pom.xml



Hello.java

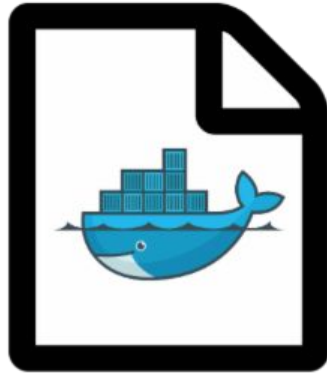
Dockerfile is a ... FILE with instructions and descriptions of an image



Dockerfile

BUILD	Boot	Run
FROM	WORKDIR	CMD
	USER	ENV
COPY		EXPOSE
ADD		VOLUME
RUN		ENTRYPOINT
ONBUILD		
.dockerignore		

Dockerfile flow



dockerfile



Build - done by Docker
daemon and not CLI



image

Dockerfile flow Overview

Text file with instructions

Build process sends entire context
to daemon recursively

```
$ docker build -t [repository:tag] .  
Sending build context to Docker daemon 6.51 MB  
...
```

Docker IMAGE created

Warning: Do not use your root directory, */*, as the PATH as it causes the build to transfer the entire contents of your hard drive to the Docker daemon.



Dockerfile - Example

```
ARG VERSION=latest
# Java8 Alpine Release
FROM frolvlad/alpine-oraclejdk8:slim
ARG VERSION
# RUN will execute shell commands
RUN echo $VERSION > image_version
# Label Use Labels for descriptions and view it with docker inspect
LABEL multi.label1="value1" \
    description="Bug fix x.0 for client y"
# configure WorkDir inside the container
WORKDIR /app
# Mount HOST Folder
VOLUME ["/spring-boot-rest-example/dockerfile/artifact/"]
# Copy Spring Boot File to target
COPY spring-boot-rest-example-0.4.0.war /app/spring-boot-rest-example-0.4.0.war
#Expose Ports - ONLY EXPOSED - IT'S NOT Mapped. -p will be needed on run
EXPOSE 8091
EXPOSE 8090
#The HEALTHCHECK instruction tells Docker how to test a container to check that it is still working
HEALTHCHECK --interval=5m --timeout=3s \ CMD curl -f http://localhost/ || exit 1
# The main purpose of a CMD is to provide defaults for an executing container
CMD java -jar /app/spring-boot-rest-example-0.4.0.war -Dspring.profiles.active=test
```


CMD VS ENTRYPOINT?

```
FROM alpine:latest  
CMD ping localhost
```

```
docker build -t playground:latest .  
....  
docker run -ti playground:latest  
PING localhost (127.0.0.1): 56 data bytes  
64 bytes from 127.0.0.1: seq=0 ttl=64 time=0.051 ms  
64 bytes from 127.0.0.1: seq=1 ttl=64 time=0.080 ms  
# in CMD - Override IS ALLOWED  
#### docker run -ti playground:latest [command]  
docker run -ti playground:latest hostname  
93d4a120e1ff
```

CREATING A CUSTOM BOOTSPRING DOCKERFILE

MAKE SURE YOU CLONED

<https://github.com/yanivomc/seminars.git>

Dockerfile - LAB

1. Make a new folder in your project directory called `jb_dockerfile`
 - a. Copy `spring-music.jar` from `/seminars/docker/artifacts` to a new folder `jb_dockerfile/artifacts`
 - b. Create an empty Dockerfile
2. SPEC
 - a. From: `frolvlad/alpine-oraclejdk8:slim`
 - b. Workdir `/data`
 - c. Copy: artifact to `/app`
 - d. Expose: `8080`
 - e. CMD: `java -jar -Dspring.profiles.active=none /app/spring-music.jar`
3. Build && Run image

Building the dockerfile in CLI

```
# Build dockerfile
# docker build -t [repo/imagename:tag] [dockerfile location]

# Run image created above
docker run -p [port_source:port_target] --rm -ti --name [container name] [image]:tag
```

Browse

<http://localhost:8080/>

DOCKER HUB

Push our docker image to docker hub

1. Create new Repo in docker hub
2. Register your newly created repo and login to it in CLI
“docker login”
3. Push your created image to your repo
“docker push repo/image:tag”



CONTAINERS ADVANCE

Limit a container's resources

- By default, a container has no resource constraints and can use as much of a given resource as the host's kernel scheduler allows. Docker provides ways to control how much memory, CPU, or block IO a container can use, setting runtime configuration flags of the docker run command.

Verify support by running:

```
yanivos@ip-10-0-0-6 ~ docker info
## If some functionality is not supported - A warning at the end will appear such as:

WARNING: No swap limit support
```

Understand the risks of running out of memory

It is important not to allow a running container to consume too much of the host machine's memory. On Linux hosts, if the kernel detects that there is not enough memory to perform important system functions, it throws an OOME, or Out Of Memory Exception, and starts killing processes to free up memory. Any process is subject to killing, including Docker and other important applications. This can effectively bring the entire system down if the wrong process is killed.

Mitigate the risk of system instability

- Perform tests to understand the memory requirements of your application before placing it into production.
- Ensure that your application runs only on hosts with adequate resources.
- Limit the amount of memory your container can use, as described below.
- Be mindful when configuring swap on your Docker hosts. Swap is slower and less performant than memory but can provide a buffer against running out of system memory.
- Consider converting your container to a [service](#), and using service-level constraints and node labels to ensure that the application runs only on hosts with enough memory

Option	Description
<code>-m</code> or <code>--memory=</code>	The maximum amount of memory the container can use. If you set this option, the minimum allowed value is <code>4m</code> (4 megabyte).
<code>--memory-swap *</code>	The amount of memory this container is allowed to swap to disk. See <code>--memory-swap</code> details.
<code>--memory-swappiness</code>	By default, the host kernel can swap out a percentage of anonymous pages used by a container. You can set <code>--memory-swappiness</code> to a value between 0 and 100, to tune this percentage. See <code>--memory-swappiness</code> details.
<code>--memory-reservation</code>	Allows you to specify a soft limit smaller than <code>--memory</code> which is activated when Docker detects contention or low memory on the host machine. If you use <code>--memory-reservation</code> , it must be set lower than <code>--memory</code> for it to take precedence. Because it is a soft limit, it does not guarantee that the container doesn't exceed the limit.
<code>--kernel-memory</code>	The maximum amount of kernel memory the container can use. The minimum allowed value is <code>4m</code> . Because kernel memory cannot be swapped out, a container which is starved of kernel memory may block host machine resources, which can have side effects on the host machine and on other containers. See <code>--kernel-memory</code> details.
<code>--oom-kill-disable</code>	By default, if an out-of-memory (OOM) error occurs, the kernel kills processes in a container. To change this behavior, use the <code>--oom-kill-disable</code> option. Only disable the OOM killer on containers where you have also set the <code>-m/--memory</code> option. If the <code>-m</code> flag is not set, the host can run out of memory and the kernel may need to kill the host system's processes to free memory.

CPU

By default, each container's access to the host machine's CPU cycles is unlimited. You can set various constraints to limit a given container's access to the host machine's CPU cycles. Most users use and configure the default CFS scheduler. In Docker 1.13 and higher, you can also configure the realtime scheduler.

CFS scheduler:

The CFS is the Linux kernel CPU scheduler for normal Linux processes. Several runtime flags allow you to configure the amount of access to CPU resources your container has (Containers uses cgroup)

Option	Description
<code>--cpus=<value></code>	Specify how much of the available CPU resources a container can use. For instance, if the host machine has two CPUs and you set <code>--cpus="1.5"</code> , the container is guaranteed at most one and a half of the CPUs. This is the equivalent of setting <code>--cpu-period="100000"</code> and <code>--cpu-quota="150000"</code> . Available in Docker 1.13 and higher.
<code>--cpu-period=<value></code>	Specify the CPU CFS scheduler period, which is used alongside <code>--cpu-quota</code> . Defaults to 100 micro-seconds. Most users do not change this from the default. If you use Docker 1.13 or higher, use <code>--cpus</code> instead.
<code>--cpu-quota=<value></code>	Impose a CPU CFS quota on the container. The number of microseconds per <code>--cpu-period</code> that the container is guaranteed CPU access. In other words, $\text{cpu-quota} / \text{cpu-period}$. If you use Docker 1.13 or higher, use <code>--cpus</code> instead.
<code>--cpuset-cpus</code>	Limit the specific CPUs or cores a container can use. A comma-separated list or hyphen-separated range of CPUs a container can use, if you have more than one CPU. The first CPU is numbered 0. A valid value might be <code>0-3</code> (to use the first, second, third, and fourth CPU) or <code>1,3</code> (to use the second and fourth CPU).
<code>--cpu-shares</code>	Set this flag to a value greater or less than the default of 1024 to increase or reduce the container's weight, and give it access to a greater or lesser proportion of the host machine's CPU cycles. This is only enforced when CPU cycles are constrained. When plenty of CPU cycles are available, all containers use as much CPU as they need. In that way, this is a soft limit. <code>--cpu-shares</code> does not prevent containers from being scheduled in swarm mode. It prioritizes container CPU resources for the available CPU cycles. It does not guarantee or reserve any specific CPU access.

CONTAINERS ADVANCE: CPU

Set container to use 50% of our CPU every second

```
yanivos@ip-10-0-0-6 ~ docker run -it --cpus=".5" ubuntu /bin/bash
```




CONTAINERS IN PRODUCTION

CONS / PROS AND INBETWEEN

And No...deploying your app inside a container - does not change it's monolith architecture to microservices

CONTAINERS IN PRODUCTION

Containers

Containers are amazing piece of technology but like anything and everything else,
There are no such thing as a free lunch.

There are many benefits that we learned about using containers and how it can make our development /
deployment / CI / CD easy and fast but a question should be asked...

What's the catch ?

PROS 101

- Containers makes our applications "virtually look" the same on most infrastructures (Physical/Cloud/VM's)
- Containers makes our application runtime dependencies the developers's responsibility - **splendid!**
- Containers require the application developers to consider application state and persistence.
- Containers, once built, provide a (mostly) consistent behavior between dev, staging, and production environments. Immutable delivery mechanism out of the box
- Blazing fast scaling our ecosystem
- Delivery time - Days and hours becomes minutes / seconds
- Handoff - Developers <> Operators - Wall of confusion ? **breached!**

The above makes using Containers a no brainer for production - But there are flaws we should know about

CONS 101

- Containers do not make your applications more secure.
- Containers do not make your applications more scalable - **This is a common misconception**
- Containers do not make your applications more portable - **Shared / Common libraries in your code?**
- Network - NAT managed by Dockerd is not how we want to work in production

All of the above becomes absolute once we move to K8S and Swarm (new Pros & cons but different...)

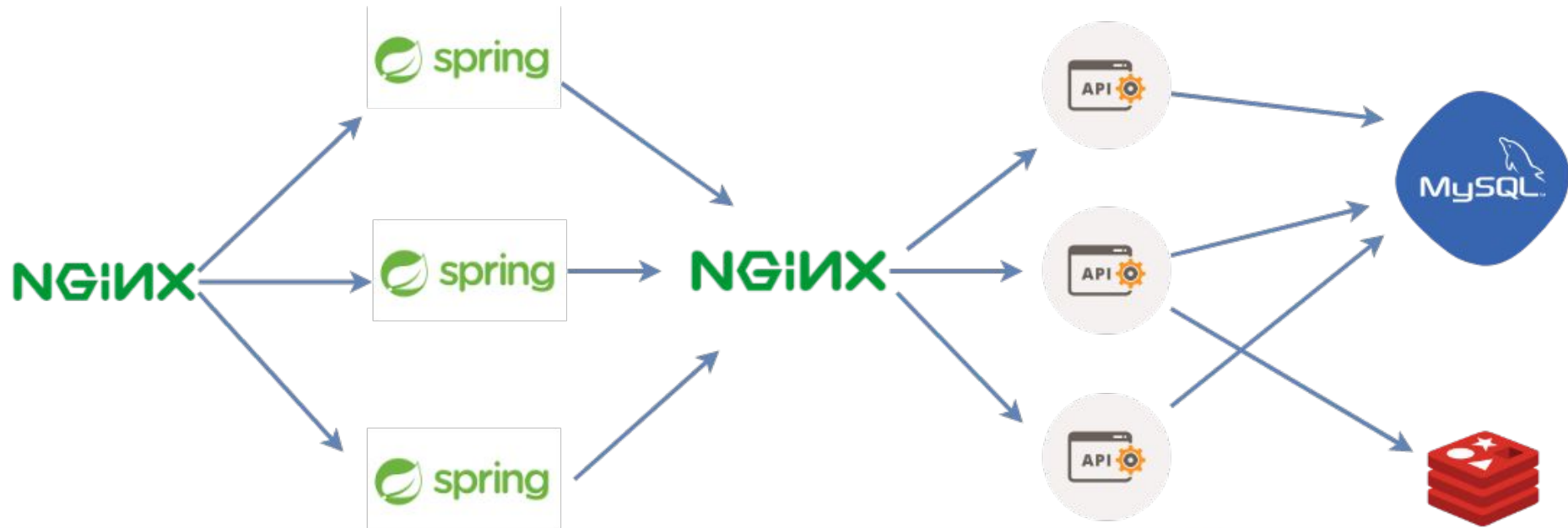


MICROSERVICES



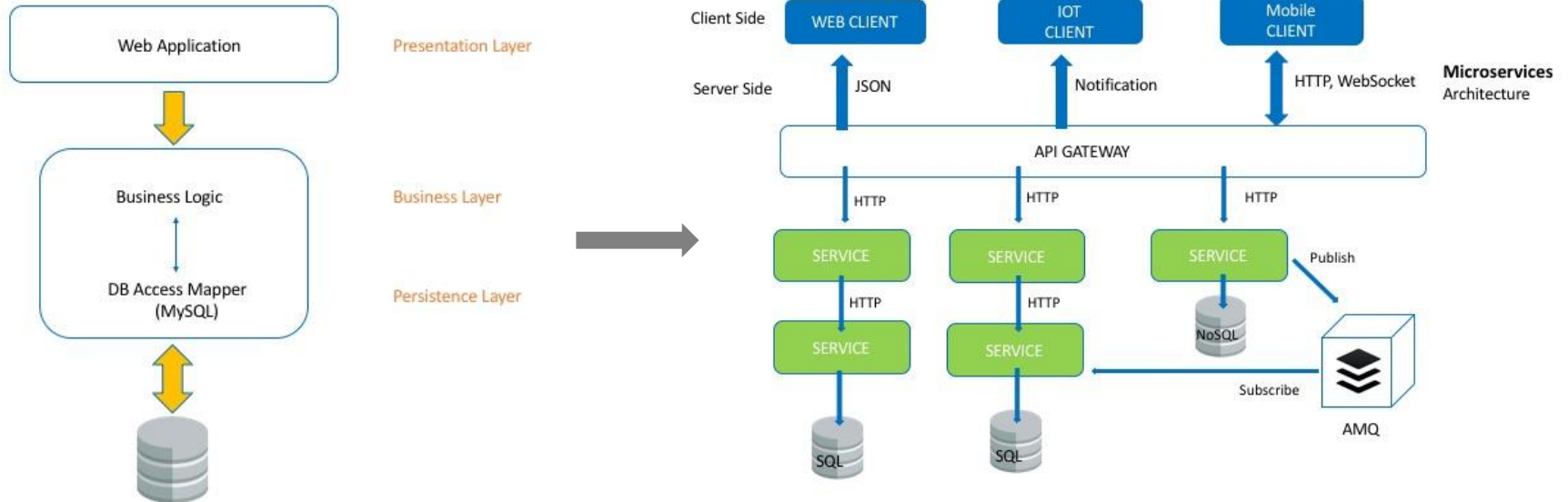
MICROSERVICES

Small decoupled services / API Layer
No shared libraries / common code



What is Microservices Architecture

microservice architecture is a “latest” method of developing software as a suite of small modular and independently deployable applications.

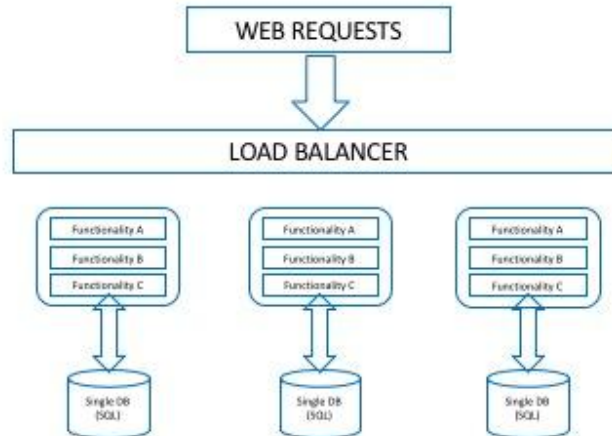


MONOLITHIC ARCHITECTURE



SCALING MONOLITHIC APPLICATION

The whole application scales under LB and not a single component



CONS MONOLITHIC APPLICATION

- A nightmare to manage in large scale due to the fact that developers keeps on adding new features and changes which makes it complex and difficult to fully understand what does what
- Everything is written and coded in the same programming language
- Each deploy -> full deploy to the whole system
- Reliability and Bug hunting (Full regression anyone?)
- Change / Upgrade framework - Easier to land on the moon (again)

Microservices - Hot topic (2014)

The main idea:

is to split our application into a set of smaller, independent but interconnected services instead of building a single monolithic application. Each microservice runs its own process and communicate to other services with a lightweight mechanism, often an HTTP APIs. In a microservice architecture, services are modeled as isolated units that manage a reduced set of functionality

In a microservice architecture topology applications don't have direct access to the backend services. Instead, communication is mediated by an upper layer know as API Gateway!.

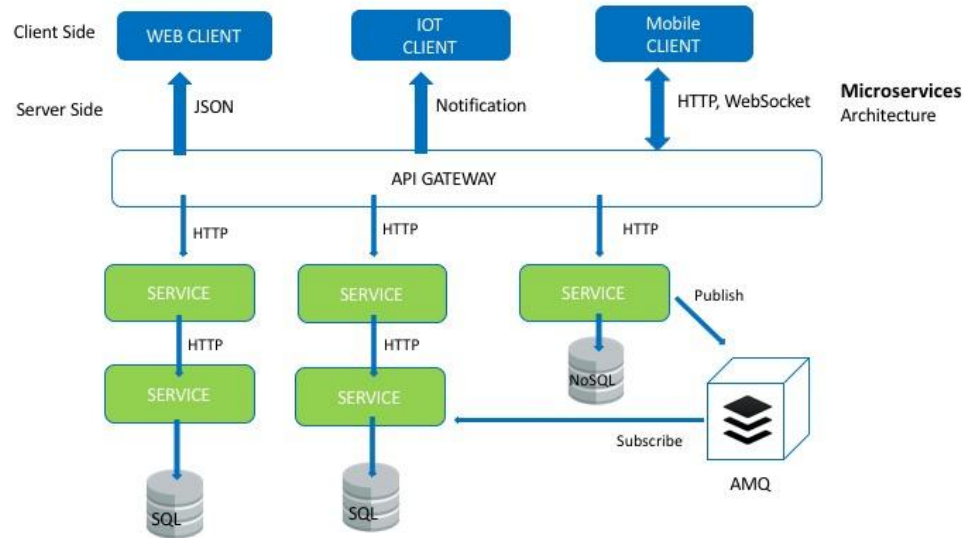
Microservices - The API Gateway

The API Gateway hides the endpoints of the internal services and exposes different endpoints to the client application. In addition, it is responsible for tasks such as load balancing, caching, access control and monitoring.

Communication between Microservices?

Can be done in two main ways: HTTP (LB<>API) and message queue

Microservices - The API Gateway



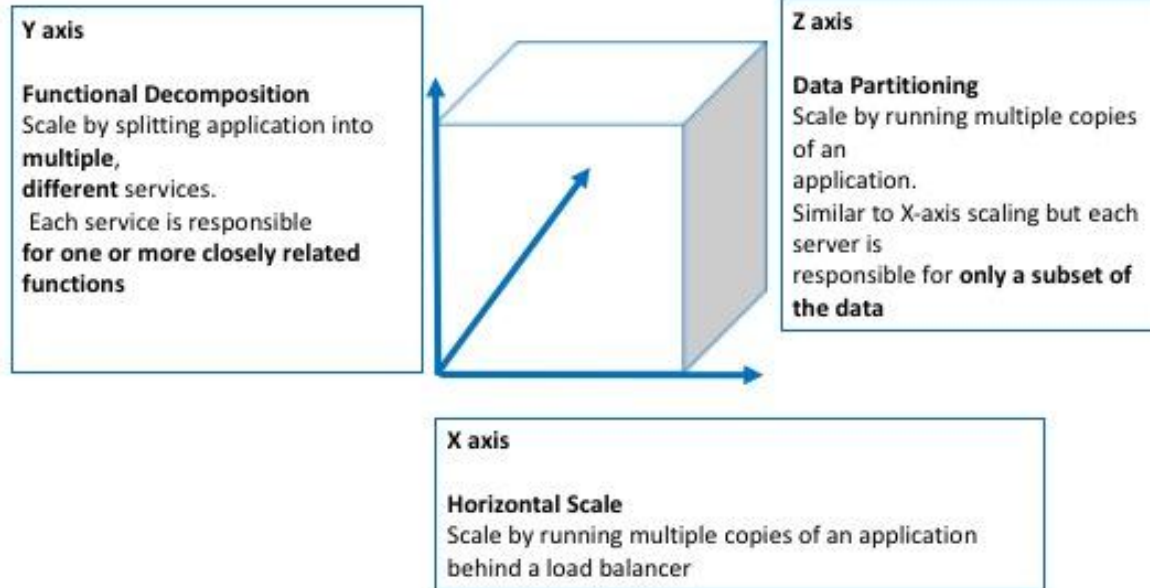
Microservices & DB Relationship

“instead of sharing a single database with other services, each service has its own database”

A different types of DB for different services/tasks finally can be easily integrated into our workspaces because the databases are not shared with others services. If one service needs some information, it will request that data using a **REST API** call to a specific service. The format of the data is usually JSON.

Microservices: Scaling

Choose your war tools:



Microservices Benefits

In Summary

- application complexity by decomposing an application into a set of manageable services which are faster to develop and much easier to understand and maintain.
- Splitting a big application in a set of smaller services also improve the fault isolation. In fact, it is more difficult for the whole system to down at the same time and a failure in processing one customer's request is less likely to affect other customer's requests.
- In second place, developers are free to choose whatever technologies make sense for their service. We are not more bound to the technologies chosen at the start of the project - **Wooo Hooo!**
- easier for a new developer to understand the reduced set of functionality of a microservice instead of understand a big application design

Microservices Benefits continue

- Each microservice could be refactored piece by piece as new technology solutions become available
- Language agnostic, so we can select the appropriate language or framework for each service
- each service can be deployed independently by a team that is focused on that service, using different technology stacks that are best suited for their purposes. This is great for continuous delivery, allowing frequent releases while keeping the rest of the system stable.
- Continuous deployment possible for complex applications and enables each service to be scaled independently.

Microservices DownSides

No free lunch remember?

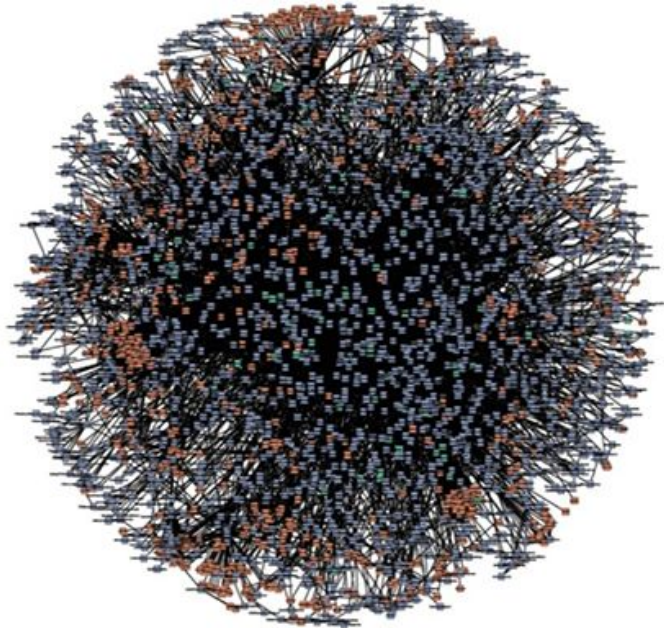
- Microservices architecture is not a silver-bullet to every problem. While it solves a lot of problems of the old monolithic applications, it also introduces new problems that we need to consider
- Microservices architecture add a complexity to the project because a microservice application is a distributed system. For this reason, we need to choose and implement an inter-process communication mechanism base on API request, message queue. On top of that we need to address partial service failure and take into account other issues of distributed system.

Microservices DownSides

No free lunch remember?

- Testing a microservices application is also much more complex than in case of monolithic web application. We need to launch the service we are going to test and any services that it depends on
- Deploying a microservices-based application is also more complex. A monolithic application is simply deployed on a set of identical servers behind a load balancer.
- Last but not least we need a service discovery mechanism in order to successfully deploy a microservices application

MICROSERVICES



amazon.com®



NETFLIX



K8S

What it's all about?

K8S: What is kubernetes?

“Kubernetes is a portable, extensible open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.”

Why Do I Need K8S And What It Is?

Kubernetes has a number of features. It can be thought of as:

- a container platform
- a microservices platform
- a portable cloud platform and a lot more.
- Kubernetes provides a container-centric management environment. It orchestrates computing, networking, and storage infrastructure on behalf of user workloads. This provides much of the simplicity of Platform as a Service (PaaS) with the flexibility of Infrastructure as a Service (IaaS), and enables portability across infrastructure providers.
- K8S Allows developers / system operators to cut to the cord and truly run a container-centric dev / microservice environment

What K8S Is Not?

Kubernetes is **not a traditional, all-inclusive PaaS** (Platform as a Service) system (Incounterary to OpenShift).

Kubernetes operates at the container level rather than at the hardware level, it provides some generally applicable features common to PaaS offerings, such as deployment, scaling, load balancing, logging, and monitoring.

- Does not deploy source code and does not build your application - **CI/CD Does**
- Does not provide application-level services, such as middleware (e.g., message buses), data-processing frameworks (for example, Spark), databases (e.g., mysql), caches, nor cluster storage systems (e.g., Ceph) as built-in services
- Does not dictate logging, monitoring, or alerting solutions.
- Does not provide nor mandate a configuration language/system (e.g., jsonnet). It provides a declarative API that may be targeted by arbitrary forms of declarative specifications.
- Does not provide nor adopt any comprehensive machine configuration, maintenance, management, or self-healing systems.



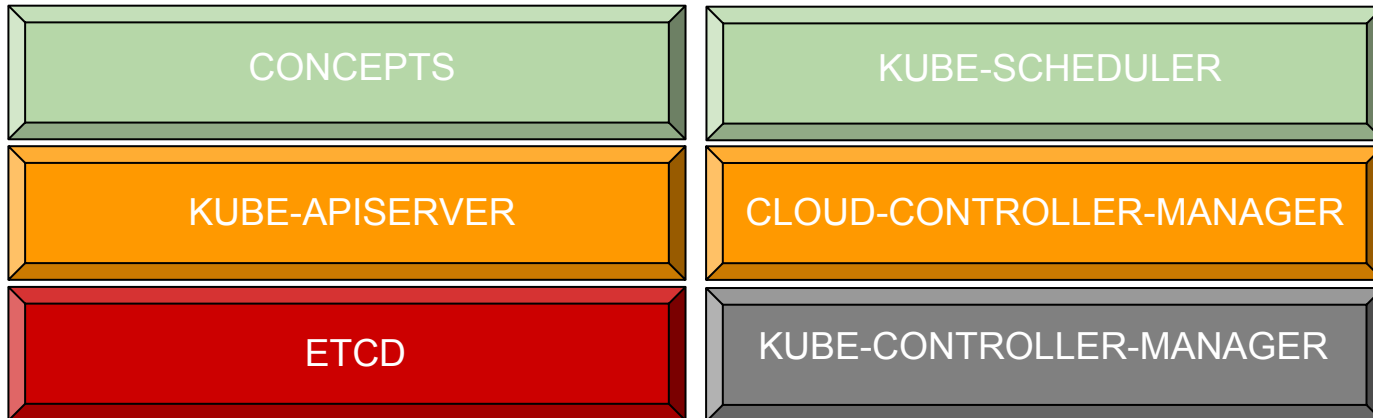
K8S

Core Concepts & Components

K8S: Concepts and Components

Module Agenda: Master Components

To understand k8s concepts , components and Objects



Concepts

Understanding K8S system and abstraction

To work with Kubernetes, we use Kubernetes API objects to describe our cluster's desired state:

- what applications or other workloads we want to run
- What container images they use, the number of replicas, what network and disk resources we want to make available.
- Setting our desired state by creating objects using the Kubernetes API (typically via the command-line interface
 - kubectl)

Once we've set our desired state,

Kubernetes Control Plane works to make the cluster's current state match the desired state

Components

Master components provide the cluster's control plane

- The **Kubernetes Master** which is a collection of **three processes** that run on a single node in our cluster, which is designated **as the master node**. Those processes are:
 - **Kube-apiserver** - Validates and configures data for the api objects which include pods, services, replicationcontrollers, and others.
 - **Kube-controller-manager** - is a an application control loop that watches the shared state of the cluster through the apiserver and makes changes attempting to move the current state towards the desired state.
 - **Kube-scheduler** - its job is to take pods that aren't bound to a node, and assign them one along with hardware/software/policy constraints
- Each **individual non-master** node in our cluster runs two processes:
 - **Kubelet** - which communicates with the Kubernetes Master.
 - **Kube-proxy** - A network proxy which reflects Kubernetes networking services on each node.

Master Node Components: Deep Dive

The Kubernetes Control Plane consists of a collection of processes running on our cluster

- **Kube-Apiserver** - Exposes the Kubernetes API. It is the front-end for the Kubernetes control plane. It is designed to scale horizontally – that is, it scales by deploying more instances.
- **etcd** - highly-available key value store used as Kubernetes' backing store for all cluster data
- **kube-Scheduler** - 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.

Master Node Components: Deep Dive

The Kubernetes Control Plane consists of a collection of processes running on our cluster

- **Kube-controller-manager** which includes separate process but for complexity reduction they are running as one binary
 - **Node Controller** - Responsible for noticing and responding when nodes go down
 - **Replication controller** - Responsible for maintaining the correct number of pods for every replication controller object in the system
 - **Endpoints Controller** - Populates the Endpoints object (that is, joins Services & Pods)
 - **Service account & Token Controllers** - Create default accounts and API access tokens for new namespaces

Kubernetes supports multiple virtual clusters backed by the same physical cluster. These virtual clusters are called namespaces.

Master Node Components: Deep Dive

The Kubernetes Control Plane consists of a collection of processes running on our cluster

- **Cloud-controller-manager** - runs controllers that interact with the underlying cloud providers.
cloud-controller-manager allows cloud vendors code and the Kubernetes core to evolve independent of each other and develops functionality (by the cloud providers) that will be linked to K8S cloud-controller-manger.
- The following controllers have cloud provider dependencies
 - **Node Controller**: For checking the cloud provider to determine if a node has been deleted in the cloud after it stops responding
 - **Route Controller**: For setting up routes in the underlying cloud infrastructure
 - **Service Controller**: For creating, updating and deleting cloud provider load balancers
 - **Volume Controller**: For creating, attaching, and mounting volumes, and interacting with the cloud provider to orchestrate volumes

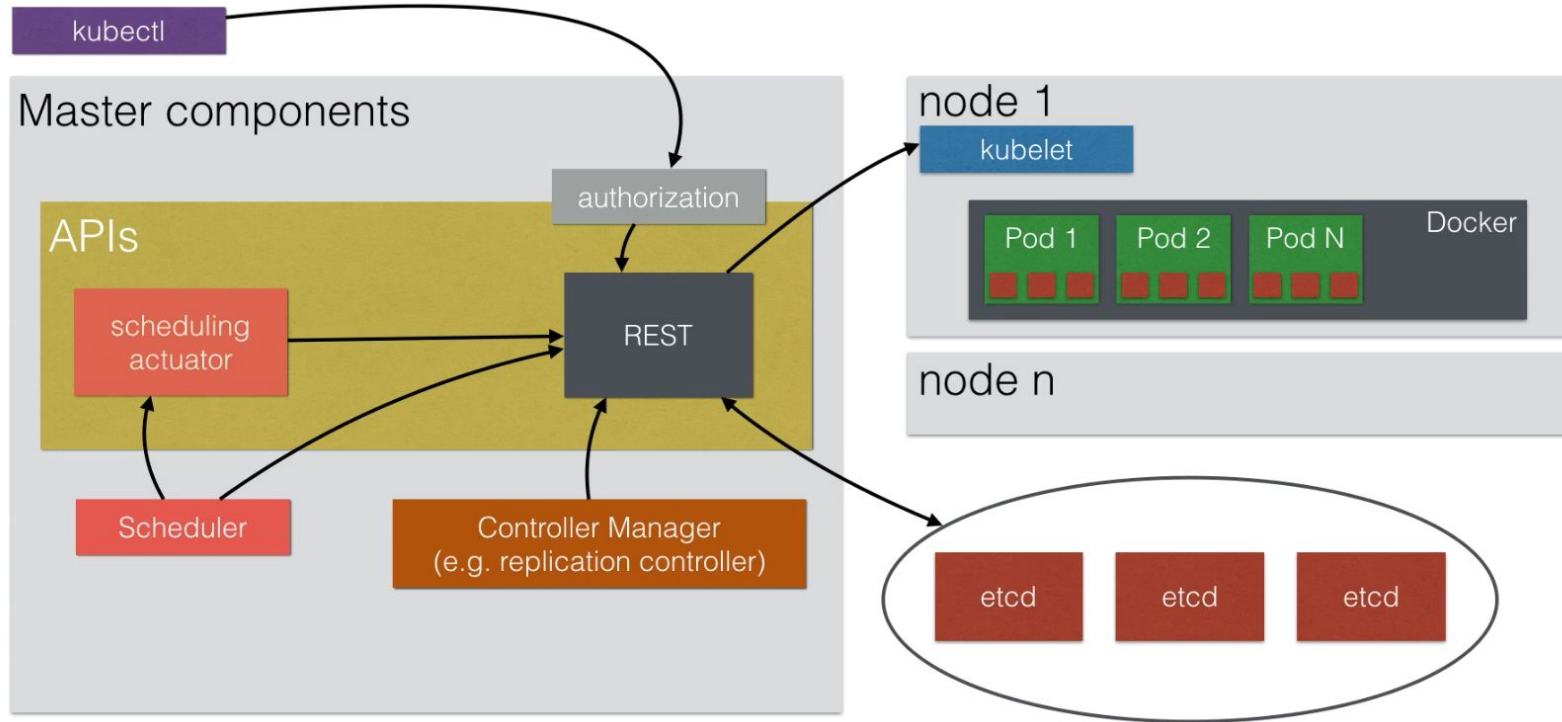
Objects

The Kubernetes Control Plane consists of a collection of processes running on our cluster

- The **Kubernetes Master** which is a collection of three processes that run on a single node in our cluster, which is designated **as the master node**. Those processes are:
 - **Kube-apiserver** - Validates and configures data for the api objects which include pods, services, replicationcontrollers, and others.
 - **Kube-controller-manager** - is a an application control loop that watches the shared state of the cluster through the apiserver and makes changes attempting to move the current state towards the desired state.
 - **Kube-scheduler** - its job is to take pods that aren't bound to a node, and assign them one along with hardware/software/policy constraints
- Each individual non-master node in our cluster runs two processes:
 - **Kubelet** - which communicates with the Kubernetes Master.
 - **Kube-proxy** - A network proxy which reflects Kubernetes networking services on each node.

K8S: Components

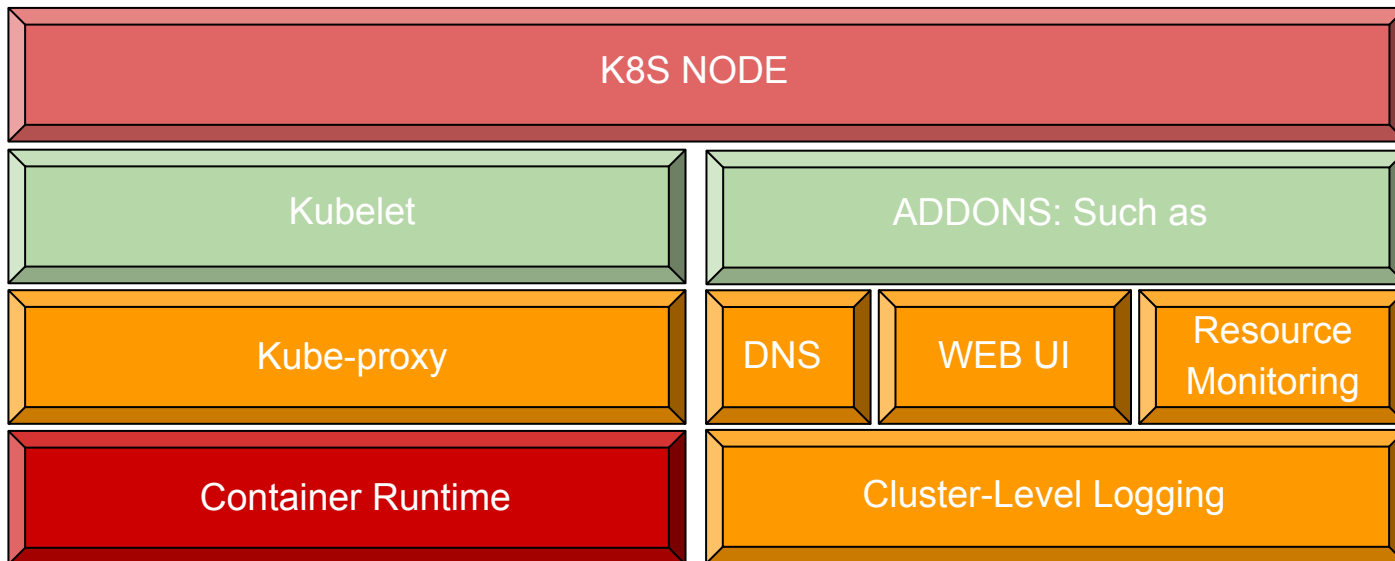
Master Node Components: Deep Dive



K8S: Concepts and Components

Module Agenda: Node Components

To understand k8s concepts , components and Objects



NODES

A worker machine in Kubernetes (AKA minion - but not used anymore)

- **Node:** may be a VM or physical machine, depending on the cluster. Each node has the services necessary to run pods and is managed by the master components. The services on a node include Docker, kubelet and kube-proxy. See The Kubernetes Node section in the architecture design doc for more details.

NODE Components

Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.

- **kublet** - 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. **The kubelet doesn't manage containers which were not created by Kubernetes.**
- **kube-proxy** - enables the Kubernetes service abstraction by maintaining network rules on the host and performing connection forwarding
- **Container Runtime** - The container runtime is the software that is responsible for running containers. Kubernetes supports two runtimes: Docker and rkt (coreOS).

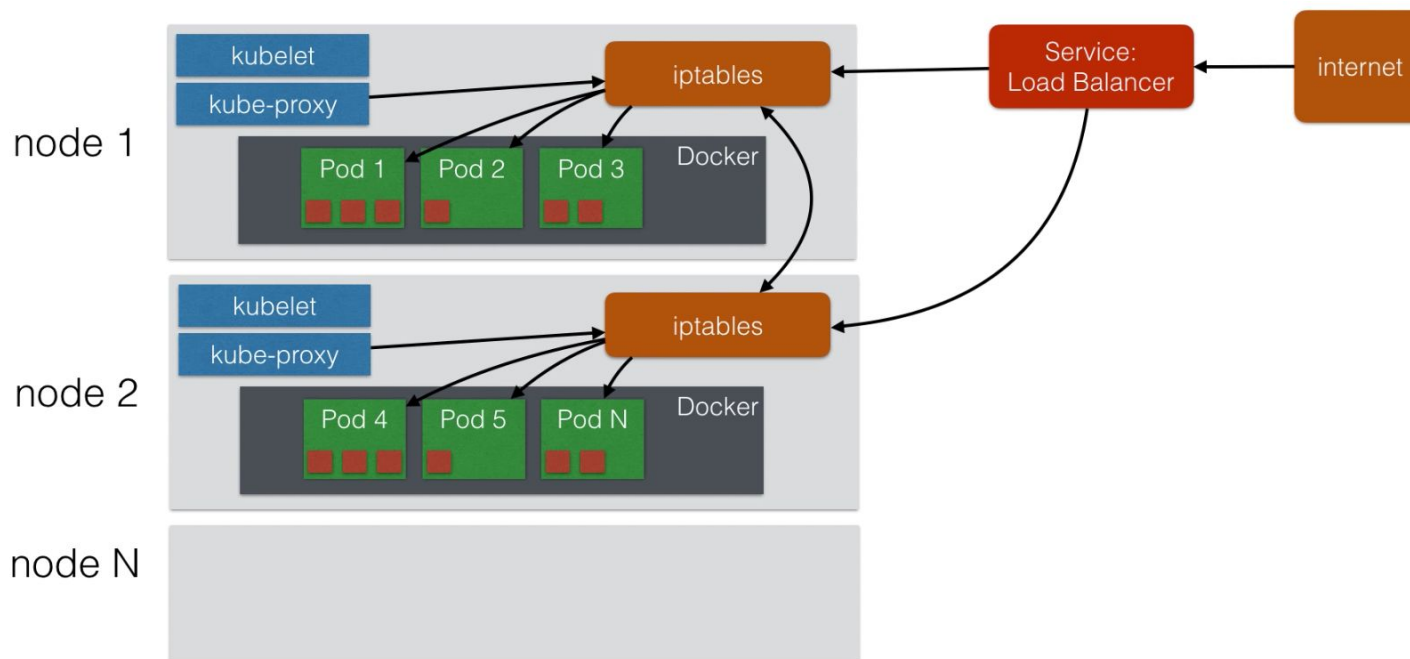
NODE Addons

Addons are pods and services that implement cluster features

- **DNS** - While the other addons are not strictly required, **all Kubernetes clusters should have cluster DNS**, as many examples rely on it. Cluster DNS is a DNS server, in addition to the other DNS server(s) in your environment, which serves DNS records for Kubernetes services. Containers started by Kubernetes automatically include this DNS server in their DNS searches.
- **WEB UI** - General purpose, web-based UI for Kubernetes clusters. It allows users to manage and troubleshoot applications running in the cluster, as well as the cluster itself
- **Container Resource monitoring** - Records generic time-series metrics about containers in a central database, and provides a UI for browsing that data
- **Cluster-level Logging** - A Cluster-level logging mechanism is responsible for saving container logs to a central log store with search/browsing interface.

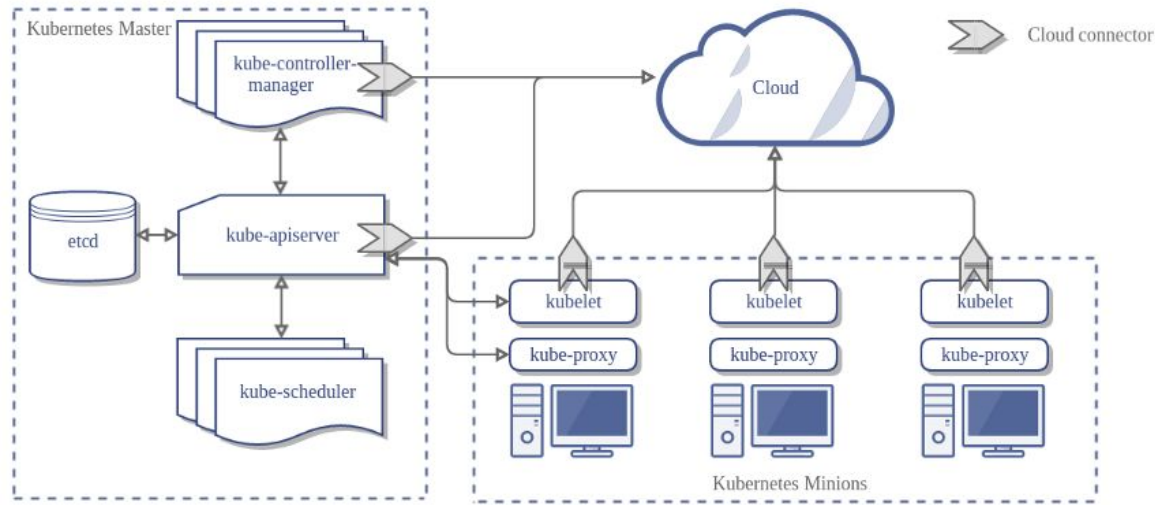
Concepts

NODE Architecture



Concepts

Summarizing Concepts



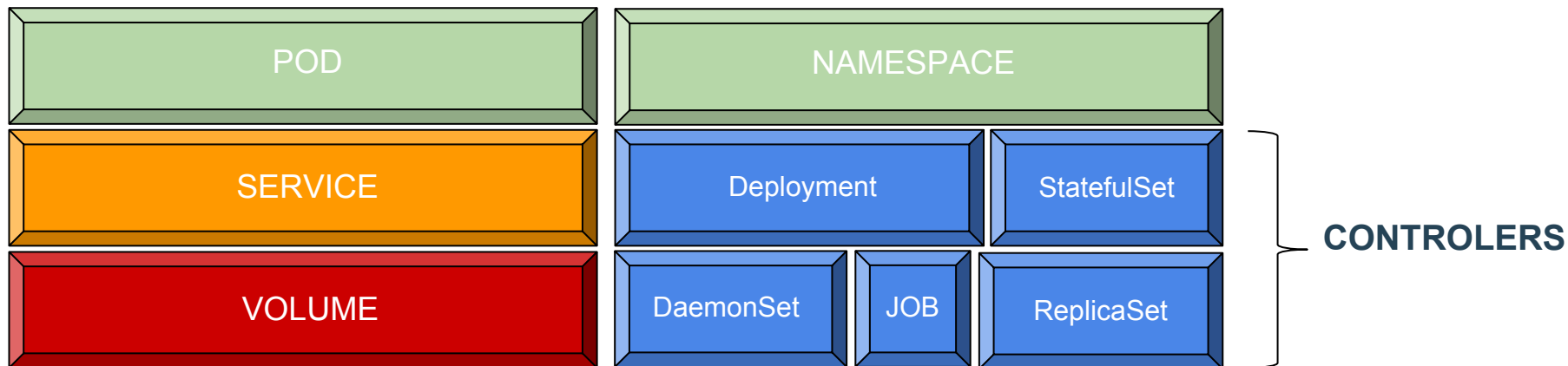


K8S

OBJECTS

Module Agenda: K8S OBJECTS

THE BUILDING BLOCKS



K8S OBJECTS: PODS

The basic Building blocks

- **POD** - A Pod is the basic building block of Kubernetes—the smallest and simplest unit in the Kubernetes object model that you create or deploy. A Pod represents a **running process** on your cluster. A Pod encapsulates an application container (or, in some cases, multiple containers), storage resources, a unique network IP, and options that govern how the container(s) should run.

There are two ways to describe a POD in a K8S cluster

- **Pods that run a single container** - The “one-container-per-Pod” model is the most common Kubernetes use case; in this case, you can think of a Pod as a wrapper around a single container, and Kubernetes manages the Pods rather than the containers directly
- **Pods that run multiple containers that need to work together**

K8S OBJECTS: PODS

The basic Building blocks

There are two ways to describe a POD in a K8S cluster

- **Pods that run a single container** - The “one-container-per-Pod” model is the most common Kubernetes use case; in this case, you can think of a Pod as a wrapper around a single container, and Kubernetes manages the Pods rather than the containers directly
- **Pods that run multiple containers that need to work together** - A Pod might encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources. These co-located containers might form a single cohesive unit of service—one container serving files from a shared volume to the public, while a separate “sidecar” container refreshes or updates those files. The Pod wraps these containers and storage resources together as a single manageable entity.

K8S OBJECTS: PODS

Pods provide two kinds of shared resources for their constituent containers: networking and storage.

- **Networking** - Each Pod is assigned a unique IP address. **Every container in a Pod shares the network namespace, including the IP address and network ports.** Containers inside a Pod can communicate with one another using localhost. When containers in a Pod communicate with entities outside the Pod, they must coordinate how they use the shared network resources (such as ports).
- **Storage** - A Pod can specify a set of shared storage volumes. All containers in the Pod can access the shared volumes, allowing those containers to share data. Volumes also allow persistent data in a Pod to survive in case one of the containers within needs to be restarted. See Volumes for more information on how Kubernetes implements shared storage in a Pod

K8S OBJECTS: PODS & Controllers

Self-healing, Replication, Rollout and more are provided by Controllers

- **Deployment** - A controller that provides declarative updates for Pods and ReplicaSets. We describe a desired state in a Deployment object, and the Deployment controller changes the actual state to the desired state at a controlled rate.
- **StatefulSet** - workload API object used to manage stateful applications.
 - StatefulSets are valuable for applications that require one or more of the following.
 - Stable, unique network identifiers.
 - Stable, persistent storage. Ordered, graceful deployment and scaling.
 - Ordered, graceful deletion and termination.
 - Ordered, automated rolling updates.

K8S OBJECTS: PODS & Controllers

Self-healing, Replication, Rollout and more are provided by Controllers

- **DaemonSet** - A DaemonSet ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created.
 - **Some typical uses of a DaemonSet are:**
 - running a cluster storage daemon, such as glusterd, ceph, on each node.
 - running a logs collection daemon on every node, such as fluentd or logstash.
 - running a node monitoring daemon on every node, such as Prometheus Node Exporter, collectd, Datadog agent, New Relic agent, or Ganglia gmond.

K8S OBJECTS: SERVICE

Pods have a short lifetime, They are born and when they die, they are not resurrected there is no guarantee about the IP address they are served on.

ReplicationControllers in particular create and destroy Pods dynamically (e.g. when scaling up or down or when doing rolling updates). This could make the communication of microservices hard. Imagine a typical Frontend communication with Backend services - how do those frontends find out and keep track of which backends are in that set?

Hence K8s has introduced the concept of a service

which is an abstraction on top of a number of pods, typically requiring to run a proxy on top, for other services to communicate with it via a Virtual IP address. This is where you can configure load balancing for your numerous pods and expose them via a service.

K8S OBJECTS: VOLUME

On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. First, **when a container crashes, kubelet will restart it, but the files will be lost** - the container starts with a clean state. Second, when running containers together in a Pod it is often necessary to share files between those containers. The Kubernetes Volume abstraction solves both of these problems.

K8S OBJECTS: NAMESPACE

Kubernetes supports multiple virtual clusters backed by the same physical cluster. These virtual clusters are called namespaces. **Namespaces** are intended for use in environments with many users spread across multiple teams, or projects. For clusters with a few to tens of users, you should not need to create or think about namespaces at all. Start using namespaces when you need the features they provide such as: **divide cluster resources between multiple users / envs / regions / hw type's**

K8S OBJECTS: JOB

A **job** creates one or more pods and ensures that a specified number of them successfully terminate. As pods successfully complete, the job tracks the successful completions. When a specified number of successful completions is reached, the job itself is complete. Deleting a Job will cleanup the pods it created.

A simple case is to create one Job object in order to reliably run one Pod to completion. The Job object will start a new Pod if the first pod fails or is deleted (for example due to a node hardware failure or a node reboot). A Job can also be used to run multiple pods in parallel.



QUESTIONS BEFORE WE REALLY START?



K8S

HANDS ON LABS



K8S

HANDS ON: BASIC

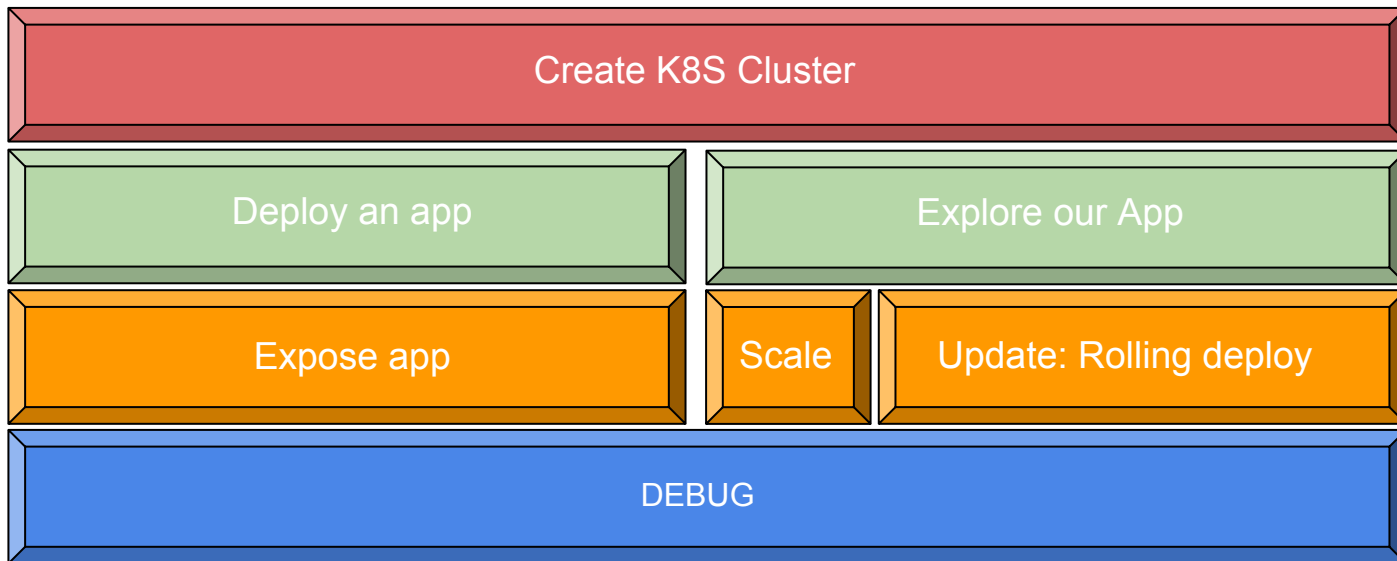
K8S LABS: BASIC

Basic modules to learn and complete

DEPLOYING AN APPLICATION

K8S LABS: BASIC

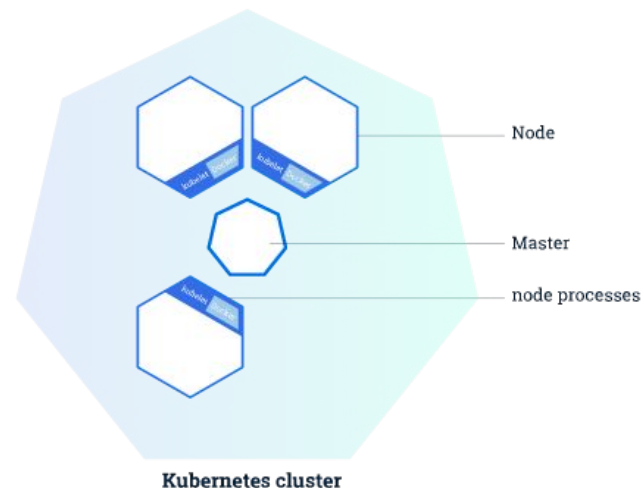
Module Agenda: Basics Kubernetes cluster orchestration



K8S LABS: BASIC

CLUSTER DIAGRAM

- The **Master** is responsible for managing & coordinates all the cluster activities such as desire state, scaling, rolling new updates.
- A **Node** is a VM or a physical computer that serves as a worker machine in a Kubernetes cluster
- The **nodes communicate with the master** through the Kubernetes API (using **Kubelet** agent for communication and management of the node)



K8S LABS: BASIC

- Get Cluster version:

```
kubectl version  
Client Version:.....
```

- Get Cluster info:

```
kubectl cluster-info  
Kubernetes master is running at http://....
```

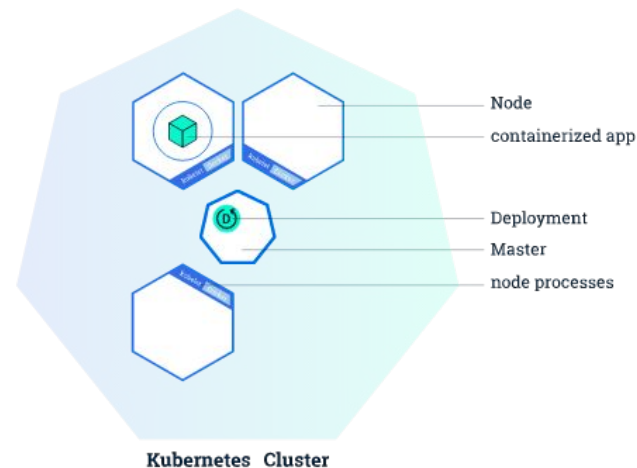
- Get get nodes:

```
kubectl get nodes  
  
NAME          STATUS    ROLES    AGE      VERSION  
host01        Ready     <none>    3m       v1.9.0
```

K8S LABS: BASIC

Create a deployment

- To deploy a new Application we will use and create k8s **Deployment** configuration. Which in turn instruct K8S how to create and update the instances of our application.
- Note that once a new application instances are created, **K8S deployment-controller** will continuously monitor those instances and if one will fail - it will be replaced by the **deployment-controller**.

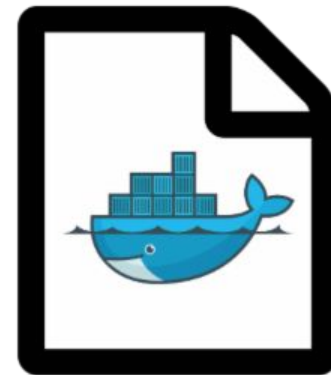


K8S LABS: BASIC

Deploying our spring music application - in memory DB

First let's create a docker file

- Make a new folder in your project directory called `jb_dockerfile`
 - **Copy** `spring-music.jar` from `/seminars/docker/artifacts` to a new folder `jb_dockerfile/artifacts`
 - **Create** an empty dockerfile with the following SPEC
 - **From:** `frolvlad/alpine-oraclejdk8:slim`
 - **Workdir** `/data`
 - **Copy:** artifact to `/data`
 - **Expose:** 8080
 - **CMD:** `java -jar -Dspring.profiles.active=none /app/spring-music.jar`
 - **Build:** `docker build -t repo/imagename:version` & Run it & Push to repo



K8S LABS: BASIC

Deploying our spring music application to our K8S Cluster

Command line:

```
kubectl run [deployment name] --image [repo/image:ver] --port=[app listen port inside the container]
```

Run

```
run spring-music --image yanivomc/spring-music:latest --port=8080
```

K8S LABS: Troubleshooting

Deploying our spring music application to our K8S Cluster

K8S Common commands to check if our application got deployed are:

```
alias k=kubectl # Making k an alias of kubectl command
K get pods OR K get deployment OR K get services
# Using Describe
k describe pods/[pod name] OR k describe deployment/[app-name] .....
# Viewing LOGS
K logs pods/[pod name] # (Get logs of all running containers in the POD) OR K logs deployment/app-name
```

K8S LABS: Using PROXY to connect

Deploying our spring music application to our K8S Cluster

Browse our Deployed POD using our proxy

```
# Run the proxy in a new terminal  
k proxy --port=8082 &
```

The proxy command above expose K8S API

```
# Test the API  
Curl http://localhost:8082/version # will give us the K8S version
```

K8S LABS: Using PROXY to connect

Deploying our spring music application to our K8S Cluster

The **kubectl** command can create a proxy that will forward communications into the cluster-wide, private network.

The proxy can be terminated by pressing control-C and won't show any output while its running

1. Get the pod name
2. Run the following:

```
curl http://localhost:8082/api/v1/proxy/namespaces/default/pods/\[pod-name\]/
```

```
Or browse http://\[k8s-master.IP\]:8082/api/v1/proxy/namespaces/default/pods/\[pod-name\]/
```

How did the proxy knew to pass traffic to that pod application port?

This is the start of Our **service** configuration and LB.

Kill the PROXY



K8S

Exposing our APP to the world

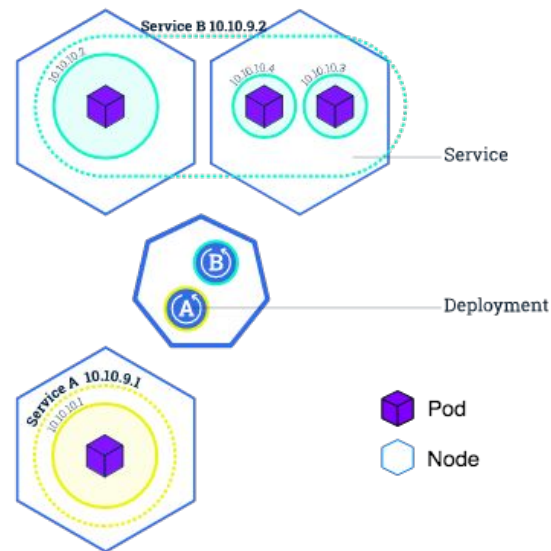
K8S BASIC : SERVICES

EXPOSING OUR APP

Reminder:

Kubernetes Pods are mortal. Pods in fact have a life cycle. When a worker node dies, the Pods running on the Node are also lost. A ReplicationController might then dynamically drive the cluster back to desired state via creation of new Pods to keep your application running.

A Service in Kubernetes is an abstraction which defines a logical set of Pods and a policy by which to access them. Services enable a loose coupling between dependent Pods. A Service is defined using YAML (preferred) or JSON, like all Kubernetes objects. The set of Pods targeted by a Service is usually determined by a LabelSelector



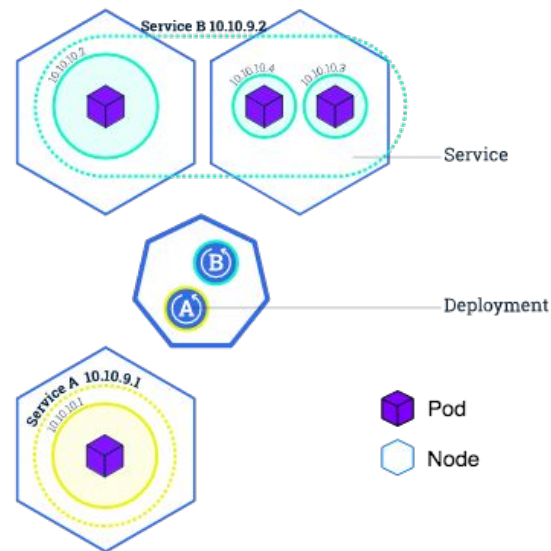
K8S BASIC : SERVICES Continue

EXPOSING OUR APP

Each Pod has a unique IP address, **those IPs are not exposed outside the cluster without a Service**. Services allow our applications to receive traffic.

Services can be exposed in different ways by specifying a **type** in the ServiceSpec:

- **ClusterIP** (Default) - **Exposes** the Service on an internal IP in the cluster
- **NodePort** - **Exposes** the Service on the same port of each selected Node in the cluster using NAT. Makes a Service accessible from outside the cluster using <NodeIP>:<NodePort>
- **LoadBalancer** - Creates an **external load balancer** in the current **cloud** (if supported) and assigns a fixed, external IP to the Service. Superset of NodePort
- **ExternalName** - Exposes the Service using an arbitrary name by returning a CNAME record with the name.



K8S BASIC : SERVICES Continue

EXPOSING OUR APP: Label Selectors

Services and LabelsAndSelectors

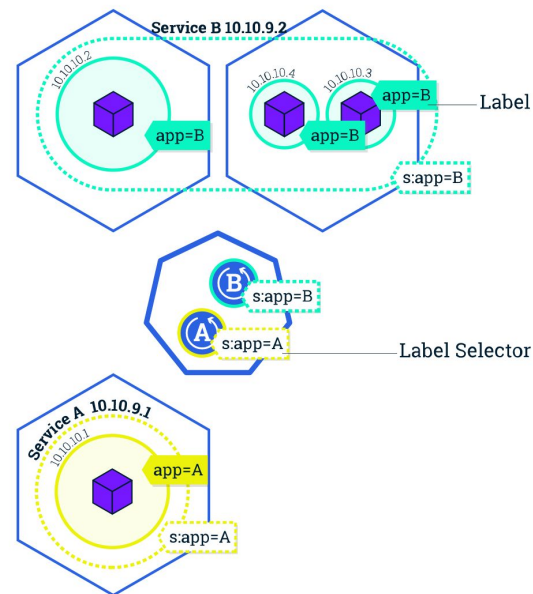
A **Service** routes traffic across a set of Pods.

Services are the abstraction that allow pods to die and replicate in K8S without impacting our application.

Discovery and routing among dependent Pods (such as the frontend and backend components in an application) is handled by K8S Services.

Services match a set of Pods **using labels and selectors**, a grouping primitive that allows logical operation on objects in K8S.

Labels are key/value pairs attached to objects and can be used in any number of ways

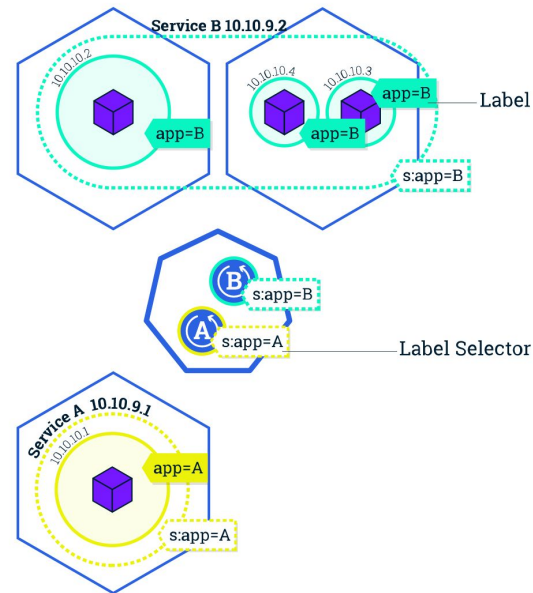


K8S BASIC : SERVICES Continue

Label and Selectors - Options to use it

Use Labels and selectors to group objects as:

- Designate for Development, Test and production
- Embed version tags
- Classify an object using tags





K8S

HANDS ON: Exposing our App and labeling objects

K8S LABS: Exposing our application

Deploying our spring music application to our K8S Cluster

Exposing a service in K8S for external access is done as followed:

```
kubectl expose deployment/[deployment name] --type=["NodePort","LoadBalancer"...] --port=[source]
--target-port=[Container/service port]
```

* Note that if we dont type --port - The Service will expose the port described in our deployment automatically

List & describe the services

```
kubectl get services
kubectl describe services/[service name]
```

K8S LABS: Exposing our application

Deploying our spring music application to our K8S Cluster

In this Lab we will first use “NodePort” to expose a node Endpoint and port and than we will use “LoadBalancer”

```
[centos@ip-172-60-2-10 ~]$ k describe services/spring-music
```

```
Name:                spring-music
Type:                NodePort
IP:                  10.233.14.71
Port:                <unset> 8080/TCP
TargetPort:          8080/TCP
NodePort:            <unset> 30602/TCP
Endpoints:           10.233.116.71:8080
Session Affinity:    None
External Traffic Policy: Cluster
Events:              <none>
```

K8S LABS: Exposing our application

Deploying our spring music application to our K8S Cluster

Type="LoadBalancer" will create and configure a **LB (elb) in AWS** and configure ingress traffic to our node

```
[centos@ip-172-60-2-10 ~]$ k describe services/spring-music
```

```
Name:                spring-music
```

```
.....
```

```
.....
```

```
LoadBalancer Ingress:  a19ecf45e2b9e11e8b20906d3d8c259d-1228201420.eu-central-1.elb.amazonaws.com
```

```
Port:                <unset> 8080/TCP
```

```
TargetPort:          8080/TCP
```

```
....
```

Type	Reason	Age	From	Message
----	-----	----	----	-----
Normal	EnsuringLoadBalancer	33s	service-controller	Ensuring load balancer

K8S LABS: Exposing our application

Deploying our spring music application to our K8S Cluster

Using Labels - The deployment created an automatic label for our pod.

```
#Get the deployment label
kubectl describe deployment/[NAME]

.....

Labels:                run=spring-music
```

Using Labels to run commands

```
kubectl get services -l run=spring-music
kubectl get pods -l run=spring-music
```

K8S LABS: Exposing our application

Deploying our spring music application to our K8S Cluster

Using Labels - Adding label

```
kubectl label pod/service/deploy [pod-name] [key]=[value]
```

K8S: Exposing our app

K8S LABS: Create new service

Using **kubect**l create new service for our Deployment using both types and browse the service.

**** When using type NodePort you will only be able to browse it from the remote K8S Master and not your local machine in our Lab.

Using **kubect**l Label your **Pod** and your **Deployment** and than run:

```
kubectl get [object] -l [key]=[value] # that you created  
kubectl get all -l [key]=[value] # that you created
```

Delete the service we created which will only close the connection to our application deployment

```
kubectl get services  
kubectl delete service/[service name]
```



K8S

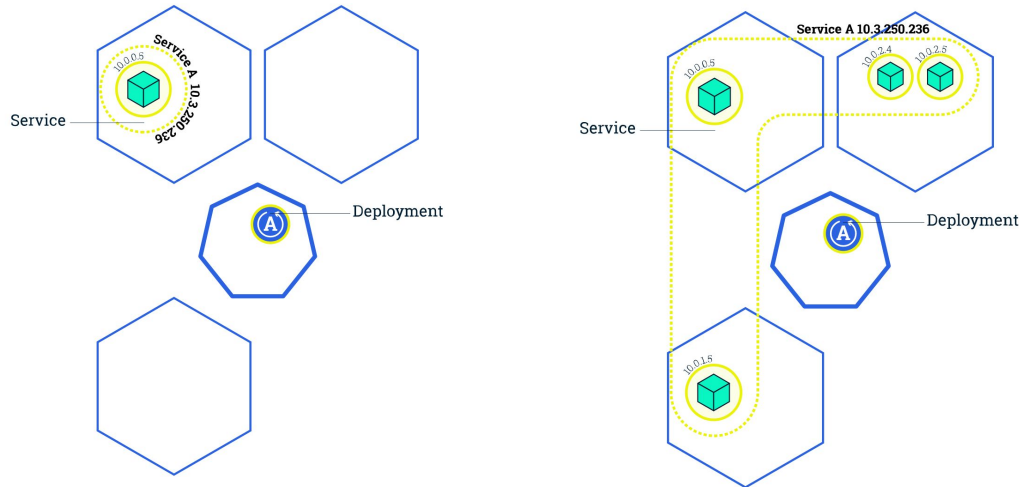
Scaling our application

K8S: Scale our application

K8S: Scaling

Using **kubectl** we can scale our instance application by running it on multiple pods and balance it on multiple nodes.

Scaling is accomplished by setting the number of replicas in a **Deployment** creation or runtime using `--replicas=[num]`



K8S: Scale our application

K8S: Scaling

Scaling out a Deployment will ensure new Pods are created and scheduled to Nodes with available resources while **Scaling in** will reduce the number of Pods to the new desired state.

K8S also supports **autoscaling** of Pods, but it is outside of the scope of this course.

Scaling to zero will terminate all Pods of the specified Deployment.

Once we have Multiple instances of our application,

Rolling deployments will be possible.

K8S: Scale our application

K8S: Scaling

Scaling out

```
kubectl get deployments/[name]
```

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
spring-music	1	1	1	1	2h

Scaling out example

```
# Scale command
kubectl scale deployment/[name] --replicas=[num]

# Get detailed deployment status and pods per node
kubectl get pods -o wide

# Run describe to view events and replicas status
Kubectl describe deployment/[name]
```



K8S

HANDS ON: Scale to 4 and Add LB

K8S: Scale our application

K8S: LAB

Scale out our deployment and add LoadBalancer service

1. **Scale** out our deployment to **min 4 instances**
2. Run `kubectl get pods -o wide`
3. **Add a service** as we learned in the previous lab type=**LoadBalancer**
 - a. Get the LB EndPoint CNAME (it takes a few moments to create)
 - b. Review using `describe` that the new service “sees” all of the endpoints
 - c. Browse the application using the cname and port

K8S: Scale our application

K8S: LAB

Testing self healing pods

Test Self Healing by killing a pod (stopping the process inside the container)

```
# Copy one of the pods that runs our application instance
kubectl get pods
# Tail (WATCH) our deployment status in a second terminal
kubectl get deployment [name] --watch
# Get Bash/Ash access to one of the containers (pods) by running Docker kubectl exec
kubectl exec -ti [podname] ash

# Kill the Java process using
ps aux | grep java
kill -9 [java PID]
```



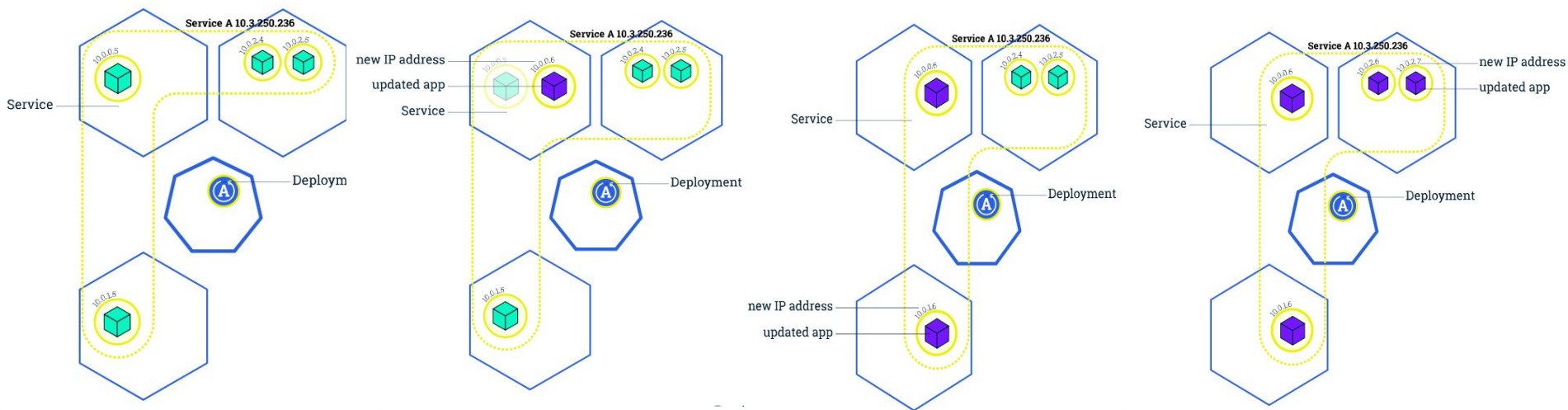
K8S

Rolling Deployments

K8S: Rolling Updates

K8S: Rolling Updates

Rolling updates allow Deployments' update to take place with zero downtime by incrementally updating Pods instances with new ones. The new Pods will be scheduled on Nodes with available resources. Same goes for Reverts !



K8S: Rolling Updates

K8S: Rolling Updates

Rolling updates allow the following actions:

- Promote an application from one environment to another (via container image updates)
- Rollback to previous versions
- Continuous Integration and Continuous Delivery of applications with zero downtime

K8S LABS: Updating our application

To update the image of our application from version X to version Y, we use the **set image** command followed by the deployment Name and the new image **version**

```
kubectl set image deployments/[name] [application name]=repo/image:version
```

The above command will notify the Deployment to use different image while the app start a rolling update.

```
# Check the status of the new pods using
kubectl get pods
# Verify rollout completion
kubectl rollout status deployments/[name]

# Rollback:
kubectl rollout undo deployments/[name]
```



K8S

HANDS ON: Rolling Updates

K8S:LAB Rolling Updates

- **Rollout new application version**
 - Image name: nginx:1.13
- Verify rollout
- Smoke test the application by browsing it

- **RollBack to the latest stable application version**
 - First Rollout an application version that dosent exists
 - Image name: nginx:latest
 - Run **get deployments** to see that deploy failed
 - Run UNDO

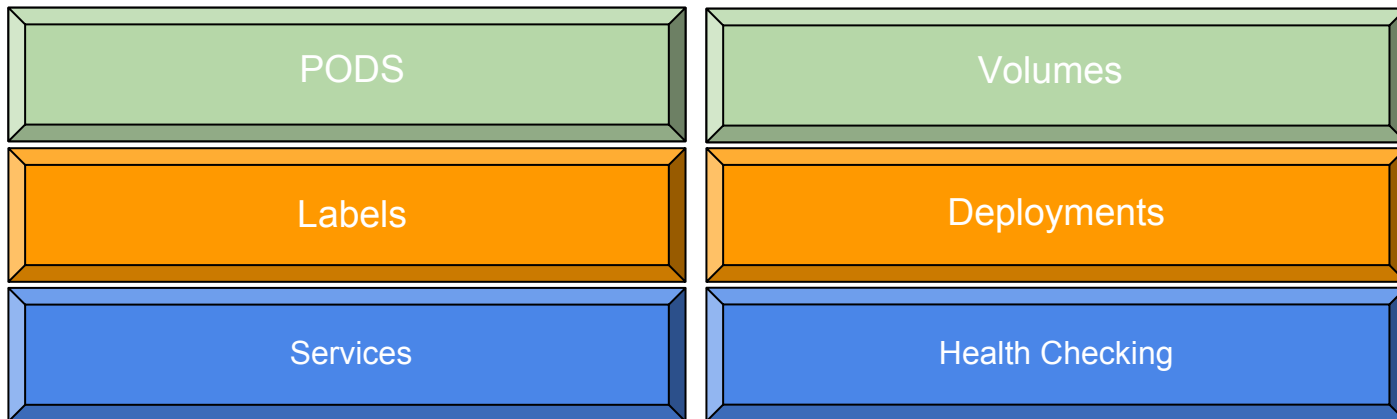


K8S

Basics Done! - What next?

Moving On: Part 1

Module Agenda: Pods management and Volumes



K8S: Moving on - Part 1

K8S PODS: Management

As you now know - In K8S, a group of one or more containers is called a pod.

Containers in a pod are deployed together, and are started, stopped, and replicated as a group

*** In K8S same as in Docker we can interact with the CLI to build and manage our objects but in real life we would like

To Automate Manual tasks by using and building Object definitions in “Yet Another Markup Language” - Hooray

The following is an example of how we describe a pod deployment of type NGINX using object definitions

```
apiVersion: v1 # K8S API Version
kind: Pod # Object Type
metadata:
  name: nginx
spec:
  containers:
    - name: nginx
      image: nginx:1.7.9
      ports:
        - containerPort: 80
```

K8S: Moving on - Part 1

K8S PODS: Management

Important to understand and remember:

A pod definition is a **declaration of a *desired state***. Desired state is a very important concept in the Kubernetes model.

Pod Management: Creating new pod base on our Yaml

```
kubectl create -f /some/folder/pod-nginx.yaml
```

Pod Management: Than list our pods

```
kubecetl get pods
```

Problem:

In most scenarios all pods receives privates IP's and are not accessible outside of the node so..

Question ?

How can we test a pod is working ? -> BuysBox

K8S: Moving on - Part 1

K8S PODS: Testing Pods - BusyBox

BusyBox is a tiny embedded linux VM/Container that has everything you need to run common unix utilities

We will deploy a new pod with BusyBox image into a node and through the busy box we will run EXEC commands

To test our newly created NGINX pod.

```
kubectl run busybox --image=busybox --restart=Never --tty -i --generator=run-pod/v1 --env \
"POD_IP=$(kubectl get pod nginx -o go-template='{{.status.podIP}}')"
```

*** --generator allows kubectl run to generate resources and in this scenario - a POD - see [link](#) for more info

Once the the above command complete - We will have access to the BusyBox container

There we will make a wget call to the NGINX POD

```
u@busybox$ wget -qO- http://$POD_IP # Run in the busybox container
u@busybox$ exit
$kubectl delete pod busybox && kubectl delete pod nginx ## Cleanup
```

K8S: Moving on - Part 1

K8S PODS: Volumes

Persistent storage!

Same as in docker (but not really the same as docker can support only one Volume driver per container and K8S multiple drivers as described [here](#)) we use Volume to store persistent data cross containers which will outlive containers restarts/crashes BUT **will be deleted if the pod/s get deleted if using driver “emptyDir”**

Defining a volume:

```
# 1. We define the volume and driver type
volumes:
  - name: redis-persistent-storage # Volume name
    emptyDir: {}

# 2. We mount the volume within a container definition
volumeMounts: # name must match the volume name defined in volumes
  - name: redis-persistent-storage # mount path within the container
    mountPath: /data/redis
```

K8S: Moving on - Part 1

K8S PODS: Volumes Real life example

Persistent storage!

Creating a volume and mounting it to our previous pod-nginx.yaml.

Volume Types

EmptyDir: Creates a new directory that will exist as long as the Pod is running on the node, but it can persist across container failures and restarts.

HostPath: Mounts an existing directory on the node's file system (e.g. /var/logs).

See [volumes](#) for more details.

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
    - name: redis
      image: redis
      volumeMounts:
        - name: redis-persistent-storage
          mountPath: /data/redis
  volumes:
    - name: redis-persistent-storage
      emptyDir: {}
```

```

apiVersion: v1
kind: Pod
metadata:
  name: xxx
spec:
  containers:
  - name: xxx # FIRST Container
    image: xxx
    volumeMounts:
    - mountPath: /some/folder # Mount location within a container
      name: name-of-volume-created # Volume name that we created
      readOnly: true # Read Only or not
  - name: git-monitor # SECOND Container
    image: XXX
    volumeMounts:
    - mountPath: /xxxx
      name: www-data # Volume name that we created
    env: # ENVIRONMENT Variables
    - name: GIT_REPO # KEY
      value: http://github.com/some/repo.git # VALUE
    command: ["COMMAND"] # Running a shell command on container boot
    args: ["ARG1", "ARG2", "${GIT_REPO}"] # with ENV ARG

# Creating Volumes to be used by the above containers
volumes:
- name: name-of-volume-created
  emptyDir: {} # Volume Drive

```




K8S

HANDS ON: Pods and Volumes

K8S: PART 1 LAB

Create new pod definition file with multiple containers and the following information

- Container 1
 - Container name: nginx
 - Image: nginx
 - Version: latest
 - Ports to expose inside the container: 80
 - Volumes:
 - Name: www-data
 - Type: emptyDir
 - mountPate: /srv/www

K8S: PART 1 LAB

- Container 2
 - Container name: sidecar_staticgen
 - Image: alpine/git
 - Version: latest
 - Volumes:
 - Name: www-data
 - Type: emptyDir
 - mountPath: /data
 - ENV Variable
 - Name: GIT_REPO
 - VALUE: <https://github.com/cfjedimaster/HTML-Code-Demos.git>



K8S

LABELS, DEPLOYMENTS, SERVICES & HealthChecks



K8S

LABELS

K8S: Moving on - Part 1

K8S Labels

Using labels we can organize Pods into groups. The system for achieving this in K8S is Labels. Labels are key-value pairs that are attached to each object in Kubernetes. Label selectors can be passed along with a RESTful list request to the apiserver to retrieve a list of objects which match that label selector.

```
labels:  
  app: nginx
```

K8S: Moving on - Part 1

K8S Labels

NGINX Pod definition with Labels:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  containers:
    - name: nginx
      image: nginx
      ports:
        - containerPort: 80
```

K8S: Moving on - Part 1

K8S Labels

Create the labeled pod

```
kubectyl create -f
https://raw.githubusercontent.com/yanivomc/seminars/master/K8S/code\_examples/walkthrough/pod-nginx-with-label.yaml

# Get pods using label
kubectyl get pods -l app=nginx

# delete pods using label
kubectyl delete pod -l app=nginx
```

More motivations to use labels ?

- "release" : "stable", "release" : "canary"
- "environment" : "dev", "environment" : "qa", "environment" : "production"
- "tier" : "frontend", "tier" : "backend", "tier" : "cache"
- "partition" : "customerA", "partition" : "customerB"
- "track" : "daily", "track" : "weekly"

K8S: Moving on - Part 1

K8S Labels selector - Playing around

Label selector is the core grouping primitive in Kubernetes

1. Set your nodes with different labels as followed:

Node 1 - env=prod , tier=fe

Node 2 - env=prod , tier=be

Node 3 - env=dev , tier=fe

2. The run:

```
kubect1 get nodes -l env=prod,tier=fe
```

```
kubect1 get nodes -l tier=fe
```

```
kubect1 get nodes -l env=prod
```

```
kubect1 get nodes -l 'env in (prod, dev)'
```

K8S: Moving on - Part 1

K8S Labels selector

Label selector is the core grouping primitive in Kubernetes

Step 2:

usage scenario for equality-based label requirement is for Pods to specify node selection criteria. For example, the sample Pod below selects nodes with the label “accelerator=nvidia-tesla-p100”.

```
apiVersion: v1
kind: Pod
metadata:
  name: cuda-test
spec:
  containers:
    - name: cuda-test
      image: "k8s.gcr.io/cuda-vector-add:v0.1"
      resources:
        limits:
          nvidia.com/gpu: 1
  nodeSelector:
    accelerator: nvidia-tesla-p100
```

K8S Affinity and anti-affinity

Affinity and anti-affinity

While nodeSelector provides a very simple way to constrain pods to nodes with particular labels.

The affinity/anti-affinity feature, currently in **BETA**, greatly expands the types of constraints you can express. The key enhancements are

1. the language is more expressive (not just “AND of exact match”)
2. you can indicate that the rule is “soft”/“preference” rather than a hard requirement, so if the scheduler can’t satisfy it, the pod will still be scheduled
3. you can constrain against labels on other pods running on the node (or other topological domain), rather than against labels on the node itself, which allows rules about which pods can and cannot be co-located

K8S Labels selector

Affinity and anti-affinity

The affinity feature consists of two types of affinity, “**node affinity**” and “**inter-pod affinity/anti-affinity**”.

- Node affinity is like the existing nodeSelector (but with the first two benefits listed in previous slide)
- inter-pod affinity/anti-affinity constraints against pod labels rather than node labels, as described in the third item listed in previous slide. in addition to having the first and second properties listed above. **nodeSelector** continues to work as usual, **but will eventually be deprecated**, as node affinity can express everything that nodeSelector can express.

K8S Node Affinity (BETA)

Affinity and anti-affinity

- it allows you to constrain which nodes your pod is eligible to be scheduled on, based on labels on the node.
- There are currently two types of node affinity, called `requiredDuringSchedulingIgnoredDuringExecution` and `preferredDuringSchedulingIgnoredDuringExecution`. You can think of them as “hard” and “soft” respectively, in the sense that the former specifies rules that *must* be met for a pod to be scheduled onto a node (just like `nodeSelector` but using a more expressive syntax), while the latter specifies *preferences* that the scheduler will try to enforce but will not guarantee. The “IgnoredDuringExecution” part of the names means that, similar to how `nodeSelector` works, if labels on a node change at runtime such that the affinity rules on a pod are no longer met, the pod will still continue to run on the node.

K8S: Moving on - Part 1

K8S Node Affinity (BETA)

Affinity and anti-affinity

This node affinity rule says the pod can only be placed on a node **with a label whose key is `kubernetes.io/e2e-az-name`** and whose value is **either `e2e-az1` or `e2e-az2`**. In addition, among nodes that meet that criteria, nodes with a label whose key is `another-node-label-key` and whose value is `another-node-label-value` should be **preferred**

```
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: kubernetes.io/e2e-az-name
                operator: In
                values:
                  - e2e-az1
                  - e2e-az2
            preferredDuringSchedulingIgnoredDuringExecution:
              - weight: 1
                preference:
                  matchExpressions:
                    - key: another-node-label-key
                      operator: In
                      values:
                        - another-node-label-value
  containers:
    - name: with-node-affinity
      image: k8s.gcr.io/pause:2.0
```

K8S: Moving on - Part 1

K8S Inter-pod (BETA)

Affinity and anti-affinity

The rules are of the form “this pod should (or, in the case of anti-affinity, should not) run in an X if that X is already running one or more pods that meet rule Y”. Y is expressed as a LabelSelector with an associated list of namespaces

A More Practical Use-cases Interpod Affinity and AntiAffinity can be even more useful when they are used with higher level collections such as ReplicaSets, Statefulsets, Deployments, etc. One can easily configure that a set of workloads should be co-located in the same defined topology, eg., the same node

K8S: Moving on - Part 1

K8S Inter-pod (BETA)

EXAMPLE

In a three node cluster, a web application has in-memory cache such as redis. We want the web-servers to be co-located with the cache as much as possible. Here is the yaml snippet of a simple redis deployment with three replicas and selector label app=store. The deployment has **PodAntiAffinity** configured to ensure the scheduler does not co-locate replicas on a single node.

K8S: Moving on - Part 1

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-cache
spec:
  selector:
    matchLabels: #install only on app:store
      app: store
  replicas: 3
  template:
    metadata:
      labels:
        app: store
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: app
                    operator: In
                    values:
                      - store
              topologyKey: "kubernetes.io/hostname"
      containers:
        - name: redis-server
          image: redis:3.2-alpine
```

Deploying 3 cache Pods on 3 of the nodes where the label selector: app=store.
Using PodAntiAffinity the scheduler will also not colocate replicas on single node



node-1	node-2	node-3
cache-1	cache-2	cache-3

K8S Inter-pod (BETA)

CO Hosting Node base on Affinity / Anti-Affinity

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-server
spec:
  selector:
    matchLabels:
      app: web-store
  replicas: 3
  template:
    metadata:
      labels:
        app: web-store
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: app
                    operator: In
                    values:
                      - web-store
              topologyKey: "kubernetes.io/hostname"
            podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: app
                    operator: In
                    values:
                      - store
              topologyKey: "kubernetes.io/hostname"
      containers:
        - name: web-app
          image: nginx:1.12-alpine
  
```



Base on the left Deployment description, we will get 3 Pods on 3 nodes with Redis due to app=store label

node-1	node-2	node-3
webserver-1	webserver-2	webserver-3
cache-1	cache-2	cache-3



K8S

DEPLOYMENTS

K8S: Moving on - Part 1

K8S DEPLOYMENTS

A Deployment object defines a Pod creation template and a desired **replica count**. The Deployment uses a label selector to identify the Pods it manages, and will create or delete Pods as needed to meet the replica count. Deployments are also used to manage safely rolling out changes to your running Pods.

K8S: Moving on - Part 1

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template: # create pods using pod definition in this template
    metadata:
      # unlike pod-nginx.yaml, the name is not included in the meta data as a unique name is
      # generated from the deployment name
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
```

K8S: Moving on - Part 1

K8S Deployments

Create new deployment

```
kubectl create -f https://raw.githubusercontent.com/yanivomc/seminars/master/K8S/code_examples/walkthrough/deployment.yaml
# Get deployment
kubectl get deployment

# List the pods created by the Deployment
kubectl get pods -l app=nginx -o wide
```

Next : Lets upgrade our nginx image from 1.7.9 to 1.9 and run apply

```
kubectl apply -f https://raw.githubusercontent.com/yanivomc/seminars/master/K8S/code_examples/walkthrough/deployment.yaml
# Get deployment
kubectl get deployment
# See rollout status of the deployment
kubectl rollout status deployment/nginx-deployment
# See rollout versions
kubectl rollout history deployment/nginx-deployment
# Rollback to specific revision
kubectl rollout undo deployment/nginx-deployment --to-revision=2
```

K8S: Moving on - Part 1

K8S Deployments

Scale deployments

```
kubectl scale deployment nginx-deployment --replicas=10  
# Delete the deployment  
kubectl delete deployment nginx-deployment
```



K8S

SERVICES

K8S: Moving on - Part 1

K8S Services

While each Pod gets its own IP address, even those IP addresses cannot be relied upon to be stable over time. This leads to a problem: if some set of Pods (let's call them backends) provides functionality to other Pods (let's call them frontends) inside the Kubernetes cluster, how do those frontends find out and keep track of which backends are in that set? - **SERVICES**

K8S Service is an abstraction which defines a logical set of Pods and a policy by which to access them - sometimes called a micro-service. The set of Pods targeted by a Service is (usually) determined by a Label Selector

For Kubernetes-native applications, Kubernetes offers a simple Endpoints API that is updated whenever the set of Pods in a Service changes. For non-native applications, Kubernetes offers a virtual-IP-based bridge to Services which redirects to the backend Pods.

K8S: Moving on - Part 1

K8S Services

Defining a service

While each Pod gets its own IP address, even those IP addresses cannot be relied upon to be stable over time. This leads to a problem: if some set of Pods (let's call them backends) provides functionality to other Pods (let's call them frontends) inside the Kubernetes cluster, how do those frontends find out and keep track of which backends are in that set? - **SERVICES**

```
kind: Service          # Create new service
apiVersion: v1
metadata:
  name: my-service # Name of the service
spec:
  selector:
    app: MyApp      # Run on any pod with label - app:myapp
  ports:            # Map incoming port to target port
    - name: http    # Setting multiple ports listeners
      protocol: TCP
      port: 80
      targetPort: 9376
    - name: https
      protocol: TCP
      port: 443
      targetPort: 9388
```

K8S: Moving on - Part 1

K8S Services: External Name

TIP: Good to know for Migration phase between Physical to K8S

There are cases where we wish to point our K8S application to an external resource such as an external DB.

With Services type: **ExternalName Service** we can let pods looking for hosts ex. my-service.prod.svc.CLUSTER will get a result of a **CNAME** we configure in our service as followed:

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
  namespace: prod
spec:
  type: ExternalName
  externalName: my.database.example.com
```

K8S: Moving on - Part 1

K8S Services

Reminder and follow up

Every node in a Kubernetes cluster runs a **kube-proxy**. **kube-proxy is responsible for implementing a form of virtual IP** for Services of type other than ExternalName. In Kubernetes v1.0, Services are a “layer 4” (TCP/UDP over IP) construct, the proxy was purely in userspace. In Kubernetes v1.1, the Ingress API was added (beta) to represent “layer 7”(HTTP) services, iptables proxy was added too, and become the default operating mode since Kubernetes v1.2. In Kubernetes v1.8.0-beta.0, ipvs proxy was added.

K8S: Moving on - Part 1

K8S Services

PROXY-MODE:Userspace & IPTables

In userspace,

kube-proxy watches the Kubernetes master for the addition and removal of Service and Endpoints objects. For each Service it opens a port (randomly chosen) on the local node. Any connections to this “proxy port” will be proxied to one of the Service’s backend Pods. Which backend Pod to use is decided based on the ****SessionAffinity** of the Service. Lastly, it installs iptables rules which capture traffic to the Service’s clusterIP (which is virtual) and Port and redirects that traffic to the proxy port which proxies the backend Pod. By default, **the choice of backend is round robin** → **BUT that’s not good enough** as due to the **session affinity and due to cases where all requests sees the same IP**. the service directs the requests from all clients to the same pod since it is unable to differentiate between the clients. This scenario will overload a pod and it would problems for the running application.

Solution: Direct LB to the pods without LB using service (for now and until ingres is fully production)

What is session affinity?

Session affinity is a feature that the requests from the same client always get routed back to the same server within a cluster of servers.

K8S: Moving on - Part 1

K8S Services

So is it a LB ?

Balance or Distribution?

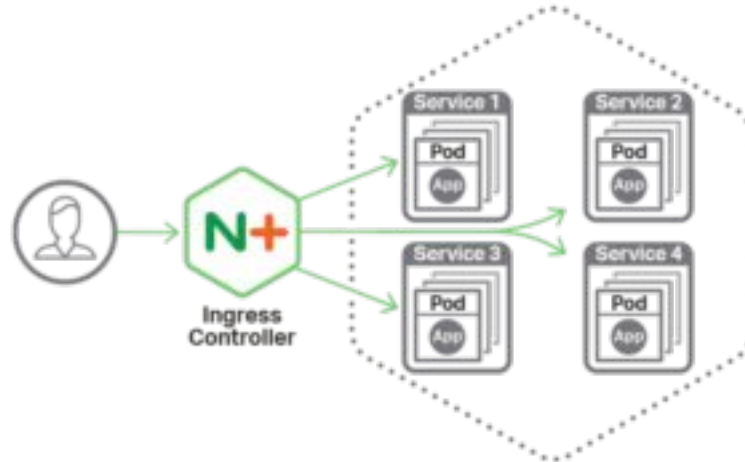
Technically, both of these methods services uses (round-robin and random) qualify as load distribution, rather than load balancing, **since they simply assign requests to an available pod without taking into account the actual load currently placed on pods represented by the service**. Any balancing beyond these distribution methods needs to be handled by other processes, which, in the case of Kubernetes, means external resources.

K8S: Moving on - Part 1

K8S Services

Solution: Ingress Controllers

Ingress can be configured to give services externally-reachable URLs, load balance traffic, terminate SSL, offer name based virtual hosting, and more. Users request ingress by POSTing the Ingress resource to the API server



K8S: Moving on - Part 1

K8S Services

Solution: HA PROXY / NGINX Ingress Controllers

The solution is to directly load balance to the pods without load balancing the traffic to the service first. This functionality is implemented with ingress-controllers in kubernetes.

Supported controllers:

<https://github.com/kubernetes/ingress-nginx/blob/master/docs/catalog.md>

```
internet
|
-----
[ Services ]

internet
|
[ Ingress ]
--|-----|--
[ Services ]
```


K8S: Moving on - Part 1

K8S Services

Summary: External Access (outside of our cluster)

K8S **ServiceTypes** allow us to specify what kind of service we want while The default is ClusterIP

- **ClusterIP**: Exposes the service on a cluster-internal IP. Choosing this value makes the service only reachable from within the cluster. This is the default **ServiceType**.
- **NodePort**: Exposes the service on each Node's IP at a static port (the **NodePort**). A **ClusterIP** service, to which the **NodePort** service will route, is automatically created. You'll be able to contact the **NodePort** service, from outside the cluster, by requesting **<NodeIP> : <NodePort>**.
- **LoadBalancer**: Exposes the service externally using a cloud provider's load balancer. **NodePort** and **ClusterIP** services, to which the external load balancer will route, are automatically created.
- **ExternalName**: Maps the service to the contents of the **externalName** field (e.g. **foo.bar.example.com**), by returning a **CNAME** record with its value. No proxying of any kind is set up. This requires version 1.7 or higher of **kube-dns**



K8S

POD HEALTHCHECK

K8S: Moving on - Part 1

K8S Services: POD HEALTH CHECK

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-http-healthcheck
spec:
  containers:
  - name: nginx
    image: nginx
    # defines the health checking
    livenessProbe:
      # an http probe
      httpGet:
        path: /_status/healthz
        port: 80
      # length of time to wait for a pod to initialize
      # after pod startup, before applying health checking
      initialDelaySeconds: 30
      timeoutSeconds: 1
    ports:
    - containerPort: 80
```



K8S

DEPLOYING STATEFUL APPLICATION

K8S: Moving on - Part 2

K8S StateFul application

Deploying stateful application

Objective:

- Create persistent volume for our DB
- Create MySql Deployment
- Expose MySql to other pods in the cluster at a known DNS Name

K8S: Moving on - Part 2

K8S Stateful application

Deploying stateful application : Deploy MYSQL

The Yaml will work on will produce a MySql DB with a presistentVolume

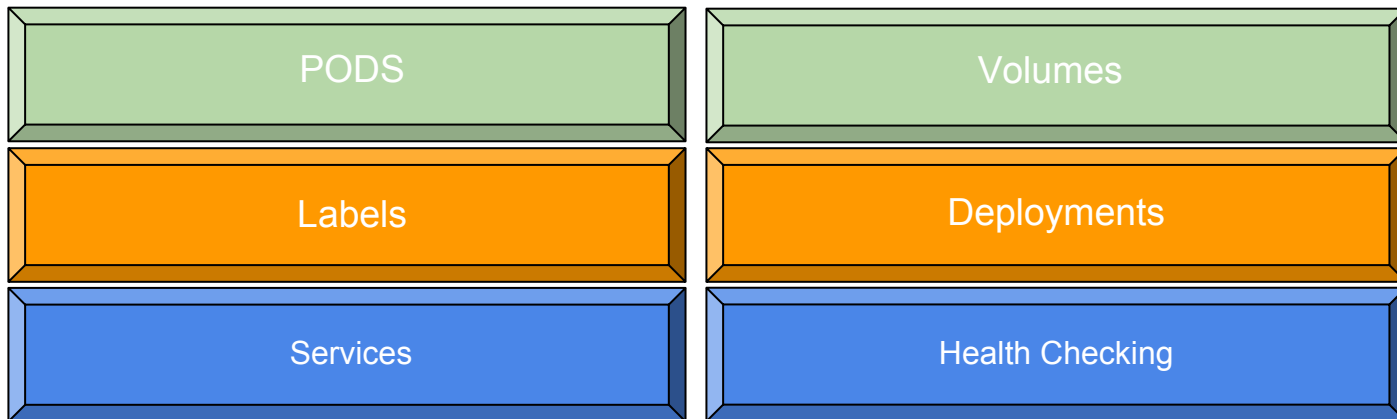
See code in `./seminars/K8S/code_examples/statefulApplication`

1. Deploy the YAML
`kubect1 create -f mysql-deployment.yml`
2. Describe deployment
`kubect1 describe deployment mysql`
3. List pods created by Deployment
`kubect1 get pods -l app=mysql`
4. Inspect Persistent Volume Claim
`kubect1 describe pvc mysql-pv-claim`
5. Test connectivity by a mysql sidecar client
`kubect1 run -it --rm --image=mysql:5.6 --restart=Never mysql-client -- mysql -h mysql -ppassword`
If the above is able to connect - we are good

K8S: Moving on

Moving On: Part 3

CI / CD With K8S



END

DevOps Course

JOHN BRYCE
Leading in IT Education
matrix company