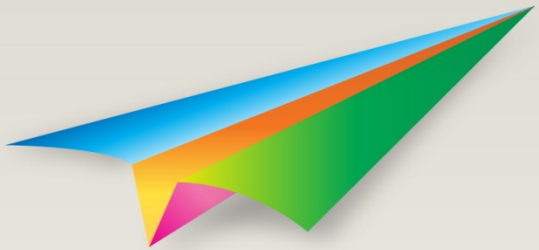

Kubernetes

trainologic

Kubernetes

- Kubernetes provides a logical abstraction of treating your data-center as a single computer.
- It allows for deploying, provisioning and self-healing of container groups (aka pods) across your cluster.

- Let's first be familiar with containers…

The Problem Domain

- You want to deploy many services/apps/micro-services.
- And you'd like the following features:
  - Isolation (e.g.: OS, resources, networking).
  - Scalability (distributed systems).
  - Evolution (upgrades/downgrades).
- So want are your options?
- What about classic VMs?

Classic VMs

- A VM provides:
  - Full isolation.
  - Evolution.
  - Distribution.
- But, the downside:
  - Heavy on resources.
  - Takes time to start.

Docker

- Docker provides:
  - Decent isolation.
  - Evolution.
  - Distribution (via Docker Swarm/ Kubernetes).
  - Fast start time.
  - Share resources (for similar images).
- The downside:
  - Only Linux.

Docker

- Docker is a lightweight container.
- Useful for deploying and running an application/service/micro-service with its environment.
- You can run your packaged application on production server, staging server, development machine or even a laptop.

- How does Docker works?

Linux Namespaces

- Docker leverages the Linux namespaces feature.
- Linux namespaces that Docker uses:
  - pid – process id numbering.
  - net – the network stack (including loopback).
  - ipc – Inter-Procces-Communication (shared memory, semaphores).
  - mnt – mounting.
  - uts – hostname.

Linux cgroups

- Docker also leverages the *cgroups* technology which allows to share and limit resources between containers.

- And because Linux kernel API is backward compatible, you can run **any Linux on any Linux** with Docker!

The Filesystem

- Docker uses Union FS which is layered.
- This means that a Docker image will only contain the difference from the parent image.
- You can deploy thousands of containers from the same image without "almost" any additional cost.

The Participants

- The following concepts participate in Docker architecture:
  - The Docker daemon.
  - The Docker client.
  - Docker images.
  - Docker registries.
  - Docker containers.

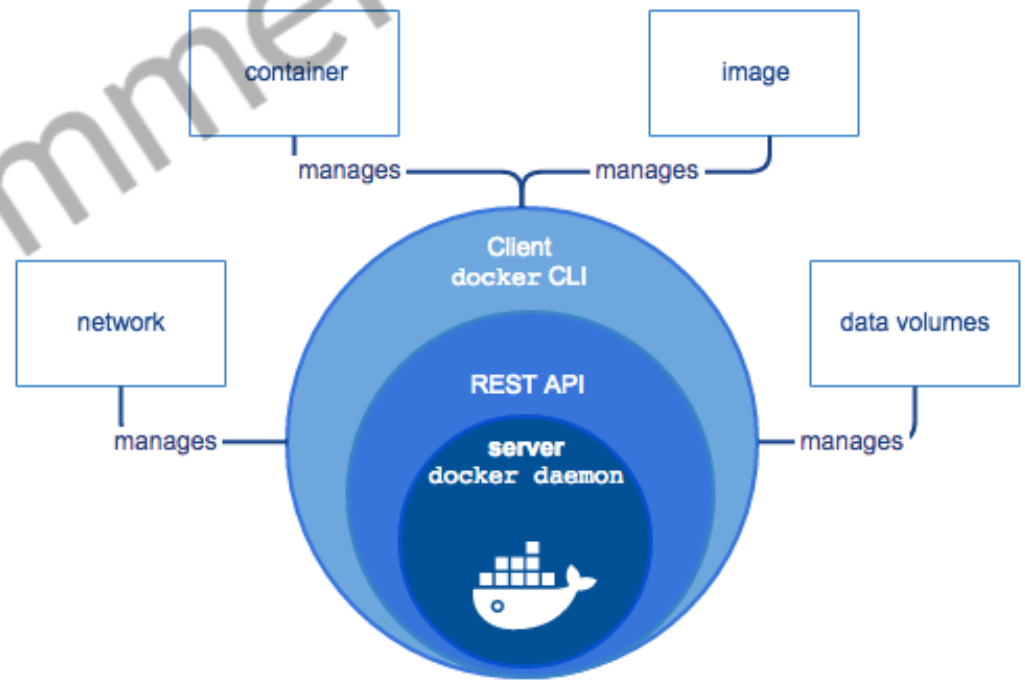

Image taken from docs.docker.com

Docker architecture

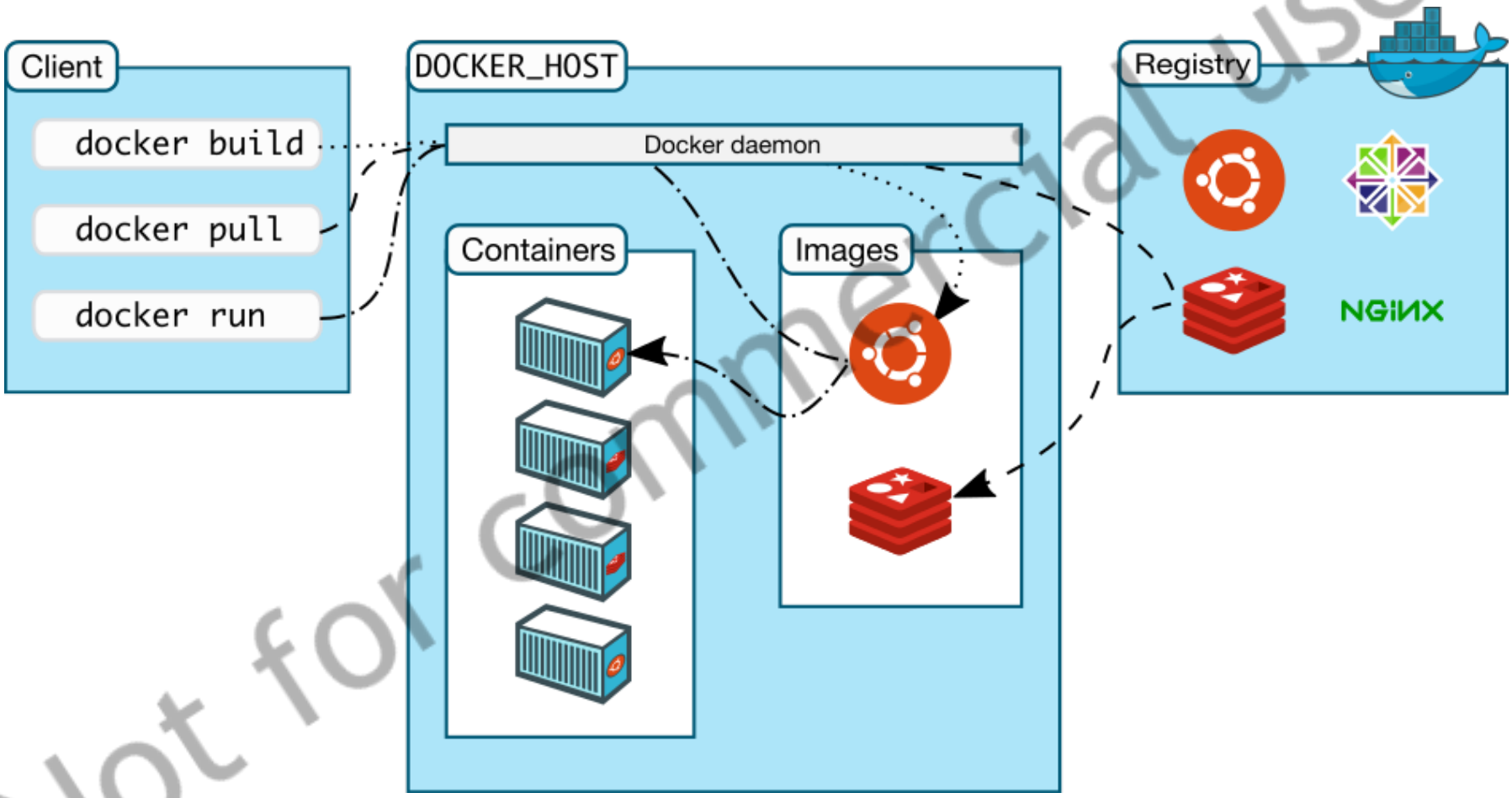

Image taken from docs.docker.com

Docker Daemon and Client

- The host process of Docker.
- Users interact with the daemon through the Docker client binary.
- The client can connect from a remote machine.
- The daemon manages the images and the containers.

Docker Image

- Based on a "Dockerfile" file
- "Dockerfile" contains instructions on how to build the image.
- The image itself should reside in a Docker registry.
- Read only template that is used to instantiate containers
- Composed of layers – Each "Dockerfile" instruction is a new layer
- Based on the UFS – Union File System

Docker Registry

- Manages Docker images.
- The main public one is: Docker Hub.
- You can create your own private registry.
- Main point of distribution.

Docker Container

- A runnable instance of the Docker image
- Can be ran, start, stopped, moved or deleted
- Secure and isolated application platform
- Can be given access to resources on other hosts

Running Containers

- docker run [options] image [command] [args]
- More than 80 options! The most useful:
  - **--name** – gives the container a name
  - **--rm** – removes the container at the end of the
  - **-d** – run container in the background
  - **-it** – interactive mode
  - **-p** [host port:container port]
  - **-v** [volume definition] – will be discussed later on
  - **--network** will also be discussed later on

Managing Containers

- You can list the current containers with the following instruction: *docker **ps***
  - Will list the running instances.
  - Use the –a argument to list all the containers (including stopped ones).

- To stop/start a container :
  - *docker **start/stop** [container name]*
- To remove a container: *docker **rm** [container name]*

Debugging Containers

- Connect to a running container:
  - docker **attach** container – connects to a running container
  - docker **exec** –it container /bin/bash – opens bash shell on the specified container

Dockerfile

- A Docker image is built from a Dockerfile.
- Should reside at SCM
- Syntax:
  - Comments are lines starting with '#'.
  - Every other line is in the format: 'instruction arguments'.
- Let's see the available instructions…

Instructions

- FROM – the first instruction in the file which specifies the base image to build upon.
- MAINTAINER – the author of the image.
- RUN <command> -- runs a command in shell form.
- RUN ["exec", "arg1", ...] – runs a command in exec form.
- CMD – discussed later.
- LABEL key=value key=value – adds metadata to the image.

Instructions

- ADD – adds file(s) from the context to the image. You can use regex and even urls.
- COPY – same as ADD but without URL and tar handling.
- ENTRYPOINT – discussed later.
- VOLUME "/dir" – creates a mount point for externally mounted volumes.
- USER – sets the user for the next instructions.
- WORKDIR – sets the working directory for the next instructions.

ENTRYPOINT / CMD

- Will define an executable to run when running the container.
- Any arguments to the 'docker run' command will be appended to the entrypoint.
- There can be only one entrypoint in the dockerfile.
- You can use CMD once in your dockerfile to provide default arguments for the entrypoint.

Volumes

- A specially designed directory that bypasses the UFS
- Meaning changes are written directly
- Used for persistent data (DB, files) or to share data between containers
- docker **run/create** –v [[host path]:][docker path] [image name]
  - docker run –v /dbdata postgresql
  - docker run –v /host_path:/docker_path postgresql

Managing Volumes

- To view all the available volumes use :
  docker volume ls
- To remove unnecessary volumes use:
  docker volume rm [volume name]
- More useful to remove volumes that belong to a specific container:
  docker rm –v [container id or alias]

Data Containers

- To share data between containers use data container
- Creating data container:
  - *docker **create -v /dbdata** --name dbstore training/postgres /bin/true*
- Using it in other containers:
  - *docker **run -d --volumes-from dbstore** --name db1 training/postgres*
  - Loads all the volumes from the dbstore container

Building Images

- docker **build** [options] PATH | URL
- Builds the Docker image from a "Dockerfile" and a the files located in the path/URL
- Searches for "Dockerfile" file by default or use –f [other docker file] option to specify the Dockerfile

- The URL can point to:
  - GIT repository
  - Plaintext file
  - Tarball context

Tagging Images

- Every image has a unique id given during the build
- It can also be tagged: [Repository[:tag]
  - Repository is used to group similar images together
  - Tag is used to differ images in the same repository, usually used for versioning
  - **Repository != Registry**
- docker build –t repo:tag1 repo:latest .

Managing Networks

- You can create private or bridge networks between containers.
- By default all the containers are on the same bridge network – can access each over by ip
- You can also use the legacy --link to make containers aware of their peer hostname.

- Use 'docker create network' command to create networks

Managing Networks

- User-defined networks can be either:
  - bridge – single host network, every container sees the other containers by host name
  - overlay – similar to bridge but across multiple hosts in docker swarm

Docker Compose

- docker-compose is a tool for managing a group of containers on a single host.
- You can use it with docker swarm for multi-node environments.

- docker-comose revolves around the docker-compose.yml.

docker-compose.yml

- Example:

```
consul:
  command: -server -bootstrap -advertise 10.0.2.15
  image: progrium/consul
  restart: unless-stopped
  ports:
  - "8300:8300"
  - "8400:8400"
db:
  image: mysql
  environment:
    MYSQL_ROOT_PASSWORD: root
  volumes_from:
   - sqldata
```

Docker Compose Commands

- In the docker-compose yml folder:
  - up – creates and starts the containers
  - stop – stops the containers
  - kill – kills the containers
  - use docker-compose –-help for more commands

Back to Kubernetes

- Kubernetes main concepts:
  - Pods – logical group of containers.
  - Labels – metadata attached to objects.
  - Replica Sets – Used mainly by deployments to provide pod orchestration.
  - Deployments – Declarative abstraction over pods and Replica Sets.
  - kubectl – the CLI tool for interacting with Kubernetes Cluster Manager.

kubectl

- exec – executes a command against a specific container in a pod.
- describe – get detailed information about a resource.
- delete – delete resources.
- explain – get documentation about a resource.
- get – list resources.
- label – add or update meta-data.
- logs – get container logs.
- run – run an image in the cluster.

kubectl

- create– create resources from a deployment file.
- autoscale – configures auto-scaling of pods.
- version – well, you know.

Labels

- Key-value meta-data that can be attached to various objects in kubernetes.
- They are not unique like names and UIDs.
- You can use selectors to filter objects according to labels.

Pods

- A Pod is a group of co-located containers.
- They run on the same node and share they same linux namespaces.
- Life-cycle of a pod consists of the following phases:
  - Pending
  - Running
  - Succeeded
  - Failed
  - Unknown

Probes

- You can define diagnostic probes for pods.
  - Liveness Probe – whether the container is alive (failures will be handled by RestartPolicy).
  - Readiness Probe – whether a container is ready to serve requests. Initial state is failed.

ImagePullPolicy

- When a container is started in a pod, this property controls whether to check a remote registry for a new image.
- Values: Always and IfNotPresent.
- The default is IfNotPresent or Always (for :latest tag).
- It is advised not to use *latest*.

RestartPolicy

- You can define a RestartPolicy for every pod.
- Can be set to:
  - Always – the default value.
  - OnFailure – only if a container exited in a failed status.
  - Never.
- A restart will have exponential backoff and cupped at 5 minutes.

Controllers

- A pod will never be moved to a different node once it was started on one.
- You need some kind of **controller** to do it.
- Controller types in Kubernetes:
  - Job – for pods which are expected to terminate.
  - Deployment – for pods which are not expected to terminate.
  - DaemonSet – for pods that must have a single instance per machine.

Jobs

- A Job creates one or more pods and makes sure that they terminate successfully.
- A typical scenario is to run a simple Job to make sure that a single pod completes successfully (even in the face of failures and deletion).
- To list successful pod names of a job use:

kubectl get pods --selector=job-name=XXX –output=jsonpath={.item..metadata.name}

Jobs

- Pods in a Job template are not allowed to have a RestartPolicy of Always.
- If a Job fails, it will be restarted forever by default.
- You can set the *activeDeadlineSeconds* property of the *spec* to specify a deadline.
- After the deadline, no more pods will be created and existing pods will be deleted.

Deployments

- Deployments allow you to manage pods and replica sets.
- In a declarative way!
- When creating a deployment, it is advisable to specify

  --*record*
- Allows to view the desired state of your replicas and pods.
- The name of the replica-set is *deploymentName-podTemplateHash*.

Deployments

- You can view the status of a deployment by:

  kubectl rollout status deploymentName

- A rollout will happen only if you change the deployment's pod template.
- For example, scaling the deployment will not trigger a rollout.

Deployments

- By default, a deployment will make sure that at most one pod is unavailable during update. (1 max unavailable).
- It will also ensure that (by default) at most one pod can be created more than the desired count (1 max surge).
- When you update a deployment, a new replica-set is created to scale up the new pods and the old replica-set (which has the same selector) will be scaled-down.

Node Selectors

- You can provide node selectors for pods.
- According to labels of-course.
- There are built-in labels for nodes (e.g., hostname).
- For example, you can select only nodes with “large memory” or “ssd disks”.
- Note that usually, there is no need for this as kubernetes will manage resources quite well on its own.

Rollback

- A rollback will affect only the pod template.
- You can list the deployment's revisions by

  kubectl rollout history deployment/NAME

- You can provide the --revision=N to see revision's details.
- Rolling-back is done with:

  kubectl rollout undo deployment/NAME --to-revision=N

Pause and Resume

- You can pause a deployment with:

  kubectl rollout pause deployment/NAME

- Resume it with *resume*.
- You can use it for canary deployments.

Services

- Services provides an abstraction over a logical set of Pods.
- Somewhat analogous to a micro-service.
- Usually exposes selector-based pods.
- Can export a port connected to pods' target-port which may even be a string (a name of a port inside).
- Allows for great flexibility.
- Supports UDP and TCP.

Services

- Services without selectors can be used to map access to external non-kubernetes applications.
- Be sure to define an Endpoint for those applications.
- You can also define an ExternalName service that will alias an external service outside of the cluster at the DNS level.

Virtual IPs

- Each node in kubernetes runs a kube-proxy application that provides virtual-ip handling for services.
- The default proxy is iptables-based.
- The proxy watches the kubernetes master and for each service it install rules in the iptable to redirect traffic to that service.
- By default the backend pod is randomly selected.
- You can set *sesionAffinity* to “ClientIP”.

Virtual IPs

- In proxy-mode "iptables" if a pod has failed, the client will not automatically be connected to a new one (like in "userspace" mode).
- It relies on a well-defined readiness-probes.
- If you want you can specify a clusterIP address for your service (must be inside your service-cluster-ip-range).

Finding Services

- A service can be found by environment variables and a DNS server (recommended).
- For env variables, kubelet adds entries compatible to docker links and also simple vars:
  - SVCNAME_SERVICE_HOST and …_PORT.
- Be-aware that a pod using these envs, must be created after the service.

DNS Server

- A cluster add-on that creates DNS records for each service.
- Does not suffer from the 'envs' ordering problem.
- You can define headless services (specify 'None' in cluserIP) and you'll get only DNS (discovery) support and not proxy and load-balancing services.

Service Types

- The following ServiceType values are supported:
  - ClusterIP – cluster only internal IP (the default).
  - ExternalName – maps to an external host without proxying.
  - NodePort – in addition to internal cluster IP, expose the service on each node of the cluster (same port for all nodes).
  - LoadBalancer – in addition to NodePort, asks the cloud provider for a load-balancing service.

Secrets

- Kubernetes provide a Secret object for managing sensitive information (e.g., ssh keys, OAuth tokens).
- Pods can be provided with volumes (must be readOnly) that contain the secrets' file.
- Secrets should be base64 encoded.
- Pods can also consume secrets as environment variables.

ConfigMaps

- ConfigMaps are like Secrets.
- However, they are not intended for sensitive information.
- Probably there will be differences in future versions.