Monolithic Architecture

What is Monolithic Architecture ?

- A monolithic architecture is a traditional model of a software program, which is built as a unified unit that is self-contained and independent from other applications
- The word “monolith” is often attributed to something large and glacial, which isn’t far from the truth of a monolith architecture for software design
- A monolithic architecture is a singular, large computing network with one code base that couples all of the business concerns together
- To make a change to this sort of application requires updating the entire stack by accessing the code base and building and deploying an updated version of the service-side interface
- Monoliths can be convenient early on in a project's life for ease of code management, cognitive overhead, and deployment
- This allows everything in the monolith to be released at once.

Pros

- Easy deployment
  - One executable file or directory makes deployment easier
- Development
  - When an application is built with one code base, it is easier to develop
- Performance
  - In a centralized code base and repository, one API can often perform the same function that numerous APIs perform with microservices
- Simplified testing
  - Since a monolithic application is a single, centralized unit, end-to-end testing can be performed faster than with a distributed application
- Easy debugging
  - With all code located in one place, it's easier to follow a request and find an issue.

Cons

- Slower development speed
  - A large, monolithic application makes development more complex and slower
- Scalability
  - You can’t scale individual components
- Reliability
  - If there’s an error in any module, it could affect the entire application’s availability
- Barrier to technology adoption
  - Any changes in the framework or language affects the entire application, making changes often expensive and time-consuming
- Lack of flexibility
  - A monolith is constrained by the technologies already used in the monolith
- Deployment
  - A small change to a monolithic application requires the redeployment of the entire monolith

Microservices

What is Microservices Architecture ?

- A microservices architecture, also simply known as microservices, is an architectural method that relies on a series of independently deployable services
- These services have their own business logic and database with a specific goal
- Updating, testing, deployment, and scaling occur within each service
- Microservices decouple major business, domain-specific concerns into separate, independent code bases
- Microservices don’t reduce complexity, but they make any complexity visible and more manageable by separating tasks into smaller processes that function independently of each other and contribute to the overall whole
- Adopting microservices often goes hand in hand with DevOps, since they are the basis for continuous delivery practices that allow teams to adapt quickly to user requirements

Pros

- Agility
  - Promote agile ways of working with small teams that deploy frequently
- Flexible scaling
  - If a microservice reaches its load capacity, new instances of that service can rapidly be deployed to the accompanying cluster to help relieve pressure
  - We are now multi-tenant and stateless with customers spread across multiple instances. Now we can support much larger instance sizes
- Continuous deployment
  - We now have frequent and faster release cycles. Before we would push out updates once a week and now we can do so about two to three times a day.
- Highly maintainable and testable
  - Teams can experiment with new features and roll back if something doesn't work
  - This makes it easier to update code and accelerates time-to-market for new features. Plus, it is easy to isolate and fix faults and bugs in individual services
- Independently deployable
  - Since microservices are individual units they allow for fast and easy independent deployment of individual features.
- Technology flexibility
  - Microservice architectures allow teams the freedom to select the tools they desire
- High reliability
  - You can deploy changes for a specific service, without the threat of bringing down the entire application
- Happier teams
  - The Atlassian teams who work with microservices are a lot happier, since they are more autonomous and can build and deploy themselves without waiting weeks for a pull request to be approved

Cons

- Development sprawl
  - These add more complexity compared to a monolith architecture, as there are more services in more places created by multiple teams
  - If development sprawl isn't properly managed, it results in slower development speed and poor operational performance
- Exponential infrastructure costs
  - Each new microservice can have its own cost for test suite, deployment playbooks, hosting infrastructure, monitoring tools, and more
- Added organizational overhead
  - Teams need to add another level of communication and collaboration to coordinate updates and interfaces
- Debugging challenges
  - Each microservice has its own set of logs, which makes debugging more complicated
  - Plus, a single business process can run across multiple machines, further complicating debugging
- Lack of standardization
  - Without a common platform, there can be a proliferation of languages, logging standards, and monitoring
- Lack of clear ownership
  - As more services are introduced, so are the number of teams running those services
  - Over time it becomes difficult to know the available services a team can leverage and who to contact for support

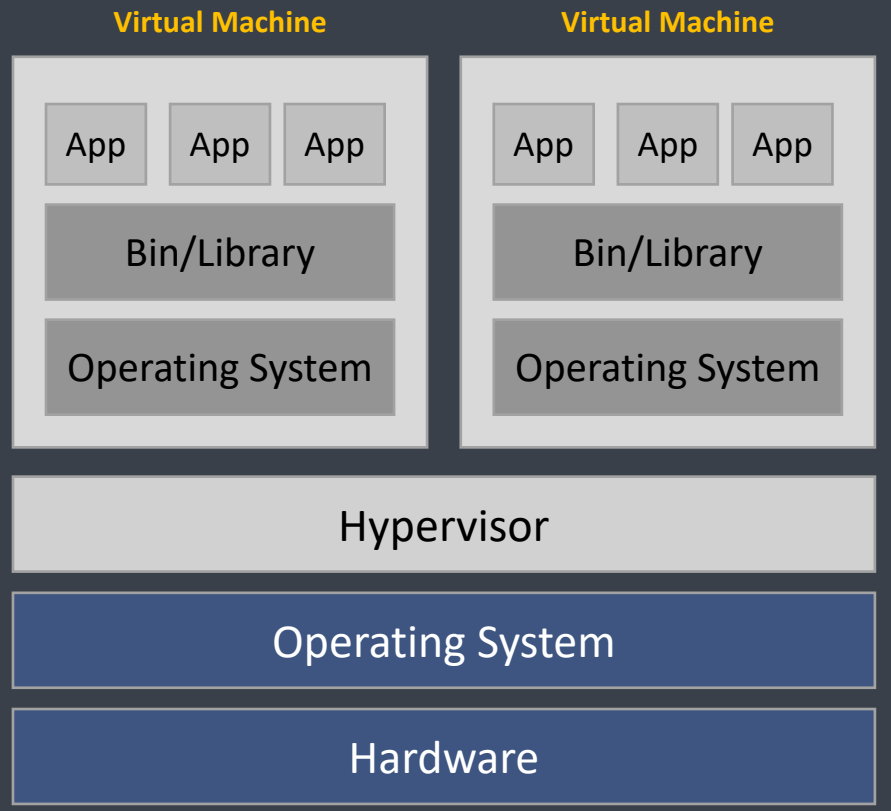
Containerization

Traditional Deployment

- Early on, organizations ran applications on physical servers
- There was no way to define resource boundaries for applications in a physical server, and this caused resource allocation issues
- For example, if multiple applications run on a physical server, there can be instances where one application would take up most of the resources, and as a result, the other applications would underperform
- A solution for this would be to run each application on a different physical server
- But this did not scale as resources were underutilized, and it was expensive for organizations to maintain many physical servers

Virtualized Deployment

- It allows you to run multiple Virtual Machines (VMs) on a single physical server's CPU
- Virtualization allows applications to be isolated between VMs and provides a level of security as the information of one application cannot be freely accessed by another application
- Virtualization allows better utilization of resources in a physical server and allows better scalability because
  - an application can be added or updated easily
  - reduces hardware costs
- With virtualization you can present a set of physical resources as a cluster of disposable virtual machines
- Each VM is a full machine running all the components, including its own operating system, on top of the virtualized hardware



What is Containerization

- Containerization is the packaging of software code with just the operating system (OS) libraries and dependencies required to run the code to create a single lightweight executable—called a container—that runs consistently on any infrastructure
- More portable and resource-efficient than virtual machines (VMs), containers have become the de facto compute units of modern cloud-native applications
- Containerization allows developers to create and deploy applications faster and more securely
- With traditional methods, code is developed in a specific computing environment which, when transferred to a new location, often results in bugs and errors
- For example, when a developer transfers code from a desktop computer to a VM or from a Linux to a Windows operating system
- Containerization eliminates this problem by bundling the application code together with the related configuration files, libraries, and dependencies required for it to run
- This single package of software or “container” is abstracted away from the host operating system, and hence, it stands alone and becomes portable—able to run across any platform or cloud, free of issues

Container deployment

- Containers are similar to VMs, but they have relaxed isolation properties to share the Operating System (OS) among the applications
- Therefore, containers are considered lightweight
- Similar to a VM, a container has its own filesystem, CPU, memory, process space, and more
- As they are decoupled from the underlying infrastructure, they are portable across clouds and OS distributions

Containerization vs Virtualization

Virtual Machine	Container
Hardware level virtualization	OS virtualization
Heavyweight (bigger in size)	Lightweight (smaller in size)
Slow provisioning	Real-time and fast provisioning
Limited Performance	Native performance
Fully isolated	Process-level isolation
More secure	Less secure
Each VM has separate OS	Each container can share OS resources
Boots in minutes	Boots in seconds
Pre-configured VMs are difficult to find and manage	Pre-built containers are readily available
Can be easily moved to new OS	Containers are destroyed and recreated
Creating VM takes longer time	Containers can be created in seconds

Benefits

- **Portability**
  - A container creates an executable package of software that is abstracted away from (not tied to or dependent upon) the host operating system, and hence, is portable and able to run uniformly and consistently across any platform or cloud
- **Agility**
  - The open source Docker Engine for running containers started the industry standard for containers with simple developer tools and a universal packaging approach that works on both Linux and Windows operating systems
  - The container ecosystem has shifted to engines managed by the Open Container Initiative (OCI)
  - Software developers can continue using agile or DevOps tools and processes for rapid application development and enhancement
- **Speed**
  - Containers are often referred to as “lightweight,” meaning they share the machine’s operating system (OS) kernel and are not bogged down with this extra overhead
  - Not only does this drive higher server efficiencies, it also reduces server and licensing costs while speeding up start-times as there is no operating system to boot
- **Fault isolation**
  - Each containerized application is isolated and operates independently of others
  - The failure of one container does not affect the continued operation of any other containers
  - Development teams can identify and correct any technical issues within one container without any downtime in other containers
  - Also, the container engine can leverage any OS security isolation techniques—such as SELinux access control—to isolate faults within containers

Benefits

- Efficiency
  - Software running in containerized environments shares the machine's OS kernel, and application layers within a container can be shared across containers
  - Thus, containers are inherently smaller in capacity than a VM and require less start-up time, allowing far more containers to run on the same compute capacity as a single VM. This drives higher server efficiencies, reducing server and licensing costs
- Ease of management
  - Container orchestration platform automates the installation, scaling, and management of containerized workloads and services
  - Container orchestration platforms can ease management tasks such as scaling containerized apps, rolling out new versions of apps, and providing monitoring, logging and debugging, among other functions. Kubernetes, perhaps the most popular container orchestration system available, is an open source technology (originally open-sourced by Google, based on their internal project called Borg) that automates Linux container functions originally
  - Kubernetes works with many container engines, such as Docker, but it also works with any container system that conforms to the Open Container Initiative (OCI) standards for container image formats and runtimes
- Security
  - The isolation of applications as containers inherently prevents the invasion of malicious code from affecting other containers or the host system
  - Additionally, security permissions can be defined to automatically block unwanted components from entering containers or limit communications with unnecessary resources

Docker

What is Docker ?

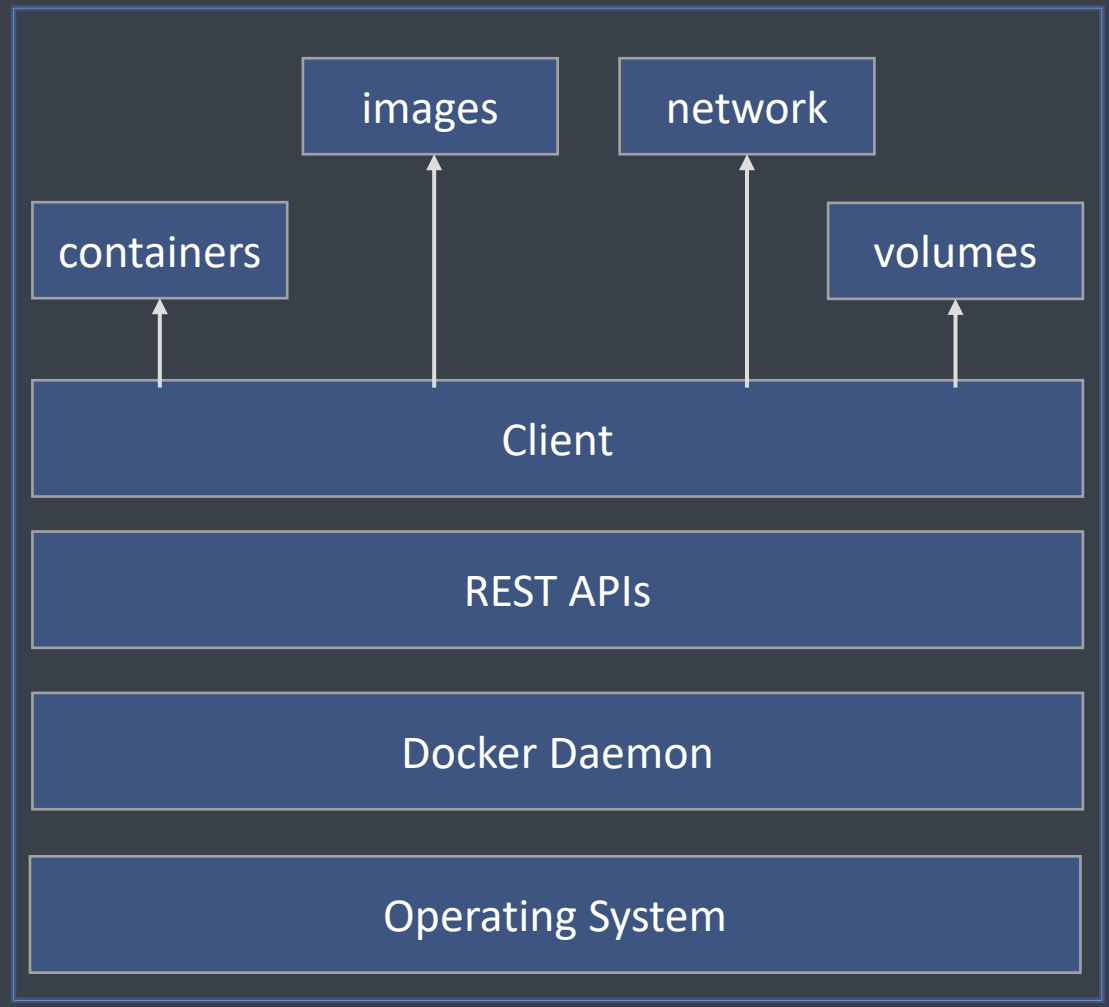
- Docker is an open source platform that enables developers to build, deploy, run, update and manage containers—standardized, executable components that combine application source code with the operating system (OS) libraries and dependencies required to run that code in any environment
- Why docker?
  - It is an easy way to create application deployable packages
  - Developer can create ready-to-run containerized applications
  - It provides consistent computing environment
  - It works equally well in on-prem as well as cloud environments
  - It is light weight compared to VM

Little history about Docker

- Docker Inc, started by Solomon Hykes, is behind the docker tool
- Docker Inc started as PasS provider called as dotCloud
- In 2013, the dotCloud became Docker Inc
- Docker Inc was using LinuX Containers (LXC) before version 0.9
- After 0.9 (2014), Docker replaced LXC with its own library libcontainer which is developed in Go programming language
- Its not the only solution for containerization
  - “FreeBSD Jails”, launched in 2000
  - LXD is next generation system container manager build on top of LXC and REST APIs
  - Google has its own open source container technology Imctfy (Let Me Contain That For You)
  - Rkt is another option for running containers

Docker Architecture

- Docker daemon (dockerd)
  - Continuous running process
  - Manages the containers
- REST APIs
  - Used to communicate with docker daemon
- Client (docker)
  - Provides command line interface
  - Used to perform all the tasks



libcontainer

- Docker has replaced LXC by libcontainer, which is used to manage the containers
- Libcontainer uses
  - Namespaces
    - Creates isolated workspace which limits what container can see
    - Provides a layer of isolation to the container
    - Each container runs in a separate namespace
    - Processes running in a namespace can interact with other processes or use resources which are the part of the same namespace
    - E.g. process ID, network, IPC, Filesystem
  - Control Groups (cgroups)
    - Used to share the available resources to the conainers
    - It optionally enforces limits and constraints on resource usage
    - It limits how much a container can use
    - E.g. CPU, Disk space, memory

libcontainer

- Union File System (UnionFS)
  - It uses layers
  - It is a lightweight and very fast FS
  - Docker uses different variants of UnionFS
    - Aufs (advanced multi-layered unification filesystem)
    - BtrFS (B-Tree FS)
    - VFS (Virtual FS)
    - Devicemapper

Docker Objects

- **Images**: read only template with instructions for creating docker containers
- **Container**: running instance of a docker image
- **Network**: network interface used to connect the containers to each other or external networks
- **Volumes**: used to persist the data generated by and used by the containers
- **Registry**: private or public collection of docker images
- **Service**: used to deploy application in a docker multi node cluster

What is Docker image ?

- A Docker image is a file used to execute code in a Docker container
- Docker images act as a set of instructions to build a Docker container, like a template
- Docker images also act as the starting point when using Docker. An image is comparable to a snapshot in virtual machine (VM) environments
- A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container
- Docker images have multiple layers, each one originates from the previous layer but is different from it
- The layers speed up Docker builds while increasing reusability and decreasing disk use
- Image layers are also read-only files
- Once a container is created, a writable layer is added on top of the unchangeable images, allowing a user to make changes

Docker Container

Docker Container

- It is a running aspect of docker image
- Contains one or more running processes
- It is a self-contained environment
- It wraps up an application into its own isolated box (application running inside a container has no knowledge of any other applications or processes that exist outside the container)
- A container can not modify the image from which it is created
- It consists of
  - Your application code
  - Dependencies
  - Networking
  - Volumes
- Containers are stored under /var/lib/docker
- This directory contains images, containers, network volumes etc

Basic Operations

- Creating container
- Starting container
- Running container
- Listing running containers
- Listing all containers
- Getting information of a container
- Stopping container
- Deleting container

Attaching a container

- There are two ways to attach to a container
- Attach
  - Used to attach the container
  - Uses only one input and output stream
  - Task
    - Attach to a running container
- Exec
  - Mainly it is used for running a command inside a container
  - Task
    - Execute a command inside container

Hostname and name of container

- To check the host name
  - Go inside the container
  - Check the hostname by using a command hostname
- Docker uses the first 12 characters of container id as hostname
- Docker automatically generates a name at random for each container

Publishing port on container

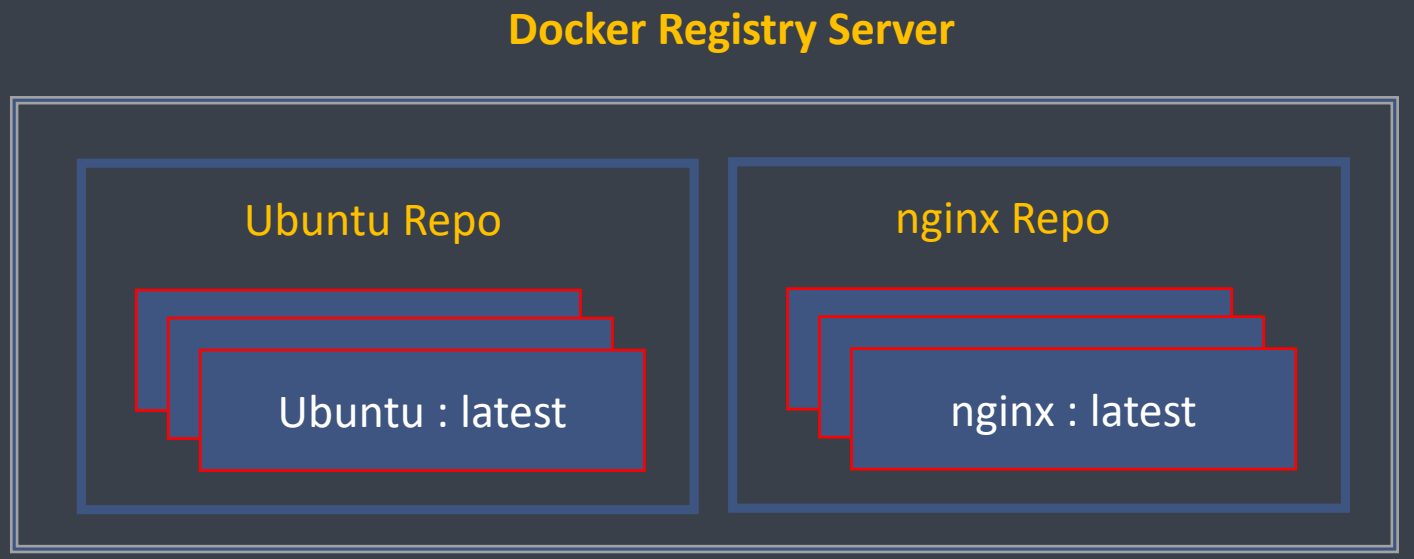
- Publishing a port is required to give an external access to your application
- Port can be published only at the time of creating a container
- You can not update the port configuration on running container
- Task
  - Run a httpd container with port 8080 published, to access apache externally

Docker Images (Advanced)

Docker Image

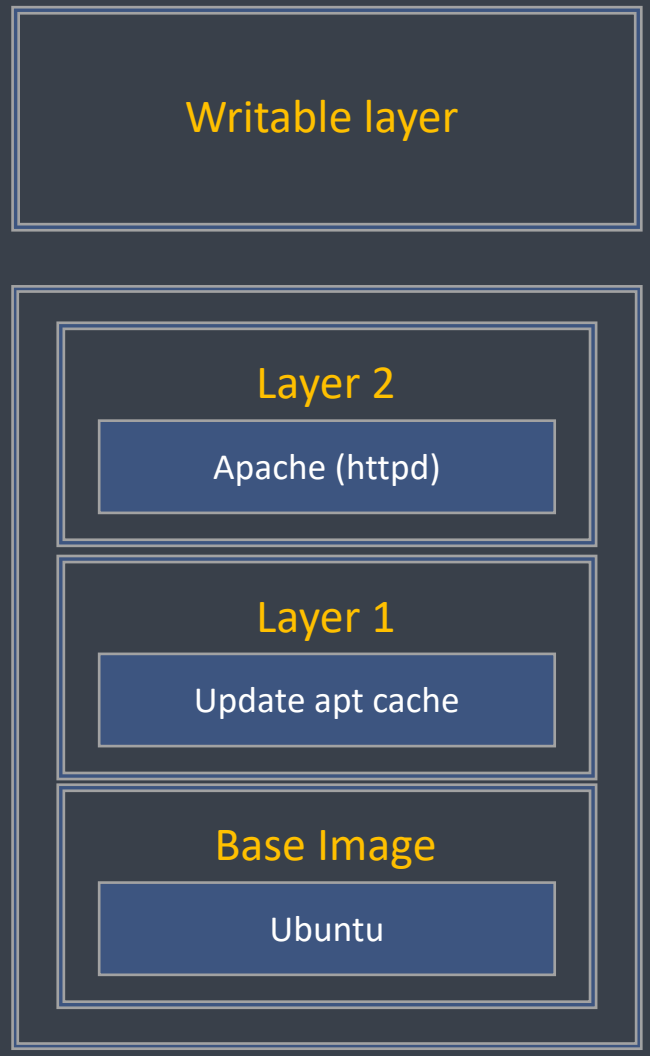
- Read-only instructions to run the containers
- It is made up of different layers
- Repositories hold images
- Docker registry stores repositories
- To create a custom image
  - Commit the running container
  - Use a Dockerfile

- Task
  - Create a container
  - Create a directory and a file within it
  - Commit the container to create a new image



Layered File System

- Docker images are made of layered FS
- Docker uses UnionFS for implementing the layered docker images
- Any update on the image adds a new layer
- All changes made to the running container are written inside a writable layer



Dockerfile

- The Dockerfile contains a series of instructions paired with arguments
- Each instruction should be in upper-case and be followed by an argument
- Instructions are processed from top to bottom
- Each instruction adds a new layer to the image and then commits the image
- Upon running, changes made by an instruction make it to the container

Dockerfile instructions

- FROM
- ENV
- RUN
- CMD
- EXPOSE
- WORKDIR
- ADD
- COPY
- LABEL
- MAINTAINER
- ENTRYPOINT

Orchestration

Container Orchestration

- Container orchestration is all about managing the lifecycles of containers, especially in large, dynamic environments
- Software teams use container orchestration to control and automate many tasks
  - Provisioning and deployment of containers
  - Redundancy and availability of containers
  - Scaling up or removing containers to spread application load evenly across host infrastructure
  - Movement of containers from one host to another if there is a shortage of resources in a host, or if a host dies
  - Allocation of resources between containers
  - External exposure of services running in a container with the outside world
  - Load balancing of service discovery between containers
  - Health monitoring of containers and hosts
  - Configuration of an application in relation to the containers running it
- Orchestration Tools
  - Docker Swarm
  - Kubernetes
  - Mesos
  - Marathon

Docker Swarm

- Docker Swarm is a container orchestration engine
- It takes multiple Docker Engines running on different hosts and lets you use them together
- The usage is simple: declare your applications as stacks of services, and let Docker handle the rest
- It is secure by default
- It is built using Swarmkit

What is a swarm?

- A swarm consists of multiple Docker hosts which run in **swarm mode**
- A given Docker host can be a manager, a worker, or perform both roles
- When you create a service, you define its optimal state
- Docker works to maintain that desired state
  - For instance, if a worker node becomes unavailable, Docker schedules that node's tasks on other nodes
- A *task* is a running container which is part of a swarm service and managed by a swarm manager, as opposed to a standalone container
- When Docker is running in swarm mode, you can still run standalone containers on any of the Docker hosts participating in the swarm, as well as swarm services
- A key difference between standalone containers and swarm services is that only swarm managers can manage a swarm, while standalone containers can be started on any daemon

Features

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

Nodes

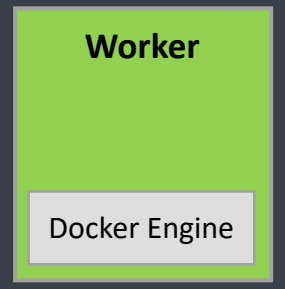
- A **node** is an instance of the Docker engine participating in the swarm
- You can run one or more nodes on a single physical computer or cloud server
- To deploy your application to a swarm, you submit a service definition to a **manager node**
- **Manager Node**
  - The manager node dispatches units of work called tasks to worker nodes
  - Manager nodes also perform the orchestration and cluster management functions required to maintain the desired state of the swarm
  - Manager nodes elect a single leader to conduct orchestration tasks
- **Worker nodes**
  - Worker nodes receive and execute tasks dispatched from manager nodes
  - An agent runs on each worker node and reports on the tasks assigned to it
  - The worker node notifies the manager node of the current state of its assigned tasks so that the manager can maintain the desired state of each worker

Services and tasks

- **Service**
  - A service is the definition of the tasks to execute on the manager or worker nodes
  - It is the central structure of the swarm system and the primary root of user interaction with the swarm
  - When you create a service, you specify which container image to use and which commands to execute inside running containers
- **Task**
  - A task carries a Docker container and the commands to run inside the container
  - It is the atomic scheduling unit of swarm
  - Manager nodes assign tasks to worker nodes according to the number of replicas set in the service scale
  - Once a task is assigned to a node, it cannot move to another node
  - It can only run on the assigned node or fail

Create a Swarm and add workers to swarm

- In our swarm we are going to use 3 nodes (one manager and two workers)
- Every node must have Docker Engine 1.12 or newer installed
- Following ports are used in node communication
  - TCP port 2377 is used for cluster management communication
  - TCP and UDP port 7946 is used for node communication
  - UDP port 4789 is used for overlay network traffic



docker swarm init --advertise-addr <ip>

docker swarm join --token <token>

Swarm Setup

- Create swarm

  > docker swarm init --advertise-addr <MANAGER-IP>

- Get current status of swarm

  > docker info

- Get the list of nodes

  > docker node ls

Swarm Setup

- Get token (on manager node)

  > docker swarm join-token worker

- Add node (on worker node)

  > docker swarm join --token <token>

Overlay Network

- It is a computer network built on top of another network
- Sits on top of the host-specific networks and allows container, connected to it, to communicate securely
- When you initialize a swarm or join a host to swarm, two networks are created
  - An overlay network called as ingress network
  - A bridge network called as docker_gwbridge
- Ingress network facilitates load balancing among services nodes
- Docker_gwbridge is a bridge network that connect overlay networks to individual docker daemon’s physical network

Service

- Definition of tasks to execute on Manager or Worker nodes
- Declarative Model for Services
- Scaling
- Desired state reconciliation
- Service discovery
- Rolling updates
- Load balancing
- Internal DNS component

Swarm Service

- Deploy a service

  > docker service create --replicas <no> --name <name> -p <ports> <image> <command>

- Get running services

  > docker service ls

- Inspect service

  > docker service inspect <service>

- Get the nodes running service

  > docker service ps <service>

Swarm Service

- Scale service

  > docker service scale <service>=<scale>

- Update service

  > docker service update --image <image> <service>

- Delete service

  > docker service rm <service>