* UNIT-3

→ Kubernetes is an open source orchestrator for deploying containerized applications

→ It provides the software necessary to successfully build & deploy reliable, scalable distributed systems.

→ Reliable and scalable distributed systems.

* ↔ More and more services are delivered over network via APIs.
* ↔ APIs running on different machines, connected via the network and coordinating their actions via network communication.
* ↔ Does not fail; even if a part of this crashes.
* ↔ Must maintain availability even during software rollouts other (or) maintainence events.
* ↔ ∵ the usage of online services is ↑sing; they must be scalable; where services can grow their capacity to keep-up with ever-↑sing usage.

→ Reasons to use containers & APIs like Kubernetes: Velocity, Scaling, Efficiency.

* Creating and Running Containers:

→ Kubernetes being a platform for creating, deploying and managing distributed applications:

→ Applications comprises of one or more programs that run on individual machines

These programs accept I/P, manipulate the data and then ~~the~~ results are returned.

→ Application programs are typically comprised of a language runtime, libraries and source code.

These applications mainly used external shared libraries which are shipped as shared components in the OS on a particular machine.

→ This usage of external libraries may cause a problem if the shared components are not available when program is sent to production OS.

→ Traditionally running all programs on a single system, would cause a problem, if all of them do not share same version.

* → A program can successfully be executed, only if it is reliably deployed onto machine where it shd run.

* Docker: default container runtime engine
  - → makes it easy to package an executable
  - → push it to a remote registry where it can later be pulled by others.

→ Container registries make it easy for users to manage & deploy private images; while image-builder services provide easy integration with continuous delivery systems.

→ Container images bundle a program & its dependencies into a single artifact under a root file system.

→ Docker Image Format: standardized by Open container initiative to OCI image format.

→ Kubernetes supports both Docker & OCI compatible images.

→ A container image is a binary package that encapsulates all of the files necessary to run a program inside of an OS container.

→ If container image is present on the system; ~~you can~~ running the application in an OS container is easier...

* Docker Image format is developed for packaging, distributing & running containers using docker command.

* → made up of series of file system layers:
  ① Each layer adds, removes (or) modifies files from preceding layer; this is called overlay system.
  ② Used in both packaging the image & the image is actually being used.

* Container images are combined with a container configuration file; which includes info. on how to set up networking, namespace isolation, resource constraints and syscall restrictions.

* Types of containers:

→ System Containers:
  - → mimic VMs & runs a full boot process.
  - → services like: ssh, cron & syslog.
  - → these were used when dockers were new.

→ Application containers.
  - → run a single program on a container.
  - → provides the perfect level of granularity for composing scalable applications.

* Command to create the simple node docker image: $ docker build -t simple node.

* Command to navigate url to access the program running in the container:

* Optimizing Image Sizes:

→ Files that are removed by subsequent layers in the system are actually still present; but are inaccessible. These are transmitted over network.

→ Everytime you change a layer, it changes every layer that comes after it.

→ Changing the preceding layers means that they need to be rebuilt, repushed & repulled to deploy ~~your~~ image to development.

→ To avoid these; order the layers from least likely to change to most likely to change in order to optimize image size for pushing & pulling.

* Image Security:

Problems due to container layers:

↳ Deleting a file in one layer doesn't delete that file from preceding layers. It takes up space and it can be accessed by anyone with right tools.

To avoid this; do not put the passwords along with containers.

* Multistage Image Builds:

→ Compiling code as part of the image build is the easiest way to build a container image from your computer.

→ This helps to avoid the unnecessary development tools, which are large, which slows down the deployments.

→ With multistage builds; rather than producing a single image; a docker file can actually produce multiple images.

* Each image is considered a stage.
* Artifacts can be copied from preceding stages to the current stage.
* Reduces the container image size by hundreds of megabytes & thus dramatically speed up your deployment times.

* Storing images in a remote registry:

→ Kubernetes relies on the fact that images described in a Pod manifest are available across every machine in the cluster.

→ To get the images on any system we work; we can manually import & export; but this may have errors.

→ To avoid this; we store docker images in a remote registry.

→ Public registries allows everyone to download images; but private registeries require authentication.

* Docker Container Runtime:

→ Kubernetes provides an API for describing an application deployment; but relies on a container runtime to set up application.

→ The interface to this container runtime is defined by the Container Run-time interface (CRI) standard.

* Running Containers with Docker:

→ Kubernetes containers are generally launched by a daemon on each node called kubelet.

But its easier with docker CLI tool.

* Limiting Resource Usage:

→ Key benefits of running applications within a container is the ability to restrict resource utilization.

→ Allows multiple applications to coexist on the same hardware & ensures fair usage.

→ If the program in the container uses too much memory, it will be terminated.

→ Restrict CPU utilization using --cpu-shares flag with the docker run command.

* Cleanup:

→ We can delete the docker image: docker rmi

→ Until and unless you explicitly delete an image it will live on the system; even if we build a new image with same name.

→ Docker provides a tool called docker system prune for doing general cleanup.

* Container deployment era:

→ Similar to VMs; but they have relaxed isolation properties to share OS among the applications.

→ Containers are lightweight.

→ Container has its own filesystem; share of CPU, memory and etc.

→ Portable across clouds & OS distributions.

→ Benefits of Containers:

1. Agile application creation & deployment.
2. continuous development, integration & deployment.
3. Dev & ops separation of concerns.
4. Observability.
5. Environmental consistency across development, testing & production.
6. Cloud & Os portability
7. Loosely coupled, distributed, elastic, liberated micro services.
8. resource isolation & utilization.

* Kubernetes: extensible, portable & open-source platform designed by Google.

→ mainly used for automating deployment, scaling & operations of container-based applications across cluster of nodes.

→ provides scalability, predictability & high availability.

→ works along with docker & follows the client-server architecture.

→ Key objects of Kubernetes.

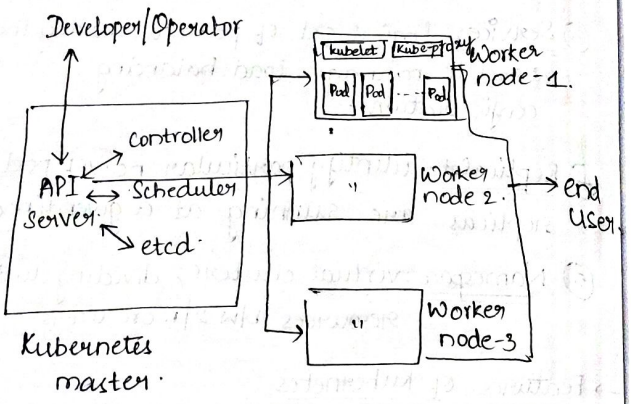
1. Pod: indicates the processes which are running in the cluster.
2. Node: single host, used to run physical (or) virtual machines
3. Service: logical set of pods; works together. used to manage load balancing configurations.
4. ReplicaSet: identify particular no. of pod replicas are running at a given time.
5. Namespace: virtual clusters; dividing cluster resources b/w 2/more users.

→ Features of Kubernetes.

1. Pod
2. Horizontal scaling: automatically ↑se/↓se no. of pods.
3. Automatic bin packing: declare the max & min resources.
4. Service discovery & load balancing: assigns IP addresses & name of DNS for set of containers.

⑤ Automated rollouts & rollbacks.
⑥ Persistent storage.
⑦ Self-healing.

* Kubernetes Architecture:



2 main components → Master Node
→ Slave | Worker Node

Master Node: → manage states of cluster.
- → API Server: receives REST commands.
- → Scheduler: schedules tasks
- → Controller Manager: perform task & manage state.
- → ETCD: stores the cluster data.

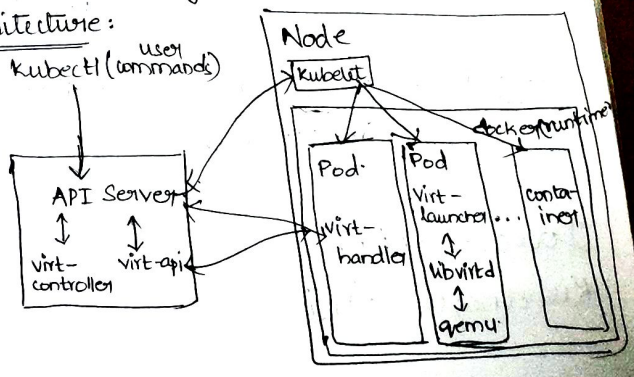
Work | Slave Node: executes applications using pods.
- → Kubelet: ensures pods & their containers are running smoothly.
- → Kube-proxy: requests forwarding
- → Pods.

* Kube Virt:

→ Extension for Kubernetes that enables running and managing VMs alongside containers.

→ Key features:
* VMs are treated as first class citizens in the Kubernetes environment.
* allows users to manage both containers & VMs.
* Portability
* Resource management.

→ Architecture:



* Virt-controller: manages lifecycle of VMs.
  ↳ interacts with Kubernetes API.

* virt-handler: translates Kubernetes VM objects into libvirt VM objects.

* virt-api: extends the Kubernetes API to introduce custom resources for defining & managing VMs.

* virt-launcher: Responsible for launching & managing the QEMU/KVM processes.

* KubeVirt adds additional functionality; to manage VM workloads: 3 categories

→ Customer Resource Definitions: (CRD)
  Used to handle pods.

→ Controllers: provide an API endpoint for managing new KubeVirt CRDs.

→ Agents: manage node tasks related to virtualization.

Docker - container
Kubernetes - API.