**Virtualization with Hypervisor**

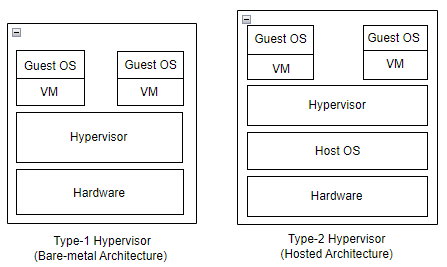
Hypervisor is a piece of software which creates virtual servers on top of physical servers.

**Why use a Hypervisor?**

Hypervisors make it possible to use more of a system’s available resources and provide greater IT mobility since the guest VMs are independent of the host hardware. This means they can be easily moved between different servers, Hypervisor is also referred to as a virtualization layer. Multiple virtual machines can run off of one physical server with a hypervisor, a hypervisor becomes the basic requirement in the cloud world.

 There are two types of hypervisors:

* Type-1, native or bare-metal hypervisors
* Type-2 or hosted hypervisors



**Type-1, Native or Bare-Metal Hypervisors**

These hypervisors run directly on the host’s hardware to control the hardware and to manage guest operating systems. For this reason, they are sometimes called bare metal hypervisors.

This type of hypervisor is most common in an enterprise data center or other server-based environments.

KVM, Microsoft Hyper-V, and VMware vSphere are examples of a type 1 hypervisor.

**Type-2 or Hosted Hypervisors**

These hypervisors run on a conventional operating system (OS) just as other computer programs do. A guest operating system runs as a process on the host. Type-2 hypervisors abstract guest operating systems from the host operating system.

A type 2 hypervisor is better for individual users who want to run multiple operating systems on a personal computer.

VMware Workstation and Oracle Virtual Box are examples of a type 2 hypervisor.

**Limitations of Virtual Machines:**

* Provisioning a virtual machine is time consuming.
* Resource (CPU, memory) allocation is not dynamic (scaling in & scaling out).
* Resources assigned to VMs are utilized by OS (like kernel services, other services) rest will be used by the actual Application.
* Number of VMs that can be spined is more compared to physical servers, but less compared with containers.

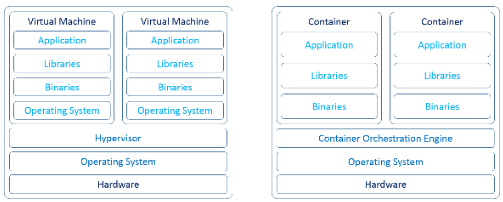
**CONTAINERS:**

Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), can handle more applications and require fewer VMs and Operating systems.

* Containers will have only libraries & binaries which are need to run OS and application to be running, without any kernel packages.
* Containers visualize the OS instead of Hardware and more portable & efficient.
* Docker has worked to make these capabilities approachable & easy to use.

**VIRTUAL MACHINES:**

Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, the application, necessary binaries and libraries – taking up tens of GBs. VMs can also be slow to boot.



Virtual machine Container

Container Tools (Open Source & Enterprise):

1. LXD/LXC (only for linux)
2. CRIO
3. RKT
4. Containerd
5. Docker Engine (Community edition)
6. Docker Enterprise
7. Podman
8. RunC

**Docker**

As per DevOps principle, we have to build, test & deploy the application in fastest way. It is launched in 2013 as an Open Source Docker Engine.

Docker platform provides multiple tools like Docker engine (CE & EE), Docker desktop, Docker compose (for container Orchestration), Docker swarm etc.,

**Docker desktop:**

Docker Desktop is a one-click-install application for your Mac, Linux, or Windows environment that enables you to build and share containerized applications and Micro services. To use all the features of Docker in one particular tool use Docker desktop.

It provides a straightforward GUI (Graphical User Interface) that lets you manage your containers, applications, and images directly from your machine. Docker Desktop can be used either on it’s own or as a complementary tool to the CLI.

Docker Desktop reduces the time spent on complex setups so you can focus on writing code. It takes care of port mappings, file system concerns, and other default settings, and is regularly updated with bug fixes and security updates.

For installing Docker desktop follow this URL: <https://docs.docker.com/desktop/get-started/>

**Docker compose:**

To do some orchestration on Docker engine we use Docker compose. Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application’s services. Then, with a single command, you create and start all the services from your configuration.

Compose works in all environments: production, staging, development, testing, as well as CI workflows. It also has commands for managing the whole lifecycle of your application:

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

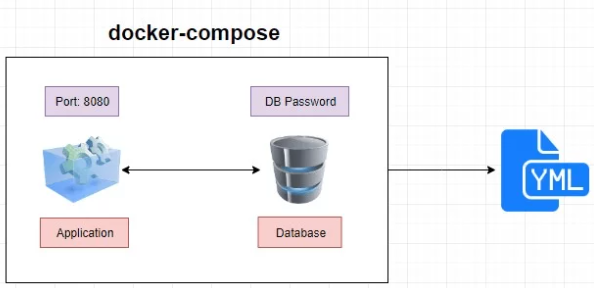
The key features of Compose that make it effective are:

* Have multiple isolated environments on a single host.
* Preserves volume data when containers are created.
* Only recreate containers that have changed.
* Supports variables and moving a composition between environments.

**Process of Docker compose:**

Docker compose is basically a three step process:

Here, is a simple graphical illustration that shows how Docker compose works.

****

* First, we need to define application environment with a [**Dockerfile**](https://ostechnix.com/a-brief-introduction-to-dockerfile/) so it can be reuse again.
* Secondly, we define the services that make up the app in **docker-compose.yml** so they can be run together in an isolated environment.
* Finally, we run **docker-compose up** command and Compose will start and run your entire application.

**Docker swarm:**

* Docker swarm is a container orchestration tool built and managed by Docker, Inc.
* It is the native clustering tool for Docker.
* Swarm uses the standard Docker API, i.e., containers can be launched using normal docker run commands and swarm will take care of selecting an appropriate host to run the container on.

**Docker Engine:**

Docker is written in Go language and release as Open Source in 2013 and takes advantages of several features of the Linux kernel to deliver its Functionality.

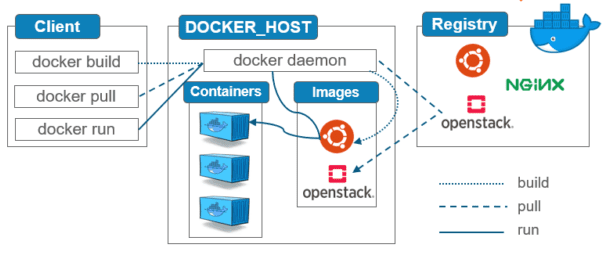
Docker Engine is an open source containerization technology for building and containerizing your applications. Docker Engine acts as a client-server application with:

* A server with a long-running daemon process [dockerd](https://docs.docker.com/engine/reference/commandline/dockerd).
* APIs which specify interfaces that programs can use to talk to and instruct the Docker daemon.
* A command line interface (CLI) client [docker](https://docs.docker.com/engine/reference/commandline/cli/).

The CLI uses [Docker APIs](https://docs.docker.com/engine/api/) to control or interact with the Docker daemon through scripting or direct CLI commands. Many other Docker applications use the underlying API and CLI. The daemon creates and manage Docker objects, such as images, containers, networks, and volumes.

**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. Another Docker client is Docker Compose, that lets you work with applications consisting of a set of containers.



Container creation workflow in Docker Engine:

* Docker CLI used for executing a command. when we execute the command it pass the information to daemon. The Docker daemon receive this information from CLI with the help of rest API calls.
* Unix socket uses rest API calls in order to transfer the information from client to daemon.

**Nginx Application Container:**

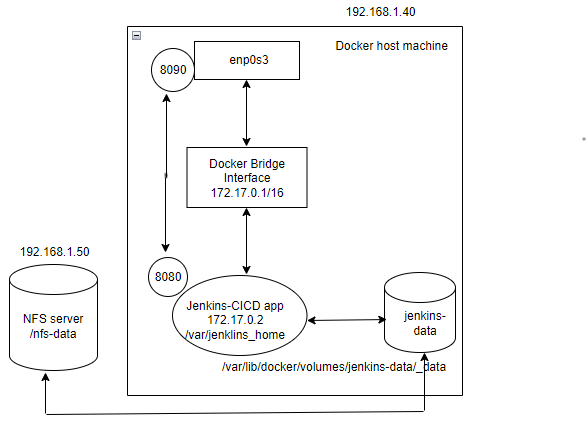
How to store application data on Docker host machine.

If our docker host machine itself is down how to recover application data by using external storage called NFS.

In this we will be having a docker host machine on the docker host machine we will be having a interface with ithe ip and we know we docker

We have a volume mount it is created in a particular path

Now or application data will be stored under NFS server. In this nfs-server I WILL SHARE



Pre-requisites: