**Basics of Docker Containers:**



**What are containers :**

* Containers are lightweight packages of your application code together with dependencies such as specific versions of programming language runtimes and libraries required to run your software services.
* Containers make it easy to share CPU, memory, storage, and network resources at the operating systems level and offer a logical packaging mechanism in which applications can be abstracted from the environment in which they actually run.



## **Virtual machines vs Containers:**



**-Virtualization**

* From our understanding thus far, both virtual machines and Docker containers provide isolated environments to run applications.
* The key difference between the two is in how they facilitate this isolation.
* Recall that a VM boots up its own guest OS. Therefore, it virtualizes both the operating system kernel and the application layer.
* Hypervisor is used to virtualize the operating system kernel along with the application layer.
* A Docker container virtualizes only the application layer, and runs on top of the host operating system.
* Container engine is used to virtualize only the application layer.

## **What are the benefits of containers?**

### **Application isolation**

### Containers virtualize CPU, memory, storage, and network resources at the operating system level, providing developers with a view of the OS logically isolated from other application

### **Separation of responsibility**

### Containerization provides a clear separation of responsibility, as developers focus on application logic and dependencies, while IT operations teams can focus on deployment and management instead of application details such as specific software versions and configurations.

### **Workload portability**

### Containers can run virtually anywhere, greatly easing development and deployment:

### It can run on Linux, Windows, and Mac operating systems; on virtual machines or on physical servers;

* It can run on a developer’s machine or in data centers on-premises; and of course, in the public cloud.

## **What are containers used for?**

* Containers offer a logical packaging mechanism in which applications can be abstracted from the environment in which they actually run.
* This decoupling allows container-based applications to be deployed easily and consistently, regardless of whether the target environment is a private data center, the public cloud, or even a developer’s personal laptop.

### **Agile development**

Containers allow your developers to move much more quickly by avoiding concerns about dependencies and environments.

### **Run anywhere**

Containers are able to run virtually anywhere. Wherever you want to run your software, you can use containers.

### **Efficient operations**

Containers are lightweight and allow you to use just the computing resources you need. This lets you run your applications efficiently.

## **Properties of containers:**

## **Containers Are Scalable**

* Since a container only includes an application and its dependencies, it has a very small footprint on the server.
* Because of this small size, several copies of a container can be launched on each server.
* A higher application density leads to more efficient processing, as there are more copies of the application to distribute the workload.
* Availability of the app is also greatly improved.
* The loss of a single container will not affect the overall function of the service, when there are several others to pick up the lost production.



## **Containers Are Portable**

* The small size of an application container makes it quick to move between servers, up to a cloud, or to mirror to another cluster.
* In addition, a container is completely self-sufficient and has all of the resources that the application needs to run.
* When moving a container, it can just be dropped onto its new home and launched in a plug-and-play fashion.
* You do not need to test for compatibility or load additional software.
* The containerized app can even run directly off of an instance in the cloud, giving you absolute flexibility over availability and the run-time environment.



## **Containers Have an Ecosystem**

* The application container ecosystem is very large and diverse.
* Thousands of containerized applications are available on hubs to use as templates or foundations for proprietary apps.
* A simple container with Apache Web Server can be downloaded, pre-built, saving the development time and resources needed to create common services over and over.



## **Containers Help with Secure Multi-Tenancy**

* Because each application container creates an isolated environment for its application, the resources allocated to it are the entire machine.
* Other copies of the same container are "unaware" of each other.
* As a result, it’s easy to have multiple different applications on the same server, all running simultaneously.
* Each application uses the resources assigned to it.
* There are no concerns about an application taking resources from another application and causing it to crash.
* When an application completes its work, resources are released back to the system.



## **Containers Offer Flexible Deployment**

* You know, now, that application containers are completely self-contained.
* Any application in a container runs in the same exact manner on any hardware.
* Because the container contains all dependencies, it doesn’t require the OS to supply any code.
* As long as the hardware can match the needs of the application, it will happily chug along no matter where you put it.



## **Container Are Easy to Maintain and Specialized for Microservices**

* When an application is containerized, it is saved as an image.
* As the application grows, and as changes are made, these changes are saved as a new image on top of the original one.
* Therefore, any time you need to push or pull an upgrade of a containerized application, all you need to move and install is the new image. You do not need the entire container each time.
* Containers are a highly specialized runtime environment, designed to run a single application as efficiently as possible.
* Because of this specialization, they are perfectly designed to break up large applications into microservices.
* Each microservice can run in its container when called upon, then release the resources back to the system when it is done.
* You can also define a different number of clones for each container, based on the use of each service, allowing for more copies of services that are used more frequently in your app.



## **Application Container Characteristics - Lightweight**

* Application containers consist of just an application, its software dependencies, and a small YAML file with its hardware requirements.
* They use the OS and infrastructure native to the system they are deployed onto.
* Therefore, a container is very lightweight when compared to other virtualization techniques, like VMs. They can be ported quickly to other environments and take just seconds to deploy, clone, or even relaunch in the case of a problem.
* While a container can include more than one application, that is not generally considered a best practice.
* An application container is a streamlined environment, designed to run an application efficiently.
* Adding more applications adds complexity and potential conflict to the package.
* In some cases, tightly coupled applications may share a container. In the vast majority of cases, however, an application will have its own specialized container.



## **High Availability with Containers**

* To create high availability for the application, each service is scaled by spinning up multiple instances of the container image and distributing them across the cluster.
* Each container only includes a single service and its dependencies, which makes them very lightweight.
* Cloning and launching new copies of a container takes just seconds.
* Quickly, several copies of the services are spread out across the cluster, each performing the work sent to it and returning results, independent of the other copies in the environment



## **Container Summary**

* You have seen how application containers speed up the pipeline from production to delivery.
* They solve the problem of having an app working reliably in different environments, making development processes much more efficient and flexible.
* Breaking up large applications into microservices and putting them into containers provides fast and flexible services that can quickly be deployed and moved between systems.
* Containers are not a stand-alone solution, however, and have a few inherent limitations.



## **Docker Architecture:**

## **What is Docker daemon?**

Docker daemon runs on the host operating system. It is responsible for running containers to manage docker services. Docker daemon communicates with other daemons. It offers various Docker objects such as images, containers, networking, and storage. s

Docker follows Client-Server architecture, which includes the three main components that are

* **Docker Client**,
* **Docker Host**
* **Docker Registry**.



## **1. Docker Client**

* Docker client uses commands and REST APIs to communicate with the Docker Daemon (Server).
* When a client runs any docker command on the docker client terminal, the client terminal sends these docker commands to the Docker daemon.
* Docker daemon receives these commands from the docker client in the form of command and REST API's request.

Note: Docker Client has an ability to communicate with more than one docker daemon.

Docker Client uses Command Line Interface (CLI) to run the following commands -

* docker build
* docker pull
* docker run

## **2. Docker Host**

* Docker Host is used to provide an environment to execute and run applications.
* It contains the docker daemon, images, containers, networks, and storage.

## **3. Docker Registry**

Docker Registry manages and stores the Docker images.

There are two types of registries in the Docker -

* **Pubic Registry -** Public Registry is also called as **Docker hub**.
* **Private Registry -** It is used to share images within the enterprise.

## **Docker Objects**

There are the following Docker Objects -

## **Docker Images**

## **Docker Containers**



## **What is a Docker Image?**

* Docker images are the **read-only binary templates** used to create Docker Containers.
* It uses a private container registry to share container images within the enterprise and also uses public container registry to share container images within the whole world.
* Metadata is also used by docket images to describe the container's abilities.
* A **Docker image** is an immutable (unchangeable) file that contains the source code, libraries, dependencies, tools, and other files needed for an application to run.
* Due to their **read-only** quality, these images are sometimes referred to as snapshots.
* They represent an application and its virtual environment at a specific point in time.
* It allows developers to test and experiment software in stable, uniform conditions.
* Since images are, in a way, just **templates**, you cannot start or run them.
* What you can do is use that template as a base to build a container. A container is, ultimately, just a running image.
* Once you create a container, it adds a writable layer on top of the immutable image, meaning you can now modify it.
* The image-base on which you create a container exists separately and cannot be altered.
* When you run a [containerized environment](https://phoenixnap.com/kb/how-to-containerize-applications), you essentially create a **read-write copy** of that filesystem (docker image) inside the container.
* This adds a **container layer**which allows modifications of the entire copy of the image.



* You can create an unlimited number of Docker images from one **image base**.
* Each time you change the initial state of an image and save the existing state, you create a new template with an additional layer on top of it.
* Docker images can, therefore, consist of a **series of layers**, each differing but also originating from the previous one. Image layers represent read-only files to which a container layer is added once you use it to start up a virtual environment.

## **What is a Docker Container?**

* A **Docker container** is a virtualized run-time environment where users can isolate applications from the underlying system.
* These containers are compact, portable units in which you can start up an application quickly and easily.
* Containers are the structural units of Docker, which is used to hold the entire package that is needed to run the application. The advantage of containers is that it requires very less resources.
* In other words, we can say that the image is a template, and the container is a copy of that template.
* A valuable feature is the **standardization** of the computing environment running inside the container.
* Not only does it ensure your application is working in identical circumstances, but it also simplifies sharing with other teammates.
* As containers are autonomous, they provide strong isolation, ensuring they do not interrupt other running containers, as well as the server that supports them.
* Docker claims that these units “provide the strongest isolation capabilities in the industry”. Therefore, you won’t have to worry about keeping your machine **secure** while developing an application.
* Unlike virtual machines (VMs) where virtualization happens at the hardware level, containers virtualize at the app layer.
* They can utilize one machine, share its kernel, and virtualize the operating system to run isolated processes.
* This makes containers extremely **lightweight**, allowing you to retain valuable resources.



## **What is the difference between a Docker image and a Docker container?**



* When discussing the difference between images and containers, it isn’t fair to contrast them as opposing entities.
* Both elements are closely **related and are part of a system** defined by the Docker platform.
* Images can exist without containers, whereas a container needs to run an image to exist.
* Therefore, containers are dependent on images and use them to construct a run-time environment and run an application.
* Having a running container is the final “phase” of that process, indicating it is dependent on previous steps and components.
* That is why docker images essentially govern and shape containers.

## **Docker Networking**

## Using Docker Networking, an isolated package can be communicated. Docker contains the following network drivers -

* **Bridge -** Bridge is a default network driver for the container. It is used when multiple docker communicates with the same docker host.
* **Host -** It is used when we don't need for network isolation between the container and the host.
* **None -** It disables all the networking.
* **Overlay** - Overlay offers Swarm services to communicate with each other. It enables containers to run on the different docker host.
* **Macvlan** - Macvlan is used when we want to assign MAC addresses to the containers.

## **Docker Storage**

## Docker Storage is used to store data on the container. Docker offers the following options for the Storage -

* **Data Volume -** Data Volume provides the ability to create persistence storage. It also allows us to name volumes, list volumes, and containers associates with the volumes.
* **Directory Mounts -** It is one of the best options for docker storage. It mounts a host's directory into a container.
* **Storage Plugins -** It provides an ability to connect to external storage platforms.

## **Written by Vinod Sairam**

The Database Administrator who helps to manage Enterprise Databases who develops Devops Engineering ideas to support Database tasks, Infrastructure tasks and to reach the right audience.

**Assisted by: Shanmugavelu (Database-Devops Engineer)**