**What is Docker and why it is used?**

Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and deploy it as one package. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker’s methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.

**The Docker platform**

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allow you to run many containers simultaneously on a given host. Containers are lightweight because they don’t need the extra load of a hypervisor, but run directly within the host machine’s kernel. This means you can run more containers on a given hardware combination than if you were using virtual machines. You can even run Docker containers within host machines that are actually virtual machines!

**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.



The Docker daemon

The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

The Docker client

The Docker client (docker) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to dockerd, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.

Docker Desktop

Docker Desktop is an easy-to-install application for your Mac or Windows environment that enables you to build and share containerized applications and microservices. Docker Desktop includes the Docker daemon (dockerd), the Docker client (docker), Docker Compose, Docker Content Trust, Kubernetes, and Credential Helper. For more information, see [Docker Desktop](https://docs.docker.com/desktop/).

Docker registries

A Docker *registry* stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry.

When you use the docker pull or docker run commands, the required images are pulled from your configured registry. When you use the docker push command, your image is pushed to your configured registry.

-----------------------------------------------------------------------------------------------------------------------------------------

Docker objects

When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects. This section is a brief overview of some of those objects.

*Images*

An *image* is a read-only template with instructions for creating a Docker container.

You might create your own images or you might only use those created by others and published in a registry.

To build your own image, you create a *Dockerfile* with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

*Containers*

A container is a runnable instance of an image.

You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state..

-------------------------------------------------------------------------------------

Docker workflow=>



**Virtualization vs Containerization**

**Virtualization** enables you to run multiple operating systems on the hardware of a single physical server, while **containerization** enables you to deploy multiple applications using the same operating system on a single virtual machine or server.

 

**Virtualization containerization**

|  |  |  |
| --- | --- | --- |
| **Area** | **Virtualization** | **Containerization** |
| Isolation | Provides complete isolation from the host operating system and the other VMs | Typically provides lightweight isolation from the host and other containers, but doesn’t provide as strong a security boundary as a VM |
| Operating System | Runs a complete operating system including the kernel, thus requiring more system resources such as CPU, memory, and storage | Runs the user-mode portion of an operating system, and can be tailored to contain just the needed services for your app using fewer system resources |
| Guest compatibility | Runs just about any operating system inside the virtual machine | Runs on the same operating system version as the host |
| Deployment | Deploy individual VMs by using Hypervisor software | Deploy individual containers by using [Docker](https://www.baeldung.com/docker-java-api) or deploy multiple containers by using an orchestrator such as [Kubernetes](https://www.baeldung.com/kubernetes) |

DockerFile=>

A Dockerfile is **a text file that Docker reads in from top to bottom**. ... A Dockerfile is a blueprint for building Docker images

Docker File commands=>

* ADD – copies the files from a source on the host into the container’s own filesystem at the set destination
* CMD – can be used for executing a specific command within the container
* ENTRYPOINT – sets a default application to be used every time a container is created with the image
* ENV – sets environment variables
* EXPOSE – associates a specific port to enable networking between the container and the outside world
* FROM – defines the base image used to start the build process
* MAINTAINER – defines a full name and email address of the image creator
* RUN – central executing directive for Dockerfiles
* USER – sets the UID (or username) which is to run the container
* VOLUME – is used to enable access from the container to a directory on the host machine
* WORKDIR – sets the path where the command, defined with CMD, is to be executed

Ex. Create file name as Dockerfile and write cmd on that

FROM node:12-alpine

RUN apk add --no-cache python2 g++ make

WORKDIR /app

COPY . .

RUN yarn install --production

CMD ["node", "src/index.js"]

Docker Commands=>

* docker build -t chimg .
* docker run -it --name mycon1 chimg
* docker start mycontainer
* docker attach mycontainer
* docker run -d -p 8000:8000 myimage

Docker Compose=>

Docker Compose is a tool that was developed to help define and share multi-container applications. With Compose, we can create a YAML file to define the services and with a single command, can spin everything up or tear it all down.

 Docker Compose can connect different containers as a single service.

Docker Compose is used for running multiple containers as a single service. Each of the containers here run in isolation but can interact with each other when required. Docker Compose files are very easy to write in a scripting language called YAML, which is an XML-based language that stands for Yet Another Markup Language. Another great thing about Docker Compose is that users can activate all the services (containers) using a single command.

For example:

If you have an application that requires an NGINX server and Redis database, you can create a Docker Compose file that can run both the containers as a service without the need to start each one separately.



Basic Commands in Docker Compose

* Start all services: Docker Compose up
* Stop all services: Docker Compose down
* Install Docker Compose using pip: pip install -U Docker-compose
* Check the version of Docker Compose: Docker-compose-v
* Run Docker Compose file: Docker-compose up -d
* List the entire process: Docker ps
* Scale a service - Docker Compose up -d -scale

Use YAML files to configure application services - Docker Compose.yml