**Docker Tutorials**

**What is Docker?**

It is an open source platform tool design to manage the containers. It allows us to build the application in a containers with the required libraries, dependencies, binaries to run the application

**(OR)**

Docker is a software platform that allows you to build, test, and deploy applications quickly. Docker packages software into standardized units called [containers](https://aws.amazon.com/containers/) that have everything the software needs to run including libraries, system tools, code, and runtime. Using Docker, you can quickly deploy and scale applications into any environment and know your code will run.

Docker is a platform for developing, shipping, and running applications in containers. Containers are lightweight, standalone, and executable packages that contain everything needed to run an application, including the code, runtime, libraries, and dependencies.

**Why we use Docker?**

Docker is an open source platform that enables developers to build, deploy, run, update and manage containers

Fast delivery & Scalable

Portability

Light weight

Resource optimization

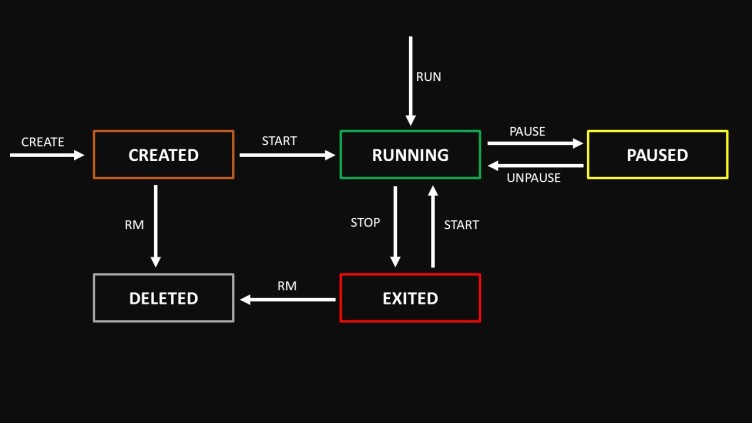
**When to use Docker**

You can use Docker containers as a core building block creating modern applications and platforms. Docker makes it easy to build and run distributed microservices architecures, deploy your code with standardized continuous integration and delivery pipelines, build highly-scalable data processing systems, and create fully-managed platforms for your developers. The recent collaboration between AWS and Docker makes it easier for you to deploy Docker Compose artifacts to Amazon ECS and AWS Fargate.

**Docker Life Cycle**

The Docker container lifecycle is comprised of several states, such as `created`, `running`, `paused`, `exited`, and dead. In this article, we will examine each of these states in detail and explore what they represent and when a container enters each state. By understanding these stages, we will have a better grasp of how to work with Docker containers.

**This animation shows a high-level overview of the container lifecycle.**



**Summary of these states before we deep dive into it:**

**created** - This is the initial state of a Docker container after it has been created, but before it has been started.

**running** - When a Docker container is started, it transitions to the running state. In this state, the container is actively executing its processes.

**paused** - If a container is paused, it is temporarily stopped from running its processes, but it is not terminated.

**exited** - If a container's main process completes, the container stops and transitions to the exited state.

**dead** - If a container fails to start, it is in the dead state. Containers in this state cannot be restarted and must be recreated.

**Container States in detail**

**Started**

This is the first state in the container lifecycle. This state denotes that the container is created but it's not running. This is achieved using the `docker create` command.

**docker container create --name container01 dev/docker:v2**

When we create a Docker container, a read-write (R/W) layer is added to the read-only (R/O) layer of the image we've chosen. This prepares the container to run the program like pulling the image, setup the environment variables, setup entrypoint, etc.

It's important to note that creating a container does not automatically start the program. However, we can set all the necessary configuration parameters during the creation process, such as CPU and memory limits, container image, and capabilities. When the container is in this state, the configuration of the container can also be updated using the `docker update` command.

This means that we can create the container once with all the required parameters and start it later without having to specify them again.

Another thing to note here is that resources are not allocated in this state.

**Running**

This state denotes that the commands listed in the image are being executed one by one by the container.

**docker container start container01**

When we start a container, Docker sets up the resources that the container needs, such as network resources, memory, CPU, etc. Then, it creates an environment for the container to run in.

Once this is done, the container is up and running and begins to perform the tasks assigned to it.

The task done by the above two commands can be achieved using a single command also (`docker run`). This command creates the container and then starts it immediately.

**docker run -d --name container02 dev/docker:v2**

**Killed / Exited**

The `exited` state refers to a container that has stopped running. A container can transition to the exited state for a variety of reasons, including:

The process running inside the container completed its task and exited.

The process running inside the container was stopped by a user or an external signal.

An error occurred while running the process inside the container.

The killed state is a sub-state of the exited state. When a container is killed, it means that the process inside the container was forcibly terminated by Docker. This can happen if a user runs the `docker kill` command, or if a container fails to respond to a SIGTERM signal and Docker sends a SIGKILL signal to force the container to stop.

Command to stop a docker container:

**docker container stop container01**

When we run `docker stop` command, it goes through the following steps:

Docker sends a SIGTERM signal to the main process (PID 1) running inside the container. This signal is a request for the process to stop gracefully.

If the process does not respond to the SIGTERM signal within a set amount of time (10 seconds by default, to override use the `-t` flag), Docker sends a SIGKILL signal to forcefully terminate the process.

Once the process is stopped, Docker moves the container to the exited state.

If the container was started with the --rm flag, Docker will remove the container and its filesystem from the system.

Overall, running the docker stop command allows us to gracefully stop a container and cleanly shut down the program or process running inside it.

**Paused**

The `paused` state refers to a container that has been temporarily suspended. When a container is paused, it is still running, but all of its processes are paused and no new processes can be started until the container is unpaused.

Pausing a container can be useful when we need to temporarily free up resources on the host system or when we need to troubleshoot an issue with the container. When a container is paused, it continues to consume resources such as memory and CPU, but at a much lower rate than when it is running normally.

Pausing the container still holds the state of the execution in the memory, and on unpausing it will resume from the same point. For e.g. if my docker container counts from 1 to 100, and if I pause it at any point, on resuming it will resume from that point. When it's in paused the CPU usage would be almost 0 but memory usage would be non-zero.

To pause a running container, you can use the docker pause command followed by the container ID or name. To unpause a paused container, you can use the docker unpause command followed by the container ID or name.

**docker pause container01**

**docker unpause container01**

It's important to note that not all containers can be paused. Containers that are running in privileged mode or with certain types of system capabilities cannot be paused. Additionally, pausing a container that is running a critical process can have unintended consequences, so it's important to use caution when pausing containers in production environments.

**Deleted**

There's no official state as Deleted, as in this state the container is already removed.

The `deleted` state refers to a container that has been removed from the Docker host system. When a container is deleted, all of its resources, including its filesystem and configuration, are permanently removed from the host system.

Deleting a container can be done using the docker rm command followed by the container ID or name. When a container is deleted, it cannot be restarted or resumed, and any data or changes that were made to the container are lost.

**docker container rm container01**

It's important to note that deleting a container does not remove the image that was used to create the container. The image can still be used to create new containers, and any changes made to the image will be reflected in new containers that are created from that image.

**Dead**

When a Docker container cannot be removed due to some resources still being in use by an external process, it is moved to the dead state. In this state, the container becomes non-functional and cannot be restarted. It can only be removed. Although it is partially removed, it does not consume any memory or CPU.

**Some other Useful docker commands:**

**List Containers**

docker container ls # Lists the running containers

docker ps # Same as docker container ls

docker container ls -a # Lists all the containers irrespective of state

docker container ls --filter=status=exited # Lists the containers matching filter.

Get usage of containers

docker stats is a command that allows you to monitor real-time resource usage for running Docker containers. When you run this command, it provides an ongoing stream of resource usage statistics for all of the running containers. The statistics include information such as CPU usage, memory usage, network I/O, and block I/O.tk I/O, and block I/O.

**docker stats # Shows usage statistics for only running containers**

**docker stats -a # Shows usage for all containers**

Remove all containers in exited state

**docker container prune**

Get all the info about the docker container

**docker container inspect <container name or container id>**

**Docker Components :**Major 3 components in Docker:

**1) Docker Client**

It mainly manages the docker components.

Ex: containers, images, storage values, build, pull or delete

Docker components : images, containers, n/w’s. Storge volume, we can able to build the image, run the image, create, pull, restart, stop

Docker client will interact with the docker host

**2) Docker Host:**

A Docker host is a physical or virtual server on which the core component of Docker runs, the Docker engine . The Docker engine encapsulates and runs workloads in Docker containers.

Docker host is the server

It accepts and listens the request from docker client

**3) Docker Hub (Docker Registry):**

Docker Hub is a service provided by Docker for finding and sharing container images

**Docker Hub** is a repository service and it is a cloud-based service where people push their Docker Container Images and also pull the Docker Container Images from the **Docker Hub** anytime or anywhere via the internet. It provides features such as you can push your images as private or public.

In this docker hub we have 2 types of registry : local docker hub, docker hub (Centralized)

Docker hub is available in the internet, it will be managed by the cloud company

Pull - download, push - upload

We have 2 services - 1. Public & 2. Private

### **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 Objects**

There are the following Docker Objects -

**Docker Images**

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.

**Docker Containers**

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.

**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.

**Docker file**

A Docker file is a text document that contains commands that are used to assemble an image. We can use any command that call on the command line. Docker builds images automatically by reading the instructions from the Docker file.

The docker build command is used to build an image from the Docker file. You can use the -f flag with docker build to point to a Docker file anywhere in your file system.

$ docker build -f /path/to/a/Docker file .

**Dockerfile Instructions**

The instructions are not case-sensitive but you must follow conventions which recommend to use uppercase.

Docker runs instructions of Dockerfile in top to bottom order. The first instruction must be **FROM** in order to specify the Base Image.

**FROM**:

Specifies the base image to use for the Docker image. It is typically the starting point for building your image and can be an official Docker image or a custom image.

**(OR)**

This instruction is used to set the Base Image for the subsequent instructions. A valid Dockerfile must have FROM as its first instruction.

**Syntax :** FROM tomcat:latest or 7

Mandatory instruction in docker is “FROM

**MAINTAINER Instruction:**

It is used to specify the author who creates the new docker image for the support

Ex:: Maintainer DevOps-MArolix

      MAINTAINER [marolix@gmail.com](mailto:marolix@gmail.com)

**LABEL**

We can add labels to an image to organize images of our project. We need to use LABEL instruction to set label for the image.

**(OR)**

To specify the metadata information to an image, we use label instruction.

Label instruction is a key-value pair.

**Ex:** LABEL “Application\_environment” = “DevopsMARCH”

LABEL”Application\_Support”= “DevopsMarch”

**Copy:**

Copies files from the host machine to the image. The COPY instruction is used to copy files and directories from the build context to the image

**(OR)**

This instruction is used to copy new files or directories from source to the filesystem of the container at the destination.

**Syntax:** Zip format - src in dest - zip format

Shell form = COPY src dest

Executable form = COPY [“src” “dest”]

**ADD:**

To copy the files from local to the docker images we will use ADD Instruction

**Synatx:** 2 Forms - Shell form, Executable form

Shell form = ADD src dest

Executable form = ADD [“src” “dest”]

Recommended form is “SHELL FORM”

In zip - in add - it will get extract unzip

**RUN**:

This instruction is used to execute any command of the current image.

It is used to execute the commands on top of the current images and this will create New layer

2 forms - Shell form = RUN “yum” “update”

Executable form = RUN systemctl start ngnix, httpd

**Volume Instruction :**

To create or mount a volume to the docker container from the docker host filesystem

Container is a application

Ex: Volume /data /appdata

**WORKDIR**:

Sets the working directory for any subsequent instructions. It is used to provide a context for relative paths in subsequent instructions like RUN, CMD, or ENTRYPOINT.

**(OR)**

To set the specific path in docker we use WORKDIR Instruction

**Synatx:** WORKDIR /opt/maven

WORKDIR usr/lib/java

**EXPOSE**:

Informs Docker that the container listens on specific network ports at runtime. However, it does not publish the ports to the host machine. This instruction is typically used for documentation purposes.

**(OR)**

If we can expose your port no we use Expose instruction.

To change the port no - listening - 8080 run - 8085

If we want to interconnect b/w your container we will use this expose instruction

**Syntax:** EXPOSE 8080 8085

Expose 8080/tcp 8085/udp

**CMD**:

Specifies the default command to run when the container starts. It can be overridden by providing a command at runtime. Only the last CMD instruction in the Dockerfile takes effect.

**(OR)**

At the last -why means - CMD ll (we required o/p as ll)

  CMD ls (It will run this command only)

It is used to set commands to execute while running the container. There must be only one command in  a docker file.

If we have used more than one cmd instruction whatever the latest instruction it will get execute.

2 forms - Shell form = CMD src dest

Executable form = CMD [“src” “dest”]

**ENTRYPOINT**: Defines the primary command to be executed when the container starts. Unlike the CMD instruction, the ENTRYPOINT instruction is not overridden by providing a command at runtime. It is commonly used for executable Docker images.

**(OR)**

In docker file we can create mutiple entrypoints. It is used to configure and run a container

Ex: ENTRYPOINT Ping google.com

      ENTRYPOINT ping youtube.com

**ENV**:

Sets environment variables in the image. These variables are accessible during the build process and when the container is running.

**Containerization and virtualization**

Containerization and virtualization are both technologies used to run multiple software applications or operating systems on a single physical server or computer, but they do so in different ways.

**Virtualization**:

**Concept**: Virtualization involves creating multiple virtual machines (VMs) on a single physical server. Each VM emulates a complete computer with its own operating system and resources, such as CPU, memory, and storage.

**Example**: Imagine you have a powerful computer, and you want to run both Windows and Linux on it. With virtualization, you can create two virtual machines on that single physical computer. One VM runs Windows, and the other runs Linux. Each VM operates independently as if it were on its own physical computer.

**Pros**:

Isolation: VMs are isolated from each other, which means issues in one VM don't affect others.

Flexibility: You can run different operating systems on the same hardware.

**Cons**:

Resource overhead: Running multiple full-fledged operating systems consumes more resources.

Slower startup: VMs may take longer to start compared to containers.

**Containerization**:

**Concept**: Containerization, on the other hand, involves running applications in lightweight containers that share the same operating system kernel. Containers package an application and its dependencies together into a single unit, making them portable and consistent across different environments.

**Example**: Think of containers like shipping containers in the real world. You can have various goods (applications) inside containers, and these containers are standardized and can be easily transported from one ship or truck (server) to another. Docker is a popular tool for creating and managing containers.

**Pros**:

Efficiency: Containers are more lightweight and efficient because they share the host OS.

Portability: Containers work consistently across different environments, from development to production.

**Cons**:

Less isolation: Containers share the same OS kernel, so they are not as isolated as VMs. This can be a security concern if not properly configured.

In summary, virtualization creates full virtual machines that run multiple operating systems on one physical server, while containerization uses lightweight containers to run applications in isolated environments, all sharing the same operating system. The choice between them depends on your specific use case, with containers often being favored for modern, cloud-native applications due to their efficiency and portability.

**What Is Bind Mounts And Volumes In Docker**

In Docker, both bind mounts and volumes are mechanisms used for managing and persisting data in containers. They allow you to share data between the host system and the container or between containers while ensuring data persistence and flexibility. Let's explore both concepts:

**Bind Mounts**:

**Description**: A bind mount is a way to mount a specific directory or file from the host machine into a container. It essentially creates a direct link between a directory on your host system and a directory in your container.

**Use Cases**:

Sharing configuration files or scripts between the host and container.

Real-time development and testing, where changes made on the host are immediately reflected in the container.

**Example**:

docker run -v /path/on/host:/path/in/container my-image

This command maps the directory **/path/on/host** on your host system to **/path/in/container** in the container.

**Pros**:

Data is easily accessible and editable on the host system.

Suitable for development and debugging.

**Cons**:

Limited to the host machine where the container is running.

Potential security concerns as container processes have access to the host's filesystem.

**Volumes:**

**Description**: Volumes are a Docker feature that provides a way to manage and persist data independently of the container's lifecycle. Docker manages volumes, and they can be shared among multiple containers.

**Use Cases**:

Storing database data, logs, or application configuration files.

Sharing data between containers in a multi-container application.

**Example**:

docker run -v my-volume:/path/in/container my-image

In this command, a volume named **my-volume** is created and mounted at **/path/in/container** in the container.

**Pros**:

Data in volumes persists even if containers are stopped or removed.

Volumes can be easily backed up, moved, or managed independently.

Multiple containers can share the same volume.

**Cons**:

Data in volumes may not be as easily accessible or editable on the host system compared to bind mounts.

In summary, bind mounts are suitable for development and debugging purposes when you need direct access to files on the host system. Volumes, on the other hand, are more versatile for production scenarios, as they offer data persistence, easy management, and the ability to share data among containers in a safer and more controlled manner. The choice between bind mounts and volumes depends on your specific use case and requirements.

**Docker Networking**

**What are different types of Networking in Docker?**

Docker provides several types of networking options:

**Bridge Network**: This is the default network type for Docker. It creates a private internal network on the host that allows containers to communicate with each other using private IP addresses.

**Host Network**: This mode allows containers to share the network namespace with the host, making them behave as if they were running directly on the host's network. It can provide higher performance but less isolation.

**Overlay Network**: Used in Docker Swarm mode for multi-host communication. Overlay networks allow containers across multiple hosts to communicate seamlessly.

**Macvlan Network**: This allows containers to have their own MAC addresses and appear as physical devices on the network. It's useful for scenarios where containers need to mimic physical devices on the network.

**Custom Bridge Networks**: You can create your own bridge networks to isolate groups of containers and control their communication. These networks are user-defined and provide isolation.

**Which Networking is default and Out of the Box?**

The default and out-of-the-box network mode for Docker is the "bridge" network. When you create a container without specifying a network, it will be attached to the bridge network by default.

**Play with Docker containers and inspect their Networks**:

To inspect the networks associated with a container, you can use the **docker inspect** command. For example:

docker inspect <container\_name\_or\_id> | grep Networks

Replace **<container\_name\_or\_id>** with the name or ID of your container. This command will display information about the container's network settings.

**Create a custom bridge network to secure it from other containers?**

To create a custom bridge network in Docker and isolate containers from other networks, you can use the following command:

docker network create my-custom-network

This command creates a new bridge network named **my-custom-network**. You can then launch containers and specify this network using the **--network** option:

docker run --network my-custom-network my-image

Containers connected to this custom network can communicate with each other but are isolated from containers in the default bridge network and other networks, providing a degree of network security and segmentation.

**Docker Compose**

Docker Compose is a tool used for defining and running multi-container Docker applications. It allows you to use a simple YAML file to configure the services, networks, and volumes required for your application, making it easy to define complex applications and manage their dependencies.

Here's an in-depth explanation of Docker Compose along with a simple example script:

**Docker Compose Explanation:**

Docker Compose simplifies the process of running multiple Docker containers together as a single application. Instead of manually running each container with lengthy Docker commands and managing their interdependencies, Docker Compose allows you to define all these configurations in a single YAML file called **docker-compose.yml**.

This YAML file contains the specifications for each service in your application, including its image, ports, environment variables, volumes, and any other configurations needed. Docker Compose then reads this file and orchestrates the creation and management of the containers accordingly.

Simple Example Script:

Let's say we want to create a simple web application with two services: a web server and a database. We'll use Docker Compose to define and run these services together.

**Step 1: Create a docker-compose.yml file**

version: '3.8'

services:

web:

image: nginx:latest

ports:

- "8080:80"

volumes:

- ./app:/usr/share/nginx/html

depends\_on:

- db

db:

image: mysql:5.7

environment:

MYSQL\_ROOT\_PASSWORD: example

MYSQL\_DATABASE: mydb

volumes:

- db\_data:/var/lib/mysql

volumes:

db\_data:

**Step 2: Explanation of the** docker-compose.yml **file:**

**version**: Specifies the version of the Docker Compose file format.

**services**: Defines the services (containers) that make up the application.

**web**: Configuration for the web server service.

**image**: Specifies the Docker image to use for the web server (nginx in this case).

**ports**: Maps port 8080 on the host to port 80 on the container.

**volumes**: Mounts the **./app** directory on the host to **/usr/share/nginx/html** inside the container.

**depends\_on**: Specifies that the **web** service depends on the **db** service.

**db**: Configuration for the database service.

**image**: Specifies the Docker image to use for the database (MySQL 5.7 in this case).

**environment**: Sets environment variables required by the MySQL container.

**volumes**: Mounts a volume named **db\_data** to persist MySQL data.

**volumes**: Defines a named volume **db\_data** used by the database service to persist data.

**Step 3: Running the Application**

To run the application, navigate to the directory containing **docker-compose.yml** file and run:

docker-compose up

This command will start both services defined in the **docker-compose.yml** file, creating the necessary containers, networks, and volumes as specified. You should now be able to access the web server at **http://localhost:8080**.

That's it! Docker Compose simplifies the management of multi-container applications, allowing you to define, run, and manage complex applications with ease using a single configuration file.

**Port Binding**

Port binding, in the context of Docker and containerization, refers to the process of mapping network ports between the host system and containers. It allows external traffic to reach services running inside containers and enables communication between containers and the host or other containers.

**How Port Binding Works:**

When a containerized application exposes a network service, such as a web server listening on port 80, it needs to bind that port to a port on the host system. Port binding allows incoming traffic on a specific port of the host to be forwarded to the corresponding port of the container where the service is running.

**Persisting container data using Docker Volume.**

**Explanation:**

When you run a program in a Docker container, any data it creates or uses inside the container usually disappears when the container stops. This includes things like log files or databases. Docker volumes help solve this problem by letting you keep that data safe and accessible even if the container stops or gets deleted.

**Example Script:**

Step 1: Create a Docker Volume

Imagine you have a special box (a Docker volume) where you want to store important papers. To create this box, you run a command like:

docker volume create my\_documents

Step 2: Dockerfile

Next, you need to prepare a container (let's call it a house) to work with this box. You write down instructions (a Dockerfile) for building the house:

FROM nginx:latest

# Prepare the house

RUN rm /var/log/nginx/access.log /var/log/nginx/error.log

# Make connections to the box

RUN ln -s /var/log/nginx/access.log /logs/access.log && \

ln -s /var/log/nginx/error.log /logs/error.log

Step 3: Docker Compose File

Then, you write down a plan (a Docker Compose file) to build the house and connect it with the box:

version: '3.8'

services:

web:

build: .

ports:

- "8080:80"

volumes:

- my\_documents:/logs

volumes:

my\_documents:

Step 4: Build and Run the Docker Container

Now, you start building the house and connect it with the box:

docker-compose up -d --build

This command builds the house and links it with the box. It also tells the house to show itself on the street (by mapping port 8080 to port 80).

Step 5: Verify Data Persistence

After some time, you check if the important papers are still safe inside the box:

docker volume inspect my\_documents

You see that the papers are still there, safe and sound.

**Conclusion:**

Using Docker volumes is like having a special storage box outside your house where you can keep important things. Even if you rebuild your house or move to a new one, your important stuff remains safe in the box. It's a simple and reliable way to make sure your data stays around even when your Docker containers are stopped or removed.