Understanding Docker Image Creation and Layers

*Docker Basics and Image Creation:*

Docker is a powerful platform for automating containerized applications. Containers are lightweight, portable, and efficient units that encapsulate an application and its dependencies. Docker images serve as the blueprint for containers, and they can be created either from public repositories like Docker Hub or custom-built using Dockerfiles.

1. Initial Setup:

● Create a new folder on your local system:

mkdir /code1

Create a Dockerfile within the newly created folder:

vim Dockerfile

1. Dockerfile Structure:

A Dockerfile is a script containing a set of instructions for building a Docker image. Each instruction creates a layer, and the combination of layers forms the final image.

Example Dockerfile:

# Dockerfile

FROM centos:7

RUN command1

RUN command2

RUN command3

1. Build Docker Image:

Build the Docker image in the current folder using the following command:

docker build -t zz:v1 .

During the build process, Docker follows a series of steps, creating intermediate images for each instruction in the Dockerfile.

1. Image Creation Process:
   * Docker starts with a base image (e.g., centos:7 ) and launches a container.
   * Each RUN command in the Dockerfile is executed within the container.
   * After successfully running a command, Docker commits the container, creating an intermediate image.
   * The process repeats for each command in the Dockerfile until the final image is created.
2. Image Layers:
   * Each intermediate image represents a layer in the image's history.
   * The final image contains all the layers, and each layer is immutable.
   * Layers are like building blocks, and Docker optimizes storage by reusing common layers.
3. Viewing Image History:

Check the history of the zz:v1 image: docker history zz:v1

This command provides insights into the layers and commands executed during image creation.

1. Image Storage and Layers:
   * Docker images are stored in the UFS (Union File System) format or overlay file system in the latest Docker versions.
   * Containers launch within seconds by mounting the image to the operating system.
2. Intermediate Image Inspection:

Inspect an intermediate image within the zz:v1 image: docker inspect --type image zz:v1

This command reveals details about the intermediate image, including its unique ID.

1. Efficient Image Usage:
   * Containers can be launched from any intermediate image, inheriting the commands and programs present in that image.
   * Docker images are like pendrives, enabling the rapid launch of multiple containers with minimal space consumption.
2. Optimizing Build Time:
   * To decrease build time, reduce the number of layers.
   * Create a new Dockerfile with fewer layers for faster image creation.
3. Layer Optimization Example:

# Optimized Dockerfile

FROM centos:7

RUN command1 && command2 && command3

1. Hash Values and Layer Verification:
   * Each layer has a unique hash value, equivalent to the intermediate image ID.
   * Docker checks for existing layers before downloading an image, optimizing space and avoiding redundant downloads.
2. Extracting Image Layers: To extract the layers from a tar file:

tar -xvf my.tar

This process allows inspecting the content of the layers.

Docker Container Management and

Image Operations

1. Attaching to Containers:
   * Use the docker attach command to interact with a running container named os123 . It usually enters the bash process:

docker attach os123

* + The first process inside the container is typically the bash process.

1. Resource Monitoring:
   * Check the number of CPUs in the container using lscpu :

lscpu

Check available RAM using free -m  :

free -m

4. Docker Run Commands:

● View all related commands for docker run : docker run --help

Linux Capabilities:

* + Use the cap keyword to manage Linux capabilities.

1. Changing Hostname:
   * Only root can change the hostname.

docker run --privileged -it centos:7

1. Memory Limit for Containers:
   * Set a memory limit for a container: docker run -it --memory 100M centos:7
2. Image Layers and Runtimes:
   * Docker runs containers using runc, while Podman uses crun in a new version.
3. Namespace and nsenter:
   * Use nsenter -h to show a list of all namespaces.
   * Explore various namespaces to understand container isolation.

nsenter --target <PID> --mount --uts --ipc --net --pid

The  nsenter  command is a powerful tool in Linux that allows you to enter namespaces of other processes. Namespaces are a key feature in containerization, providing isolation for processes, filesystems, network, etc. The  nsenter  command enables you to join an existing process's namespaces, essentially allowing you to interact with and manipulate the resources associated with that process. Let's break down the command you provided:

nsent--target <PID> --mount --uts --ipc --net --p

* + --target <PID> : Specifies the process ID (PID) whose namespaces you want to enter. This is the target process whose namespaces you want to access.
  + --mount : Enters the target process's mount namespace. The mount namespace determines the filesystem mount points visible to a process. By entering this namespace, you gain access to the same filesystem view as the target process.
  + --uts : Enters the target process's UTS namespace. The UTS namespace isolates the hostname and the NIS (Network Information Service) domain name. By entering this namespace, you can view or modify the hostname and related information.
  + --ipc : Enters the target process's IPC (Inter-Process Communication) namespace. The IPC namespace isolates inter-process communication resources such as message queues and semaphores. By entering this namespace, you can interact with the same IPC resources as the target process.
  + --net : Enters the target process's network namespace. The network namespace provides isolation for network-related resources, such as network interfaces, routing tables, and firewall rules. By entering this namespace, you can manipulate network settings associated with the target process.
  + --pid : Enters the target process's PID namespace. The PID namespace isolates the process ID space, making processes inside a namespace appear as if they are the only processes running on the system. By entering this namespace, you can view or manipulate processes within the same PID namespace as the target process.

1. Network Namespace:
   * Each container has its own network card.

1. Image Registry:
   * Push an image to a registry, either public (e.g., Docker Hub) or private (self-managed).
2. AWS Elastic Container Registry (ECR):
   * AWS offers ECR to share, store, and deploy container images.
3. Image Upload to Docker Hub:
   * Build and create a custom image locally: docker build -t your\_id/your\_image:tag .

Search for the image on Docker Hub:

docker search your\_id

1. Docker Hub Login and Image Push:

Log in to Docker Hub: docker login

Push the image to Docker Hub:

docker push your\_id/your\_image:tag

1. Image Tagging and Versioning: Tag the image with a version: docker tag your\_id/your\_image:tag your\_id/your\_image:new\_version Push the new version:

docker push your\_id/your\_image:new\_version

1. Fast Updates with Image Versions:
   * When updating the image, use versioning for faster pushes:

docker push your\_id/your\_image:new\_version