Assignment 6 - Sharams Kunwar – Docker:

Images, Containers, Services

Docker Images

- It's a file used to execute code in Docker container.
- It acts as a set of instructions to build a Docker container, basically, a template.
- The docker image contains application code, libraries, tools, dependencies and required files to run the application.
- When a user runs an image, it becomes one or many instances of a container.
- It has multiple layers: each one originates from the previous layer, but each layer is different, and they are read-only which can be shared between any container started from the same image.
- The layers allow reusability and make it lightweight and portable.
- After the creation of container by pulling image, a writeable layer is added on top to allow user to make desirable changes.
- Size of the container refers to disk space of a writeable layer of container and the virtual size of the container refers to disk space used for container and writeable layer.
- The 'docker run' command is used to create a container from specific image.

Layers of Docker Image

- **Base Image:** This is the first layer which can be built by user entirely from scratch with the 'docker build' command.
- **Parent Image:** A parent image can also be used as the first layer, which is a reused image which serves as a foundation for other layers.
- **Layers:** Layers are added to the base image, using code to enable it to run in a container. The layers can be viewed using 'docker history' command. Docker shows all top-layer images by

default, intermediate layers are cached to allow easy viewing of the top layers. Docker also has storage drives to handle management of image layer contents.

- **Container layer:** This layer hosts changes made to running container and stores newly written and deleted files allowing customization to containers.
- **Docker manifest:** This layer is an additional file and uses JSON format to describe the image, like image tags and digital signature.

Docker Image Repositories

Docker images get stored in private or public repositories, such as those in the 'Docker Hub' cloud registry service, from which users can deploy containers and test and share images. Docker Hub's Docker Trusted Registry also provides image management and access control capabilities. Users can also create new images from existing ones and use the docker push command to upload custom images to the Docker Hub.

Basic Docker Image Commands

docker image build.	Builds an image from a Dockerfile.
docker image inspect.	Displays information on one or more images.
docker image load.	Loads an image from a tar archive or streams for receiving or reading input (STDIN).
docker image prune.	Removes unused images.
docker image pull.	Pulls an image or a repository from a registry.
docker image push.	Pushes an image or a repository to a registry.
docker image rm.	Removes one or more images.
docker image save.	Saves one or more images to a tar archive (streamed to STDOUT by default).
docker image tag.	Creates a tag TARGET_IMAGE that refers to SOURCE_IMAGE.
docker image history.	Shows the history of an image, including changes made to it and its layers.

docker update.	Enables a user to update the configuration of containers.
docker tag.	Creates a tag, such as TARGET_IMAGE, which enables users to group and organize container images.
docker search.	Looks in Docker Hub for whatever the user needs.
docker save.	Enables a user to save images to an archive.
docker compose.	Used to handle an environment variable.

Command Options

Option	Short	Default	Description
--all	-a		Show all images (default hides intermediate images)
--digests			Show digests
--filter	-f		Filter output based on conditions provided
--format			Format output using a custom template: 'table': Print output in table format with column headers (default) 'table TEMPLATE': Print output in table format using the given Go template 'Json': Print in JSON format 'TEMPLATE': Print output using the given Go template. Refer to https://docs.docker.com/go/formatting/ for more information about formatting output with templates
--no-trunc			Don't truncate output
--quiet	-q		Only show image IDs

Docker Containers

- Docker containers are lightweight, standalone, and executable packages that encapsulate an application along with its dependencies, libraries, and runtime environment.
- They are created from Docker images and act as isolated, runnable instances of an application.
- Containers are designed to be consistent across different environments, ensuring that an application behaves the same way in development, testing, and production environments.
- Docker containers leverage containerization technology to provide process isolation, allowing multiple containers to run on the same host without interfering with each other.
- Each container has its own filesystem, network stack, and isolated process space, making them efficient, portable, and secure.
- Containers are often compared to virtual machines (VMs), but they are more lightweight and share the host OS kernel, reducing overhead and resource usage.
- Containers can be started, stopped, and removed quickly, making them ideal for scaling applications up or down in response to changing demand.
- Docker containers can be orchestrated and managed using tools like Docker Compose, Kubernetes, and Docker Swarm, enabling the deployment and scaling of containerized applications across clusters of hosts.
- They can be used for various purposes, from running microservices, web applications, and databases to batch processing and continuous integration workflows.
- Containerization technology and Docker have revolutionized software development and deployment by simplifying the process of building, shipping, and running applications in a consistent and reproducible manner.

Basic Docker Container Commands

Command	Options	Description
docker run	-d, --detach	Run a container in the background.
	-it, --interactive	Run a container interactively (attach to it).
	--name	Assign a name to the container.
	--rm	Automatically remove the container when it exits.
	-p, --publish	Publish container ports to the host.
	-v, --volume	Mount a host directory as a volume inside the container.
docker stop		Stop a running container.
docker start		Start a stopped container.
docker restart		Restart a running or stopped container.
docker pause		Pause a running container.
docker unpause		Unpause a paused container.
docker exec	-it, --interactive	Run a command inside a running container.
docker attach		Attach to a running container's standard input, output, and error.
docker logs	-f, --follow	Stream container logs in realtime.
--tail	Show the last N lines of container logs.	

docker ps	-a, --all	List all containers (including stopped ones).
-q, --quiet	Only display container IDs.	
docker top		Display running processes inside a container.
docker inspect		View detailed information about a container, including configuration.
docker stats		Display real-time resource usage statistics of a running container.
docker rm	-f, --force	Remove one or more containers.
docker kill		Send a signal to a running container to stop it forcefully.
docker rename		Rename a container.
docker wait		Block until a container stops, then print its exit code.

Docker Services

- Docker services are a higher-level abstraction used in Docker Swarm mode to manage and deploy applications as a set of interconnected containers across a cluster of Docker nodes. They enable the orchestration and scaling of containerized applications in a distributed environment.
- Docker services provide a blueprint for defining how an application should run within a Swarm cluster. They include instructions for creating and managing a group of related containers, making it easy to define complex multi-container applications.
- When a user creates a Docker service, it represents one or more instances of containers running the same application. These instances are collectively managed as a service, making it simple to scale and update the application as needed.
- Docker services allow you to specify the desired number of replicas for an application. The Swarm orchestrator ensures that the specified number of containers is running, distributing them across available cluster nodes for load balancing.
- Services in Docker Swarm are designed for high availability. If a container or node fails, Swarm automatically reschedules the task to a healthy node to maintain the desired replica count.
- Each service gets its own unique DNS name, making it easy for services to communicate with each other using this DNS-based discovery. Docker services can be attached to custom overlay networks to provide network isolation.
- Docker services support environment variables, secrets, and configuration files, allowing for the dynamic configuration of containers. This ensures that containers adapt to different environments seamlessly.
- Docker services support rolling updates, allowing for controlled and zero-downtime application updates. If an update fails or needs to be rolled back, Swarm provides mechanisms to revert to the previous service version.
- Docker services facilitate centralized logging and monitoring, making it easier to aggregate and analyze container logs and performance metrics.
- Docker services offer built-in service discovery and load balancing. They automatically distribute incoming traffic among the containers providing the service.
- Users can define constraints, such as node labels or affinity rules, to influence the placement of service tasks on specific nodes within the cluster.

- Docker services can define health checks, ensuring that unhealthy containers are automatically replaced to maintain service availability.
- Docker services can be defined as part of a Docker Compose file, making it straightforward to manage multi-service applications using a single configuration file.
- Docker provides a comprehensive command-line interface (CLI) and REST API for managing Docker services, allowing both manual and programmatic control over service creation and management.
- Docker services can pull images from container registries like Docker Hub, private registries, or local repositories.

Basic Docker Services Commands

Command	Options	Description
docker service create	-d, --detach	Run the service in the background.
	--name	Assign a name to the service.
	--replicas	Set the number of desired replicas for the service.
	--publish	Publish a service's ports to the host.
	-e, --env	Set environment variables for the service.
docker service scale		Scale a service to the specified number of replicas.
docker service ls		List all services in the swarm.
docker service ps		List tasks (containers) of a service.
docker service inspect		Display detailed information about a service, including its configuration.
docker service update	--replicas	Update the number of replicas for a service.
	--publish-add	Add a new published port to the service.
	--publish-rm	Remove a published port from the service.
	-e, --env-add	Add environment variables to the service.
	-e, --env-rm	Remove environment variables from the service.

docker service rm		Remove a service from the swarm.
docker service logs	-f, --follow	Stream logs of tasks (containers) in a service.
	--tail	Show the last N lines of service logs.
docker service scale		Scale a service to the specified number of replicas.
docker service rollback		Roll back to a previous service version.
docker service pause		Pause a running service.
docker service unpause		Unpause a paused service.
docker service tasks		List tasks (containers) of a service.
docker service update	--update-delay	Set the delay between updates of a service.
--update-parallelism	Set the maximum number of tasks updated simultaneously.	
--update-failure-action	Set the action to take on update failure.	
docker service inspect	--pretty	Display formatted and human-readable output.
docker service create	--constraint	Set node placement constraints for a service.
	--mount	Mount a volume to a service.
	--network	Attach a service to a specific network.

Docker Images vs Docker Containers vs Docker Services

Feature	Docker Images	Docker Containers	Docker Services
Definition	Static, immutable templates for containers.	Runnable instances of images.	Abstraction for deploying containerized apps.
Contents	Contains application code, libraries, etc.	Running instances of images with state.	Defines how containers should run and scale.
Execution	Not directly executable.	Executable and runnable instances.	Orchestrates and manages containers.
Modifications	Immutable; changes require a new image.	Mutable; changes can be made to a running container.	Dynamic scaling, rolling updates, etc.
Lifespan	Persistent, exists until deleted.	Short-lived, created and destroyed as needed.	Long-lived, manages application uptime.
Scaling	Not designed for horizontal scaling.	Single instance per container.	Designed for horizontal scaling.
High Availability	Not designed for high availability.	No built-in high availability mechanisms.	Built-in support for high availability.
Networking	No network configuration by default.	Can be attached to networks for communication.	Attached to overlay networks for service discovery.
Configuration	Configuration stored within image.	Configuration applied when starting the container.	Dynamic configuration updates supported.
Load Balancing	No built-in load balancing.	No built-in load balancing.	Built-in load balancing for service tasks.

Logs and Monitoring	Logs captured within images.	Logs captured during container runtime.	Centralized logging and monitoring.

Hypervisors vs Containers

- Hypervisor
  A hypervisor, also known as a virtual machine monitor (VMM), is a software or hardware technology that enables the creation and management of multiple virtual machines (VMs) on a single physical host server.
- Container
  A container provides a consistent and isolated environment sharing the host OS's kernel for running applications, allowing developers to package an application along with everything it needs to run, from the code to the system libraries, into a single unit. They can be easily deployed, ensuring scalability.

Here are few differences between both:

Feature	**Hypervisor**	**Containers**
Isolation	Provides hardware-level virtualization.	Provides OS-level virtualization.
Overhead	Typically has higher resource overhead.	Generally, has lower resource overhead.
Performance	Slightly lower performance due to emulation.	High performance as there's no emulation.
Boot Time	Slower boot time as it emulates a full OS.	Extremely fast startup since it shares the host OS.
Portability	Less portable, as VMs may have specific OS and driver requirements.	Highly portable; containers are OS-agnostic.