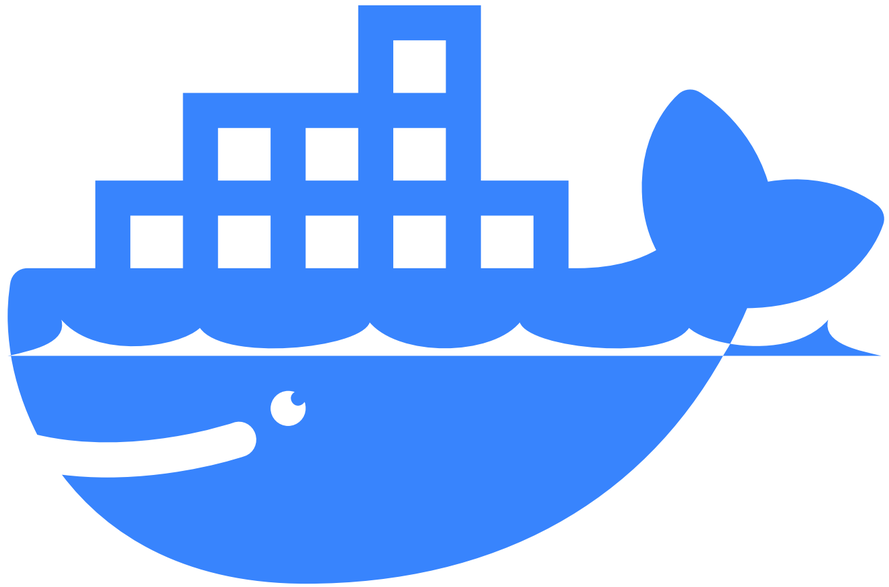

DOCKER
DEEP DIVE
HANDS-ON
LIVE

What is Docker?

Docker is an open-source platform used for developing, shipping, and running applications in isolated environments called containers. These containers allow for easy deployment and scaling of applications across different computing environments.

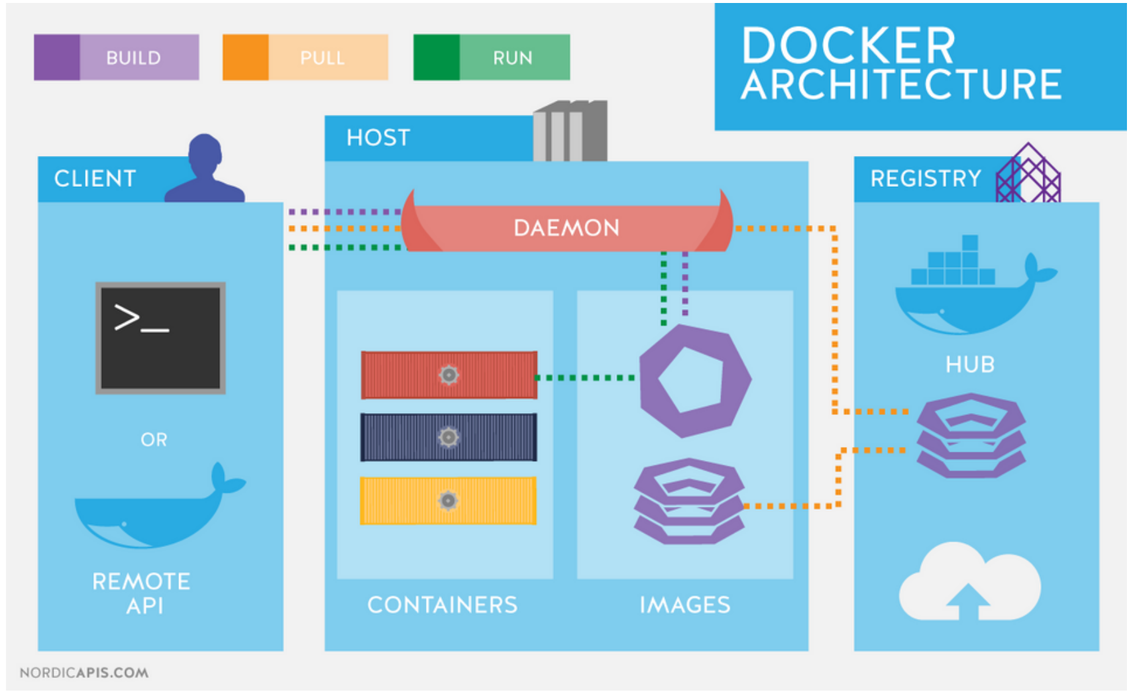


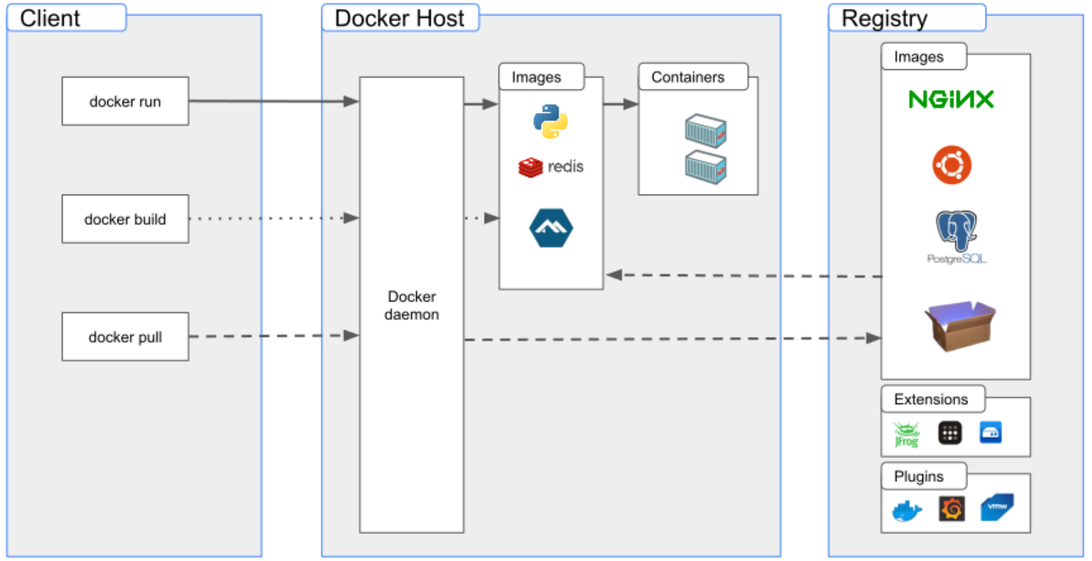
@Sandip Das

Docker Architecture

The Docker architecture is designed as a client-server architecture. The client, known as the Docker client, is a command-line tool that allows users to interact with the Docker daemon, which is the Docker server responsible for managing Docker images, containers, networks, and volumes. Here's a breakdown of the Docker architecture:

1. **Docker Client**: The Docker client is a command-line interface (CLI) tool that users use to interact with Docker. The client sends commands to the Docker daemon and receives responses back. The Docker client can be installed on the same machine as the Docker daemon, or it can be installed on a remote machine and connected to the Docker daemon over the network.
2. **Docker Daemon**: The Docker daemon is the server component of Docker that manages Docker images, containers, networks, and volumes. The daemon runs in the background and listens for API requests from the Docker client. It's responsible for starting, stopping, and managing Docker containers, creating and managing Docker images, and managing Docker networks and volumes.
3. **Docker Registry:** A Docker registry is a repository that stores Docker images. Docker Hub is the default public registry provided by Docker, where users can store, share, and download Docker images. However, users can also set up their own private Docker registry for storing and sharing images within their organization.
4. **Docker Images:** A Docker image is a read-only template that contains the application and its dependencies. Docker images are built using a Dockerfile, which is a text file that contains instructions for building the image. Once built, Docker images can be stored in a registry and used to create Docker containers.
5. **Docker Containers:** A Docker container is a running instance of a Docker image. Containers are isolated from each other and from the host system, which provides a secure environment for running applications. Docker containers can be started, stopped, and managed using the Docker daemon.
6. **Docker Networks**: Docker networks are used to connect Docker containers to each other and to the outside world. Docker provides several types of network drivers, including bridge, host, overlay, and macvlan, to support different networking scenarios.
7. **Docker Volumes**: Docker volumes are used to persist data generated by Docker containers. Docker volumes can be mounted into a container at runtime, allowing data to be shared and persisted across multiple containers.





Docker Client


@Sandip Das

The Docker client is a command-line interface (CLI) tool that allows users to interact with the Docker daemon, which is the Docker server responsible for managing Docker images, containers, networks, and volumes.

The Docker client communicates with the Docker daemon through the Docker API, which allows users to execute Docker commands from the command line or from scripts. The Docker client can be installed on the same machine as the Docker daemon, or it can be installed on a remote machine and connected to the Docker daemon over the network.

The Docker client provides a wide range of commands for managing Docker images, containers, networks, and volumes. Here are some of the most common commands:

- **docker run**: Creates and starts a new Docker container from a Docker image.
- **docker build**: Builds a new Docker image from a Dockerfile.
- **docker pull**: Downloads a Docker image from a Docker registry.
- **docker push**: Uploads a Docker image to a Docker registry.
- **docker ps**: Lists all running Docker containers.
- **docker stop**: Stops a running Docker container.
- **docker rm**: Removes a stopped Docker container.
- **docker network**: Manages Docker networks.
- **docker volume**: Manages Docker volumes.

In addition to the command-line interface, Docker also provides a graphical user interface (GUI) called Docker Desktop, which provides a more user-friendly interface for managing Docker resources on a local machine. The Docker Desktop GUI is available for Windows and macOS.

Docker daemon

The Docker daemon is the server component of Docker that manages Docker images, containers, networks, and volumes. The daemon runs in the background and listens for API requests from the Docker client. It's responsible for starting, stopping, and managing Docker containers, creating and managing Docker images, and managing Docker networks and volumes.

The Docker daemon is responsible for the following tasks:

1. **Image management:** The Docker daemon manages Docker images, which are used to create Docker containers. It downloads images from a Docker registry, such as Docker Hub, and stores them locally on the Docker host.
2. **Container management:** The Docker daemon is responsible for starting, stopping, and managing Docker containers. It creates a new container from a Docker image when the user executes the "docker run" command, and it stops and removes the container when the user executes the "docker stop" or "docker rm" command.
3. **Networking:** The Docker daemon manages Docker networks, which are used to connect Docker containers to each other and to the outside world. It provides several types of network drivers, including bridge, host, overlay, and macvlan, to support different networking scenarios.
4. **Volume management:** The Docker daemon manages Docker volumes, which are used to persist data generated by Docker containers. Docker volumes can be mounted into a container at runtime, allowing data to be shared and persisted across multiple containers.
5. **API**: The Docker daemon exposes a REST API that allows users to interact with Docker programmatically. This API is used by the Docker client to send requests to the Docker daemon.

The Docker daemon runs as a background process on the Docker host and listens on a Unix socket or a network port for API requests from the Docker client. The Docker daemon can be configured using various options, including storage drivers, network drivers, and logging options, to customize its behavior.

@Sandip Das

Docker Registry

A Docker registry is a repository that stores Docker images. Docker Hub is the default public registry provided by Docker, where users can store, share, and download Docker images. However, users can also set up their own private Docker registry for storing and sharing images within their organization.

The Docker registry provides the following features:

1. **Image storage**: The Docker registry stores Docker images, which can be downloaded and used to create Docker containers.
2. Image management: The Docker registry allows users to manage Docker images, including tagging, pushing, and pulling images from the registry.
3. **Access control**: The Docker registry allows users to control access to Docker images by configuring authentication and authorization policies.
4. **Replication**: The Docker registry supports replication of Docker images across multiple servers, allowing users to distribute images to different geographic locations for faster downloads.
5. **Search**: The Docker registry provides a search functionality that allows users to search for Docker images based on keywords and tags.

Users can use the Docker CLI to interact with Docker registries. For example, the "docker pull" command downloads a Docker image from a registry, and the "docker push" command uploads a Docker image to a registry. Docker also provides an open-source registry implementation called Docker Distribution, which can be used to set up a private Docker registry. Additionally, users can use third-party registry solutions, such as JFrog Artifactory and Google Container Registry, to host their Docker images.


@Sandip Das

@Sandip Das

Docker Images

Docker images are the building blocks of Docker containers. An image is a read-only template that contains the application and its dependencies. Docker images can be built using a Dockerfile, which is a text file that contains instructions for building the image. Once built, Docker images can be stored in a registry and used to create Docker containers.

Here are some key features of Docker images:

1. **Layered architecture**: Docker images are built using a layered architecture. Each layer in the image represents a change or modification to the previous layer. This layered architecture allows Docker to reuse layers across multiple images, reducing the size of images and improving build times.
2. **Versioning**: Docker images can be versioned using tags. A tag is a label that is applied to an image, indicating the version of the image. Users can use tags to identify and manage different versions of an image.
3. **Caching**: Docker images are cached locally on the Docker host, which allows Docker to reuse images that have already been built. This caching mechanism can speed up build times and reduce network traffic.
4. **Portability**: Docker images are portable, which means they can be easily moved between different Docker hosts and environments. This portability is achieved through the use of a standardized image format and the Docker registry.
5. **Security**: Docker images can be scanned for vulnerabilities using tools like Docker Security Scanning. This allows users to identify and address security issues in their Docker images.

Users can use the Docker CLI to interact with Docker images. For example, the "docker build" command builds a Docker image from a Dockerfile, and the "docker push" command uploads a Docker image to a Docker registry. Additionally, Docker provides a public registry called Docker Hub, where users can store, share, and download Docker images. Users can also set up their own private Docker registry to store and share images within their organization.

Here are some examples of Docker CLI commands that can be used to manage Docker images:

List local images:

docker images

This command lists all the Docker images that are currently stored locally on the Docker host.

Pull an image from a registry:

docker pull <image-name>

This command downloads a Docker image from a Docker registry, such as Docker Hub, and stores it locally on the Docker host.

Build an image from a Dockerfile:

docker build -t <image-name> <path-to-dockerfile>

This command builds a Docker image from a Dockerfile located at the specified path and gives it the specified name (-t option).

Tag an image:

docker tag <image-id> <new-image-name>:<tag>

This command applies a new tag to an existing Docker image. The tag is used to identify different versions of the same image.

Push an image to a registry:

docker push <image-name>

This command uploads a Docker image to a Docker registry, such as Docker Hub.

Remove an image:

docker rmi <image-name>

This command removes a Docker image from the local Docker host.

Search for an image:

docker search <keyword>

This command searches for Docker images on Docker Hub based on the specified keyword.

Show detailed information about an image:

docker inspect <image-name>

This command shows detailed information about a Docker image, including its metadata, configuration, and layers.

Dockerfile


@Sandip Das

A Dockerfile is a text file that contains a set of instructions for building a Docker image. The instructions in a Dockerfile are executed in order to create a Docker image that can be used to run a containerized application.

Here are some key components of a Dockerfile:

1. **Base image**: The base image is the starting point for the Docker image. It provides the operating system and basic set of tools and libraries that the application needs to run. For example, to create a Docker image for a Python application, the base image might be the official Python image from Docker Hub.
2. **Environment variables**: Environment variables can be set in the Dockerfile to configure the application environment. For example, environment variables can be used to set the port number that the application listens on or to specify the database connection string.
3. **Copying files**: Files can be copied from the host machine to the Docker image using the "COPY" or "ADD" commands. This is used to include the application code and any necessary configuration files in the Docker image.
4. **Running commands**: Commands can be executed in the Dockerfile to install dependencies, compile code, and configure the application. For example, to install Python dependencies, the "RUN pip install" command can be used.
5. **Exposing ports**: Ports can be exposed in the Dockerfile using the "EXPOSE" command. This tells Docker to expose the specified port when the container is run.
6. **Entrypoint**: The entrypoint is the command that is executed when the container is run. This can be specified in the Dockerfile using the "CMD" or "ENTRYPOINT" commands. For example, to run a Python application, the entrypoint might be the command "python app.py".

Example:

```
FROM python:3.8
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
CMD [ "python", "app.py" ]
```

Here are the most commonly used Dockerfile commands and their purposes:

1. **FROM**: Specifies the base image for the Docker image being built. The base image must be the first instruction in the Dockerfile. For example, **FROM python:3.8** specifies that the base image is the official Python 3.8 image.
2. **RUN**: Runs a command in the Docker image. This command can be used to install packages, update the system, or build the application. For example, **RUN apt-get update && apt-get install -y some-package** installs the "some-package" package using the "apt-get" package manager.
3. **COPY** and **ADD**: Copies files from the host machine to the Docker image. **COPY** is preferred over **ADD** because it has fewer side effects. For example, **COPY app.py /app/** copies the "app.py" file from the host machine to the "/app" directory in the Docker image.
4. **WORKDIR**: Sets the working directory for subsequent instructions in the Dockerfile. For example, **WORKDIR /app** sets the working directory to "/app".
5. **ENV**: Sets environment variables in the Docker image. For example, **ENV PORT 8000** sets the "PORT" environment variable to 8000.
6. **EXPOSE**: Exposes a port from the Docker image to the host machine. For example, **EXPOSE 8000** exposes port 8000 in the Docker image.
7. **CMD** and **ENTRYPOINT**: Specifies the command that is run when the Docker image is started as a container. **CMD** is used to provide default arguments to the command, while **ENTRYPOINT** is used to specify the main command. For example, **CMD ["python", "app.py"]** specifies that the "app.py" file should be run with the Python interpreter.
8. **LABEL**: Adds metadata to the Docker image. This metadata can be used to provide information about the image, such as the version, maintainer, or source code repository. For example, **LABEL version="1.0"** adds a "version" label to the Docker image with the value "1.0".
9. **USER**: Sets the user that the Docker image should run as. For example, **USER appuser** sets the user to "appuser".

These are just some of the most commonly used Dockerfile commands. There are many other commands available, such as **ARG**, **VOLUME**, **HEALTHCHECK**, and **STOPSIGNAL**, that can be used to further customize the Docker image.

Docker Containers

@Sandip Das

Docker containers are running instances of Docker images. A container is a lightweight, portable, and isolated environment that can run an application and its dependencies consistently across different systems, including development, testing, and production environments.

Here are some key features of Docker containers:

1. **Isolation**: Docker containers are isolated from each other and from the host system. Each container has its own file system, network, and process namespace, which provides a secure environment for running applications.
2. **Lightweight**: Docker containers are lightweight because they share the same kernel as the host system. This means that Docker containers require fewer resources than traditional virtual machines.
3. **Portability**: Docker containers are portable, which means they can be easily moved between different Docker hosts and environments. This portability is achieved through the use of a standardized container format and the Docker runtime.
4. **Reproducibility**: Docker containers provide reproducible builds because they are built from a Docker image, which contains all the application's dependencies. This means that Docker containers can be reliably built and deployed across different systems.
5. **Scalability**: Docker containers can be scaled horizontally by running multiple instances of the same container across different systems. This allows applications to handle increased traffic and load.

Users can use the Docker CLI to interact with Docker containers. Here are some examples of Docker CLI commands that can be used to manage Docker containers:

List running containers:

docker ps

This command lists all the Docker containers that are currently running on the Docker host.

Start a container:

docker run <image-name>

This command starts a new Docker container from the specified Docker image.

Stop a container:

docker stop <container-id>

This command stops a running Docker container.

Remove a container:

docker rm <container-id>

This command removes a stopped Docker container from the Docker host.

View logs from a container:

docker logs <container-id>

This command shows the logs generated by a Docker container.

Connect to a running container:

docker exec -it <container-id> /bin/bash

This command connects to a running Docker container and opens a terminal session inside the container.

Docker Networking

Docker networks are used to connect Docker containers to each other and to the outside world. Docker provides several types of network drivers, including bridge, host, overlay, and macvlan, to support different networking scenarios.

Here are some key features of Docker networks:

1. **Isolation**: Docker networks provide isolation between different Docker containers. Each container can be assigned to one or more Docker networks, and communication between containers is only allowed within the same network.
2. **Flexibility**: Docker networks provide flexibility in how containers communicate with each other. For example, containers can be connected to multiple networks, and different types of network drivers can be used to support different network topologies.
3. **Security**: Docker networks provide security by allowing users to control which containers can communicate with each other. This can be achieved by using network policies and access controls.
4. **Scalability**: Docker networks provide scalability by allowing users to create multiple containers that can communicate with each other. This allows applications to handle increased traffic and load.

@Sandip Das

Users can use the Docker CLI to interact with Docker networks. Here are some examples of Docker CLI commands that can be used to manage Docker networks:

List Docker networks:

docker network ls

This command lists all the Docker networks that are currently available on the Docker host.

Create a new Docker network:

docker network create <network-name>

This command creates a new Docker network with the specified name.

Connect a container to a Docker network:

docker network connect <network-name> <container-name>

This command connects a Docker container to the specified Docker network.

Disconnect a container from a Docker network:

docker network disconnect <network-name> <container-name>

This command disconnects a Docker container from the specified Docker network.

Inspect a Docker network:

docker network inspect <network-name>

This command shows detailed information about the specified Docker network, including its configuration and connected containers.

Remove a Docker network:

docker network rm <network-name>

This command removes the specified Docker network from the Docker host.

Docker Networking Modes


@Sandip Das

Docker provides several network modes that can be used to connect Docker containers to each other and to the outside world. Each network mode provides a different level of isolation and network connectivity.

Here are the four main Docker network modes:

1. **Bridge network mode**: The bridge network mode is the default network mode for Docker containers. Each container is assigned an IP address on a private network, and communication between containers is only allowed within the same network. Containers can access the outside world through the Docker host's network interface.
2. **Host network mode**: In the host network mode, Docker containers share the same network namespace as the Docker host. This means that containers are assigned the same IP address as the host, and they can access the network resources directly without any network address translation (NAT).
3. **Overlay network mode**: The overlay network mode is used to connect Docker containers across multiple Docker hosts. This mode uses a software-defined network (SDN) to create a virtual network that spans across multiple hosts, allowing containers to communicate with each other regardless of their physical location.
4. **Macvlan network mode**: The macvlan network mode allows Docker containers to be directly connected to the physical network, which allows them to be assigned IP addresses on the same subnet as the host. This mode is useful for applications that require direct access to the physical network, such as network monitoring or packet sniffing tools.

Users can specify the network mode when running a Docker container using the "docker run" command. For example, to run a container in bridge network mode, you can use the following command:

```
docker run --network=bridge <image-name>
```

Additionally, users can create custom Docker networks with specific configurations, such as IP address range and subnet mask, to provide more control over container networking.

Docker Networking Tools

Docker provides several networking tools that can be used to manage Docker networks and troubleshoot network connectivity issues. Here are some of the most commonly used Docker networking tools:

1. **Docker Compose**: Docker Compose is a tool for defining and running multi-container Docker applications. It allows users to define the network configuration for their containers, including networks, IP addresses, and port mappings, in a single YAML file. Docker Compose can be used to start, stop, and manage multiple containers at once.
2. **Docker Network CLI:** The Docker Network CLI is a command-line tool for managing Docker networks. It allows users to create, list, inspect, and delete Docker networks. Additionally, it can be used to connect and disconnect Docker containers from networks, and to configure network policies and access controls.
3. **Docker Network Inspector**: The Docker Network Inspector is a command-line tool for troubleshooting network connectivity issues in Docker containers. It allows users to inspect the network configuration of a Docker container, including IP addresses, network interfaces, and routing tables. Additionally, it can be used to perform network diagnostics, such as pinging or tracing network routes.
4. **Weave Scope**: Weave Scope is a tool for visualizing and monitoring Docker networks. It provides a real-time map of the Docker network topology, including containers, hosts, and network connections. Additionally, it can be used to monitor network traffic, troubleshoot network issues, and identify security threats.
5. **Calico**: Calico is a network plugin for Docker that provides advanced networking features, such as policy-based network segmentation, network isolation, and network encryption. Calico can be used to deploy and manage large-scale Docker environments, such as Kubernetes clusters, and to secure containerized applications.

These tools can be used in conjunction with each other to manage and troubleshoot Docker networks. Additionally, there are many third-party tools available that can be used to extend the functionality of Docker networking, such as CNI plugins and network security solutions.

@Sandip Das

Docker DNS

Docker DNS (Domain Name System) is used to resolve domain names to IP addresses within Docker containers. By default, Docker containers use the DNS server provided by the Docker host. The DNS server is responsible for resolving domain names to IP addresses, and it can be configured to use different DNS servers or to forward DNS requests to another DNS server.

Here are some key features of Docker DNS:

1. **Default DNS server**: By default, Docker containers use the DNS server provided by the Docker host. This allows containers to resolve domain names to IP addresses without the need for additional configuration.
2. **Custom DNS servers**: Docker allows users to configure custom DNS servers for their containers. This can be useful for applications that require specific DNS servers, such as those used in a corporate network.
3. **DNS caching:** Docker caches DNS records locally on the Docker host to improve performance and reduce network traffic. The DNS cache is periodically refreshed to ensure that it contains up-to-date information.
4. **DNS round-robin**: Docker uses DNS round-robin to load balance network requests across multiple containers. This means that multiple containers can be assigned the same DNS name, and Docker will automatically distribute network requests across them.
5. **Network aliases**: Docker allows users to assign network aliases to containers, which can be used to map multiple domain names to the same container. This can be useful for applications that require multiple domain names to be associated with the same container.

Users can configure Docker DNS by specifying the --dns option when running a Docker container. For example, to use a custom DNS server, you can use the following command:

```
docker run --dns=<dns-server> <image-name>
```

Additionally, Docker provides a built-in DNS server, called Docker DNS, that can be used to provide DNS resolution for Docker containers. Docker DNS is automatically configured when a Docker network is created, and it can be used to provide DNS resolution for containers within the same network.

@Sandip Das

Docker Volumes

Docker volumes are used to persist data generated by Docker containers. Docker volumes can be mounted into a container at runtime, allowing data to be shared and persisted across multiple containers.

Here are some key features of Docker volumes:

1. **Persistence**: Docker volumes provide persistence by allowing data to be stored outside of the container's file system. This means that data can be preserved even if the container is deleted or recreated.
2. **Isolation**: Docker volumes provide isolation by allowing data to be shared between containers without requiring the containers to be on the same Docker network or Docker host.
3. **Flexibility**: Docker volumes provide flexibility by allowing users to choose where the data is stored and how it is accessed. For example, volumes can be stored locally on the Docker host, or they can be stored on a remote storage system like NFS or AWS EBS.
4. **Security**: Docker volumes provide security by allowing users to control access to the data stored in the volume. This can be achieved by using file system permissions or access controls.
5. **Scalability**: Docker volumes provide scalability by allowing data to be shared across multiple containers. This allows applications to handle increased traffic and load.

Users can use the Docker CLI to interact with Docker volumes. Here are some examples of Docker CLI commands that can be used to manage Docker volumes:

List Docker volumes:

docker volume ls

This command lists all the Docker volumes that are currently available on the Docker host.

Create a new Docker volume:

docker volume create <volume-name>

This command creates a new Docker volume with the specified name.

Attach a Docker volume to a container:

docker run -v <volume-name>:<mount-point> <image-name>

This command starts a new Docker container from the specified Docker image and attaches the specified Docker volume to the container.

Inspect a Docker volume:

docker volume inspect <volume-name>

This command shows detailed information about the specified Docker volume, including its configuration and attached containers.

Remove a Docker volume:

docker volume rm <volume-name>

This command removes the specified Docker volume from the Docker host.

@Sandip Das

Docker Volumes and Bind Mounts

Docker provides two ways to persist data generated by Docker containers: volumes and bind mounts. Both volumes and bind mounts allow data to be shared and persisted across multiple containers, but they work in slightly different ways.
Here are the key differences between Docker volumes and bind mounts:

1. Storage location: Docker volumes are stored in a Docker-managed volume storage area, which can be located on the Docker host or in a remote storage system, such as AWS EBS or NFS. Bind mounts, on the other hand, can be located anywhere on the Docker host file system, and they are not managed by Docker.
2. Persistence: Docker volumes are persistent, meaning that they can be preserved even if the container is deleted or recreated. Bind mounts are not persistent, meaning that they are tied to the underlying file system, and they are removed if the host file system is deleted or recreated.
3. Sharing: Docker volumes can be shared across multiple containers, allowing data to be accessed and modified by multiple containers at the same time. Bind mounts can also be shared across multiple containers, but they can cause conflicts if multiple containers write to the same file at the same time.
4. Access control: Docker volumes can be managed with access control lists (ACLs) to provide fine-grained control over who can access the data stored in the volume. Bind mounts do not have built-in access control features and rely on the underlying file system permissions for access control.
5. Ease of use: Docker volumes are easy to use and manage, and they can be created, deleted, and backed up using simple Docker commands. Bind mounts require more manual configuration, and they can be difficult to manage in large-scale deployments.

Here are some examples of how to use Docker volumes and bind mounts:

Create a Docker volume:

docker volume create <volume-name>

This command creates a new Docker volume with the specified name.

Create a Docker container with a volume:

docker run -v <volume-name>:<mount-point> <image-name>

This command starts a new Docker container from the specified Docker image and attaches the specified Docker volume to the container.

Create a bind mount:

docker run -v /host/folder:/container/folder <image-name>

This command starts a new Docker container from the specified Docker image and mounts the specified host folder to the container's file system.

Remove a Docker volume:

docker volume rm <volume-name>

This command removes the specified Docker volume from the Docker host.
Remove a bind mount: To remove a bind mount, simply delete the file or folder that was mounted to the container.

@Sandip Das

Docker BuildKit

Docker BuildKit is a tool for building Docker images that provides improved performance and security over the traditional Docker build process. BuildKit uses a new build engine that is designed to be more efficient and flexible than the old engine.

Here are some key features of Docker BuildKit:

1. Parallelism: BuildKit allows for parallel building of Docker images, which can significantly improve build times for large images. This is achieved through the use of a concurrent build graph, which enables multiple dependencies to be built simultaneously.
2. Cache management: BuildKit provides improved cache management for Docker builds, which reduces the amount of time required to rebuild images. BuildKit can cache individual layers of an image, allowing for incremental builds that only rebuild what has changed.
3. Security: BuildKit provides improved security for Docker builds, by isolating the build process from the host system. This is achieved through the use of user namespaces, which create a separate user and group ID space for each build, preventing privilege escalation attacks.
4. Extensibility: BuildKit is highly extensible, with a modular architecture that allows for the addition of new build components and customization of the build process. BuildKit also supports the use of external tools and plugins, which can be used to add new functionality to the build process.
5. Dockerfile syntax: BuildKit supports a new Dockerfile syntax that provides additional functionality and flexibility over the traditional Dockerfile syntax. The new syntax includes support for multi-stage builds, build-time variables, and build-time secrets.

To use BuildKit, users must first enable it by setting the DOCKER_BUILDKIT environment variable to 1. This can be done using the following command:

```
export DOCKER_BUILDKIT=1
```

Once BuildKit is enabled, users can use the standard Docker build command to build their images. However, to take full advantage of BuildKit's features, users should use the new Dockerfile syntax and enable caching and parallelism. Here is an example command for building a Docker image with BuildKit:

```
docker build --progress=plain --no-cache --build-arg VERSION=1.0 -t my-image:latest .
```

This command builds a Docker image with the name "my-image:latest", using the Dockerfile in the current directory, with caching and parallelism enabled, and a build-time argument of "VERSION=1.0". The "--progress=plain" option is used to disable the new BuildKit progress bar, which can cause issues in some environments.

@Sandip Das

Docker Build Target Architecture

Docker allows users to target specific architectures when building Docker images. This is useful for creating images that can run on different platforms, such as ARM or x86, without the need to maintain multiple images.

Docker supports a wide range of architectures, including:

1. AMD64/x86_64: This is the most common architecture and is used by most modern desktop and server processors.
2. ARMv6: This architecture is used by older Raspberry Pi models, such as the Raspberry Pi 1 and Raspberry Pi Zero.
3. ARMv7: This architecture is used by newer Raspberry Pi models, such as the Raspberry Pi 2 and Raspberry Pi 3.
4. ARMv8/AArch64: This architecture is used by newer 64-bit ARM processors, such as the Raspberry Pi 4 and the Amazon AWS Graviton processors.
5. PowerPC: This architecture is used by IBM PowerPC processors and is commonly found in high-performance computing environments.
6. IBM Z: This architecture is used by IBM mainframes and is commonly used in enterprise environments.
7. s390x: This architecture is used by IBM System z mainframes and is commonly used in enterprise environments.

To build Docker images for different architectures, users can use the "docker buildx" command, which is part of the Docker CLI. The "--platform" option can be used to specify the target architecture when building the Docker image. Here are some example commands for building Docker images for different architectures:

AMD64/x86_64: `docker buildx build --platform linux/amd64 -t my-image .`
ARMv6: `docker buildx build --platform linux/arm/v6 -t my-image .`
ARMv7: `docker buildx build --platform linux/arm/v7 -t my-image .`
ARMv8/AArch64: `docker buildx build --platform linux/arm64 -t my-image .`
PowerPC: `docker buildx build --platform linux/ppc64le -t my-image .`
IBM Z: `docker buildx build --platform linux/s390x -t my-image .`
Multi-architecture image: `docker buildx build --platform linux/amd64,linux/arm64 -t my-image .`

Test the Docker image:
Once the Docker image has been built, it can be tested on the target platform to ensure that it runs correctly. This can be done using a Docker container with the same target platform. For example, to run the Docker image on an ARM64 platform, you can use the following command:

```
docker run --platform linux/arm64 my-image
```

@Sandip Das

Multi-Stage Docker Builds

Multi-stage Docker builds allow users to create Docker images that are optimized for production environments while still using a single Dockerfile. Multi-stage builds allow users to compile code, install dependencies, and generate artifacts in one stage and then copy only the necessary files to the final stage. This results in smaller and more efficient Docker images that are optimized for production use.

Here are the steps to create a multi-stage Docker build:

Define multiple stages in the Dockerfile: To create a multi-stage Docker build, users must define multiple stages in the Dockerfile. Each stage can include its own set of commands and dependencies. For example, here is a **Dockerfile** that uses two stages: one for compiling code and generating artifacts and another for running the application:

```
# Stage 1: Install dependencies and build the application
FROM python:3.8-slim-buster AS builder
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
RUN python setup.py bdist_wheel

# Stage 2: Run the application
FROM python:3.8-slim-buster
WORKDIR /app
COPY --from=builder /app/dist/*.whl .
RUN pip install --no-cache-dir *.whl
CMD [ "python", "-m", "my_app" ]
```

In this example, the first stage installs the Python dependencies and builds the application into a wheel file. The second stage installs the wheel file and runs the application.

Build the Docker image: Once the Dockerfile has been defined, users can build the Docker image using the standard "docker build" command. For example, to build the Docker image with the name "my-image", users can use the following command:

```
docker build -t my-python-app .
```

RUN the container:

```
docker run -p 8000:8000 my-python-app
```

@Sandip Das

Docker Logging and Monitoring

@Sandip Das

Docker logging and monitoring are essential components of managing and maintaining Docker containers and applications. Docker provides built-in logging and monitoring capabilities, as well as third-party tools that can be used to manage and monitor Docker containers and applications.
Here are some key components of Docker logging and monitoring:

1. **Docker logging drivers**: Docker provides a variety of logging drivers that can be used to manage and store container logs. These drivers include the built-in "json-file" driver, which stores logs as JSON files on the Docker host, as well as drivers for forwarding logs to third-party services such as Elasticsearch, Fluentd, and Syslog.
2. **Docker log collection**: Once logs are generated by Docker containers, they must be collected and stored for analysis and troubleshooting. Docker provides a built-in log collection tool called "docker logs", which can be used to collect and view container logs. Additionally, third-party tools such as Fluentd and Logstash can be used to collect and store logs from multiple Docker hosts.
3. **Docker monitoring:** Monitoring Docker containers and applications is essential for maintaining their performance and availability. Docker provides a built-in monitoring tool called "docker stats", which can be used to monitor container resource usage, including CPU, memory, and network usage. Additionally, third-party tools such as Prometheus, Grafana, and Nagios can be used to monitor Docker containers and applications.
4. **Docker health checks:** Docker health checks allow users to define custom health checks for their containers and applications. These health checks can be used to monitor the health of containers and applications and to ensure that they are running correctly. Docker provides built-in health checks, as well as the ability to define custom health checks using scripts or commands.

By using Docker logging and monitoring tools, users can effectively manage and maintain Docker containers and applications. These tools provide insights into container and application performance, as well as the ability to troubleshoot issues and optimize resource usage.

Docker Logging and Monitoring with commands
Here are some commands that can be used to perform logging and monitoring in Docker:
Docker logging commands:
To view logs for a running container:
docker logs <container-name>
This command displays the logs for a running container in the terminal.
To follow logs in real-time for a running container:
docker logs -f <container-name>
This command follows the logs in real-time for a running container, displaying new logs as they are generated.
To view logs for a stopped container:
docker logs <container-name> --tail=100
This command displays the last 100 logs for a stopped container.
To set the logging driver for a container:
docker run --log-driver=<driver-name> <image-name>
This command sets the logging driver for a container to the specified driver name.
Docker monitoring commands:
To view resource usage for running containers:
docker stats
This command displays real-time statistics for running containers, including CPU usage, memory usage, and network I/O.
To view detailed information about a running container:
docker inspect <container-name>
This command displays detailed information about a running container, including its configuration, network settings, and resource usage.
To view a list of all running containers:
docker ps
This command displays a list of all running containers, including their names, image names, and resource usage.
To view a list of all stopped containers:
docker ps -a
This command displays a list of all stopped containers, including their names, image names, and exit status.

Python Dockerfile Example

```
# Use an official Python runtime as a parent image
FROM python:3.9-slim-buster

# Set the working directory to /app
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY . /app

# Install any needed packages specified in requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# Set environment variables
ENV PYTHONUNBUFFERED=1 \
    PYTHONDONTWRITEBYTECODE=1 \
    PYTHONPATH=/app:$PYTHONPATH

# Expose the port for the application
EXPOSE 8000

# Run the command to start the application
CMD ["gunicorn", "-w", "4", "-b", "0.0.0.0:8000", "app:app"]
```

Assuming you have a requirements.txt file in the same directory as the Dockerfile, you can build and run the container using these commands:

```
docker build -t my-python-app .
docker run -p 8000:8000 my-python-app
```

This example uses the official **python:3.9-slim-buster** base image and installs packages from **requirements.txt**. It also sets environment variables and exposes the application's port (8000 in this case). Finally, it uses the **CMD** instruction to run the **gunicorn** server with the appropriate settings to serve the application. Note that you will need to modify the **CMD** instruction to match the command needed to start your particular application.

@Sandip Das

GO Dockerfile Example

```
# Use an official Golang runtime as a parent image
FROM golang:1.17.2-alpine3.14 AS builder

# Set the working directory to /app
WORKDIR /app

# Copy the source code into the container
COPY . .

# Build the application
RUN go build -o myapp

# Use a smaller base image for the final image
FROM alpine:3.14

# Copy the application binary from the builder stage
COPY --from=builder /app/myapp /usr/local/bin/

# Set environment variables
ENV PORT=8080

# Expose the port for the application
EXPOSE 8080

# Run the command to start the application
CMD ["myapp"]
```

This example uses the official golang:1.17.2-alpine3.14 base image and builds the application inside the container. It then uses a smaller base image for the final image and copies the binary from the builder stage. It also sets environment variables and exposes the application's port (8080 in this case). Finally, it uses the CMD instruction to run the application.

@Sandip Das

Node.js Dockerfile Example

```
# Use an official Node.js runtime as a parent image
FROM node:16.13.0-alpine3.14 AS builder

# Set the working directory to /app
WORKDIR /app

# Copy the source code into the container
COPY . .

# Install dependencies and build the application
RUN npm ci --only=production && npm run build

# Use a smaller base image for the final image
FROM node:16.13.0-alpine3.14

# Copy the application files from the builder stage
COPY --from=builder /app/dist /app/node_modules
/app/package.json /app/package-lock.json /app/

# Set environment variables
ENV PORT=8080

# Expose the port for the application
EXPOSE 8080

# Run the command to start the application
CMD ["npm", "start"]
```

This example uses the official node:16.13.0-alpine3.14 base image and builds the application inside the container. It then uses a smaller base image for the final image and copies the necessary files from the builder stage. It also sets environment variables and exposes the application's port (8080 in this case). Finally, it uses the CMD instruction to run the application.

@Sandip Das

Java Dockerfile Example

```
# Use an official OpenJDK runtime as a parent image
FROM openjdk:17.0.1-jdk-slim-buster AS builder

# Set the working directory to /app
WORKDIR /app

# Copy the source code into the container
COPY . .

# Build the application
RUN ./mvnw package

# Use a smaller base image for the final image
FROM openjdk:17.0.1-jdk-slim-buster

# Copy the application binary from the builder stage
COPY --from=builder /app/target/myapp.jar /app/

# Set environment variables
ENV PORT=8080

# Expose the port for the application
EXPOSE 8080

# Run the command to start the application
CMD ["java", "-jar", "/app/myapp.jar"]
```

This example uses the official openjdk:17.0.1-jdk-slim-buster base image and builds the application inside the container using Maven. It then uses a smaller base image for the final image and copies the JAR file from the builder stage. It also sets environment variables and exposes the application

@Sandip Das

Rust Dockerfile Example

```
# Use the official Rust image as a builder
FROM rust:1.60 as builder

# Create a new empty shell project
RUN USER=root cargo new --bin myapp
WORKDIR /myapp

# Copy the Cargo.toml and Cargo.lock files and download
dependencies
COPY ./Cargo.toml ./Cargo.lock ./
RUN cargo build --release
RUN rm src/*.rs

# Copy the source code
COPY ./src ./src

# Build for release
RUN rm ./target/release/deps/myapp*
RUN cargo build --release

# Create the final image
FROM debian:buster-slim
COPY --from=builder /myapp/target/release/myapp
/usr/local/bin/myapp

# Set the startup command to run the binary
CMD ["myapp"]
```

- Uses the official Rust image to compile the Rust application.
- Creates a new Rust project.
- Copies the Cargo.toml and Cargo.lock files and builds the dependencies to cache them.
- Copies the actual source code of the project.
- Builds the Rust application for release.
- Creates a new stage based on debian:buster-slim for a smaller final image.
- Copies the compiled binary from the first stage.
- Sets the command to run the application.

@Sandip Das

Thank You

@LearnTechWithSandip

SUBSCRIBE

For reading till the end