Cloud native factor

1. Code base - git
2. Dependencies – pyproject.toml , package.json
3. Configuration – Database connection string, certs , keys in secret manager
4. Backing services - databases, message queues, caching systems, and email providers
5. Build Release Run - CICD
6. Process - state less service
7. Port binding – service should be hosted on a port
8. Concurrency – Service can easily scale out
9. Disposability – A service can be started and stopped quickly
10. Dev Prod parity – Same environment
11. Logs – Should be aggregate and processed in (ELK)
12. Admin process – migration scripts, data backup
13. API first - The application should be designed with an API-first approach, meaning that APIs are the primary method of communication between different services, whether internal or external.
14. Isolation - Cloud-native applications should be built with security and reliability in mind. Resources such as databases, networks, and environments should be isolated to ensure that each part of the application operates in a secure, controlled manner.

|  |  |  |
| --- | --- | --- |
| **Feature** | **Docker** | **Virtual Machines** |
| **Isolation** | OS-level isolation (sharing the kernel) | Hardware-level isolation (separate OS) |
| **Startup Time** | Fast (seconds) | Slower (minutes) |
| **Resource Efficiency** | Lightweight, minimal overhead | Heavyweight, higher resource usage |
| **Performance** | Near-native, faster execution | Slower, due to overhead of full OS |
| **Portability** | Highly portable across environments | Less portable, more tied to the hypervisor |
| **Security** | Less isolated, relies on the kernel security | Stronger isolation between environments |
| **Management** | Easier to manage with Docker Compose/K8s | Managed by hypervisor tools (e.g., VMware) |
| **Use Cases** | Microservices, CI/CD, DevOps, lightweight apps | Legacy systems, OS-specific workloads |

|  |  |  |
| --- | --- | --- |
| **Feature** | **docker run** | **docker create** |
| **Creates Container** | Yes, and starts it. | Yes, but does not start it. |
| **Starts Container** | Yes, automatically starts the container. | No, you need to use docker start to start it later. |
| **Use Case** | Ideal for when you want to immediately start the container. | Ideal for when you want to create the container first, then decide when to start it. |
| **Typical Workflow** | Most commonly used in everyday Docker workflows. | Used in specific cases where you need to create containers but not start them right away. |

**How Docker Networking Works**

1. **IP Addressing**: Each container on a network is assigned an IP address, but how this IP is allocated depends on the network driver. For example:
   * In the bridge network, containers are assigned IPs from the bridge's internal subnet.
   * In overlay networks, containers may be assigned IPs from a larger subnet managed by Docker Swarm or other orchestration tools.
2. **DNS Resolution**: Docker provides an internal DNS service. Containers on the same network can resolve each other by container name (not just IP), which simplifies communication between containers.
   * For example, if you have a container named db and another named web, the web container can access the dbcontainer by simply using db as the hostname.
3. **Network Isolation**: Docker containers on the same network can communicate with each other, but containers on different networks are isolated by default. For example, a container on the bridge network cannot communicate with a container on the host network unless network rules are explicitly configured to allow such communication.
4. **Port Mapping (for Bridge and Host Networks)**: For containers on bridge and host networks, you can map container ports to host ports using the -p or --publish flag. This allows services running inside containers to be accessed from outside the container (i.e., from the host or external networks).
   * Example: docker run -p 8080:80 my\_image exposes port 80 of the container to port 8080 on the host.
5. **Communication Between Containers**:
   * **Same Network**: Containers on the same network can communicate with each other directly.
   * **Different Networks**: Containers on different networks need special configurations (such as using an external network, connecting networks together, or using a multi-host network solution like Docker Swarm or Kubernetes).
6. **Routing Traffic**: Docker manages routing between containers and the host using internal routing tables and network drivers. For example, when a container on the bridge network sends traffic to the host, the traffic is routed through the Docker bridge network.

|  |  |  |
| --- | --- | --- |
| **Network Driver** | **Description** | **Use Case** |
| **Bridge** | Default network; containers are isolated but can communicate with each other. | Single host, isolated container communication. |
| **Host** | Removes network isolation; containers share the host’s network interface. | Performance-sensitive applications requiring direct access to host network. |
| **Overlay** | Containers on multiple hosts can communicate as if they’re on the same network. | Multi-host communication (Docker Swarm, Kubernetes). |
| **Macvlan** | Containers get their own MAC addresses, appearing as physical devices on the network. | Applications requiring containers to appear as separate network devices. |
| **None** | No network; container is isolated from all networks. | Specialized use cases where networking is not needed. |

 **What is the purpose of the FROM instruction in a Dockerfile?**

* The FROM instruction specifies the base image from which you want to build your image. It is the first instruction in a Dockerfile.

 **What is the difference between ADD and COPY commands in a Dockerfile?**

* ADD can copy files and also supports extracting tar files and downloading files from URLs. COPY is simpler, used purely to copy files from the build context into the container.

 **How do you create a Docker image from an existing container?**

* You can create an image from a container using the docker commit command.

 **How can you optimize a Docker image to reduce its size?**

* By using multi-stage builds, minimizing the number of layers, avoiding unnecessary dependencies, using smaller base images, and removing temporary files.

 **Explain the purpose of the CMD and ENTRYPOINT instructions in Dockerfiles.**

* CMD provides default commands to run when the container starts, but it can be overridden. ENTRYPOINT defines the main command to run in the container, and it cannot be overridden by default arguments.

 **What is the significance of the --no-cache flag when building a Docker image?**

* The --no-cache flag forces Docker to ignore the build cache and rebuild all layers from scratch, ensuring that no old layers are reused.

 **What is the difference between docker pull and docker build?**

* docker pull fetches an image from a remote repository, whereas docker build creates a new image from a Dockerfile.

 **What are the potential issues with using the latest tag for Docker images?**

* Using latest can introduce unpredictability because it points to the most recent image version, which may lead to inconsistencies in environments.

 **How do you update a Docker image with a new version of an application?**

* You can update a Docker image by modifying the Dockerfile or the underlying application and rebuilding the image.

 **How can you check the history of a Docker image?**

* You can use the docker history command to inspect the layers and commands that were used to build the image.

A **Docker volume** is a persistent storage mechanism in Docker, designed to store data that is independent of the lifecycle of containers. Volumes are stored outside the container's filesystem, meaning data is not lost when a container is deleted or recreated. Volumes are useful for databases, logs, and other data that needs to persist even if containers are stopped or removed.

**Key Benefits of Docker Volumes:**

* **Persistence**: Data survives container restarts and removals.
* **Sharing**: Volumes can be shared between multiple containers.
* **Backup and Restore**: Volumes allow for easy backup and restore of data.
* **Decoupling**: Volumes decouple the data from container file systems, making it easier to manage.

**Example: Using Docker Volumes**

**1. Create a Volume**

Create a Docker volume to store persistent data:

docker volume create my-volume

This creates a named volume called my-volume.

**2. Using the Volume with a Container**

Next, let's run a container and mount the volume inside it. For example, use a mysql container and mount the volume to persist the MySQL database data.

docker run -d \

--name mysql-container \

-e MYSQL\_ROOT\_PASSWORD=my-secret-pw \

-v my-volume:/var/lib/mysql \

mysql:latest

* -v my-volume:/var/lib/mysql: This mounts the my-volume Docker volume to the /var/lib/mysql directory inside the container. Any data written to this directory (such as the MySQL database files) will be stored in the volume, and it will persist even if the container is stopped or removed.

**3. Verify the Volume Is Mounted**

You can verify that the volume is correctly mounted by inspecting the container:

docker inspect mysql-container

Look for the "Mounts" section to see where the volume is mounted in the container.

**4. Accessing Volume Data**

If you want to inspect the data stored in the volume, you can start another container and mount the same volume:

docker run -it --rm -v my-volume:/data alpine sh

This starts a container with an Alpine Linux image, mounts the volume my-volume to /data, and gives you an interactive shell to inspect the contents.

**5. Stop and Remove the Container, but Keep the Volume**

Now, stop and remove the mysql-container, but the data in my-volume will remain intact:

docker stop mysql-container

docker rm mysql-container

Even though the container is removed, the volume (my-volume) still exists. You can create a new container and mount the volume again.

**6. Using the Volume in a New Container**

You can use the same volume with a new container:

docker run -d \

--name new-mysql-container \

-e MYSQL\_ROOT\_PASSWORD=my-new-pw \

-v my-volume:/var/lib/mysql \

mysql:latest

Since the volume already contains data (e.g., a MySQL database), the new container will use the existing data when it starts.

**Example of a Volume for Log Persistence**

Suppose you have a web server (e.g., nginx), and you want to persist its logs. You can mount a volume to store logs:

docker run -d \

--name nginx-container \

-v nginx-logs:/var/log/nginx \

nginx

This command will mount the nginx-logs volume to the Nginx container's /var/log/nginx directory, ensuring that log files persist even after the container is stopped or deleted.

**Removing a Volume**

If you're done with a volume and want to clean it up, you can remove it:

bash

Copy code

docker volume rm my-volume

If the volume is not in use by any containers, it will be removed. Otherwise, you'll need to stop and remove the container using the volume before removing it.

**Benefits of Multi-Stage Builds in this Example:**

* **Smaller Image**: The final image only contains the Go binary and the runtime dependencies (from the alpineimage), without any of the development tools (like go or gcc).
* **Cleaner Dockerfile**: The Dockerfile is organized in stages, making it easier to maintain and understand.
* **Faster Builds**: Layers that don't change (e.g., the Go dependencies) are cached, improving build speed on subsequent builds.

# Stage 1: Build the Go binary

FROM golang:1.19 AS builder

# Set the Current Working Directory inside the container

WORKDIR /app

# Copy the Go module files

COPY go.mod go.sum ./

# Download all dependencies. Dependencies will be cached if the go.mod and go.sum files are not changed

RUN go mod tidy

# Copy the source code into the container

COPY . .

# Build the Go app

RUN go build -o myapp .

# Stage 2: Create the final, minimal image

FROM alpine:latest

# Set the Current Working Directory inside the container

WORKDIR /root/

# Copy the compiled Go binary from the builder stage

COPY --from=builder /app/myapp .

# Expose port 8080

EXPOSE 8080

# Command to run the executable

CMD ["./myapp"]

**What is the docker-compose.override.yml file?**

* The docker-compose.override.yml file is used to override or extend the configuration in the main docker-compose.yml file. Docker Compose automatically looks for this file and applies its settings when running docker-compose up.
* It's useful for setting different configurations for development, testing, and production environments.
* Example:

yaml

Copy code

# docker-compose.override.yml

version: '3'

services:

web:

environment:

- NODE\_ENV=development

ports:

- "3000:3000"