Docker Introduction

By Eng: Abdelrahman Ayman Lotfy

DevOps track

Table of contents

why Docker?

- **Solve the "it works on my machine" problem**: Docker addressed inconsistencies in application behavior across different environments due to varying dependencies, libraries, and configurations.
- **Simplify containerization**: Built on Linux container technologies like LXC, Docker provided a user-friendly API and ecosystem for building, shipping, and managing containers.
- **Enable portability**: Containers package applications with all required components, ensuring consistent execution across any Docker-supported system.
- **Improve efficiency**: Offers lightweight, isolated environments using OS-level virtualization, reducing the overhead of full virtual machines.
- **Automate deployment**: Streamlines the process of deploying applications with reproducible environments.

History of Docker

- **2008/2010**: DotCloud, a PaaS company, founded by Solomon Hykes, Kamel Founadi, and Sebastien Pahl in Paris, part of Y Combinator's Summer 2010 incubator.
- **2011**: DotCloud launched its PaaS platform; Docker began as an internal project to leverage container technology for application isolation.
- **March 2013**: Docker debuted at PyCon in Santa Clara, CA, and was released as open-source software, initially using LXC as its execution environment.
- **2013**: DotCloud rebranded to Docker Inc., focusing on containerization.

- **2014**:
  - ❖ Docker version 0.9 replaced LXC with libcontainer (written in Go) for better control and portability.
  - ❖ Partnerships formed with Red Hat (Fedora, OpenShift integration), Microsoft (Windows Server support), Amazon EC2, and IBM.
  - ❖ Docker Compose beta released in December 2013, with version 1.0 in October 2014.
- **2015**: New open standard for containers announced with multiple companies.
- **2016**: Swarm mode integrated into Docker Engine (version 1.12).
- **2017**: Moby project launched for open R&D on container systems; Docker usage grew 160% on LinkedIn by January.
- **2019**: Enhanced Windows support via WSL 2 announced in May.
- **2021**: Docker introduced a Personal Plan and ended free Docker Desktop for large businesses in August.
- **2023**: Docker acquired AtomicJar in December to enhance testing features.
- **2024**: Docker celebrated its 11th anniversary, solidifying its role in the container revolution.
- **Present**: Docker Engine remains open-source, widely used in cloud-native ecosystems, with ongoing development by Docker Inc.

Docker Build Process

- **Parse Dockerfile**: Docker reads the Dockerfile in the current directory (or specified path) to understand the instructions for building the image.
- **Create a build context**: Collects all files in the current directory (or specified context) to be used during the build, excluding those listed in .dockerignore.
- **Set up temporary container**: Docker creates an intermediate container for each instruction in the Dockerfile to execute commands in isolation.
- **Execute Dockerfile instructions**:
    - ❖ Processes commands like FROM (sets base image), COPY/ADD (adds files to image), RUN (executes commands), ENV (sets environment variables), and others.
    - ❖ Each instruction creates a new layer in the image, cached for reuse in future builds unless changes occur.
- **Generate image layers**: Each instruction's result is stored as a read-only layer, forming the final image's filesystem.
- **Tag the image**: Assigns a tag (e.g., myapp:latest) to the built image, as specified by the -t flag or default naming.
- **Save the image**: Stores the final image in the local Docker image registry, ready for use or pushing to a remote registry (e.g., Docker Hub).
- **Clean up**: Removes intermediate containers used during the build to save space, unless --no-cache is used.

Internal process:

- ❖ **Client-Server communication**: The Docker CLI sends the build request to the Docker daemon (via REST API over a Unix socket or TCP).

- **BuildKit integration**: If enabled, BuildKit (Docker's build backend) optimizes the build with parallel processing and efficient caching.
- **Storage driver**: The daemon uses a storage driver (e.g., overlay2, aufs) to manage image layers and copy-on-write filesystem operations.
- **Namespace isolation**: Leverages Linux namespaces (e.g., PID, mount, network) for container isolation during intermediate container execution.
- **Cgroups**: Applies control groups to limit resources (CPU, memory) for build processes.
- **Image manifest**: Creates a JSON manifest detailing the image's layers, configuration, and metadata, stored in the local registry.

Docker Run Process

- **Locate the image**: Docker checks the local registry for the specified image; if not found, it pulls it from a remote registry (e.g., Docker Hub) unless --pull=never is set.
- **Create a container**: Allocates a new container from the specified image, setting up an isolated environment with its own filesystem, network, and process space.
- **Configure container settings**:
  - Applies options from the docker run command, such as port mappings (-p), environment variables (-e), volumes (-v), or detach mode (-d).
  - Sets the container's entrypoint and command as defined in the Dockerfile (ENTRYPOINT and CMD) or overridden via command-line arguments.
- **Start the container**: Initializes the container's runtime, executing the specified command or entrypoint in the isolated environment.

- **Allocate resources**: Assigns CPU, memory, and other resources based on defaults or user-specified limits (e.g., --memory, --cpu-shares).
- **Set up networking**: Configures the container's network, connecting it to the specified network (e.g., bridge, host) and mapping ports if requested.
- **Run the application**: Executes the container's main process, outputting logs to the terminal (unless detached with -d) and keeping the container running until the process exits.
- **Handle container lifecycle**: If the process exits, the container stops unless configured to restart (e.g., --restart=always); stopped containers remain for inspection unless removed with --rm.

Internal process:

- **Client-Server interaction**: The Docker CLI sends the run request to the Docker daemon via the API.
- **Container runtime**: Uses a runtime (e.g., runc, containerd) to create and manage the container, interfacing with the Linux kernel.
- **Namespaces**: Sets up Linux namespaces (PID, mount, network, user, UTS, IPC) to isolate the container's processes, filesystem, hostname, and network stack.
- **Cgroups**: Configures control groups to enforce resource limits and track resource usage.
- **Storage driver**: Mounts the image's read-only layers with a writable layer (using storage drivers like overlay2) for the container's filesystem.

- **Networking setup**: Configures network namespaces, virtual ethernet pairs (veth), and bridges (e.g., docker0) for connectivity, with iptables rules for port mapping.
- **Seccomp and AppArmor**: Applies security profiles (seccomp filters, AppArmor policies) to restrict system calls and enhance container security.
- **Log management**: Streams container logs to the CLI or a logging driver (e.g., json-file, syslog) based on configuration.

Docker Instructions

All the following inside a dockerfile:

1. FROM

Defines the **base image**.

```
FROM ubuntu:20.04
```

---

2. RUN

Runs a command during the image build (creates a new layer).

```
RUN apt-get update && apt-get install -y python3
```

---

3. CMD

Defines the **default command** to run when a container starts.
(Only one CMD, last one overrides).

```
CMD ["python3", "app.py"]
```

4. ENTRYPOINT

Similar to CMD, but **cannot be overridden** easily. Used for fixed executables.

```
ENTRYPOINT ["python3", "app.py"]
```

---

5. COPY

Copies files from the **host machine → image**.

```
COPY . /app
```

---

6. ADD

Like COPY, but supports remote URLs and tar extraction.

```
ADD https://example.com/file.tar.gz /app/
```

---

7. WORKDIR

Sets the **working directory** inside the container.

```
WORKDIR /app
```

---

8. ENV

Set environment variables.

```
ENV PORT=5000
```

---

9. EXPOSE

Documents the port your app will use (doesn't publish it automatically).

```
EXPOSE 5000
```

10. VOLUME

Creates a mount point for external volumes.

VOLUME ["/data"]

---

11. ARG

Defines build-time variables (used only during docker build).

ARG APP_VERSION=1.0

---

12. LABEL

Add metadata to the image.

LABEL maintainer="your email"

LABEL version="1.0"

---

13. USER

Set the user that the container will run as.

USER appuser

---

14. SHELL

Change the default shell used by RUN commands.

SHELL ["/bin/bash", "-c"]

Docker Commands

Image Management

```
docker build -t name:tag .      # Build an image from Dockerfile
docker images                   # List all images
docker rmi image_id             # Remove an image
```

Container Lifecycle

```
docker run image                # Create & start a container
docker run -it ubuntu bash      # Run container with interactive terminal
docker ps                       # List running containers
docker ps -a                    # List all containers
docker stop container_id        # Stop a container
docker start container_id       # Start a container
docker rm container_id          # Remove a container
```

Debugging & Logs

```
docker logs container_id            # Show container logs
docker exec -it container_id bash   # Access container shell
docker inspect container_id         # Detailed info about container
```

Networking

```
docker port container_id            # Show port mappings
docker network ls                   # List networks
docker network create mynet         # Create a custom network
```

Volumes (Persistent Data)

```
docker volume ls                    # List volumes
docker volume create myvol          # Create a volume
docker run -v myvol:/data image     # Mount volume inside container
```

Docker Hub (Registry)

```
docker login          # Login to Docker Hub
docker pull image     # Download image
docker push image     # Upload image
```

Conclusion

- Docker solves real-world problems like environment inconsistencies, dependency conflicts, and heavy virtual machines.
- It became popular because it is lightweight, portable, and fast.
- Images are the blueprints, and containers are the running instances of those images.
- docker build creates an image (read-only layers), while docker run creates a container (with a writable layer).
- Dockerfile instructions (FROM, RUN, COPY, CMD, ENTRYPOINT, etc.) define how to build images.
- Docker has become a standard in DevOps and cloud computing, and is often combined with Kubernetes for orchestration.

Docker makes building, shipping, and running applications easier, faster, and more reliable everywhere.