

Containers

Tópicos de Informática para Automação

Mário Antunes

October 13, 2025

Universidade de Aveiro

Introduction to Containers

A Modern Way to Package and Run Applications

Terminology

Before we dive in, let's define some key terms.

- **Image:** A read-only, inert template containing an application and its dependencies. Think of it as a **blueprint** or a class in object-oriented programming.
- **Container / Instance:** A runnable **instance** of an image. This is the actual, living application (like an object created from a class). The terms are often used interchangeably.
- **Registry:** A storage system for container images. **Docker Hub** is a popular public registry.
- **Docker Engine:** The underlying client-server application that builds and runs containers.
- **Volume:** A mechanism for persisting data outside a container's ephemeral filesystem.

The Problem: “It Works on My Machine!” 🤔

Every developer has faced this classic problem:

- Your application works perfectly on your laptop (which has Python 3.9, a specific library version, and runs Debian).
- When you give it to a colleague (who has Python 3.8 and runs macOS) or deploy it to a server (running an older OS), it fails.

These differences in environments create a massive challenge for software portability.

The Solution: Containers

A **container** is a standard, executable unit of software that packages up an application's code along with all its runtime dependencies. This package is **isolated**, ensuring the application runs uniformly and consistently everywhere.

Analogy: A container is like a standardized shipping container. It doesn't matter what's inside; it can be handled by any compatible ship (host machine).

How Isolation is Achieved: Namespaces

Containers run at **full hardware** speed because they are just isolated processes on the host's kernel. The isolation is provided by **Linux Namespaces**.

Namespaces virtualize system resources for a process, making it seem like it has its own private copy. Key namespaces include:

- **PID:** Isolates process IDs. Inside the container, your app is PID 1.
- **NET:** Provides an isolated network stack (IP addresses, routing tables).
- **MNT:** Isolates filesystem mount points.

Analogy: Namespaces are like the walls, private mailboxes, and unique door keys for each apartment in a building.

How Resources are Managed: Cgroups

To prevent one container from consuming all system resources, the Linux kernel uses **Control Groups (cgroups)**.

Cgroups allow the host to limit and monitor the resources a container can use, such as:



- CPU usage (e.g., limit to 1 CPU core).
- Memory (e.g., limit to 512 MB of RAM).
- Disk I/O bandwidth.

Analogy: Cgroups are like the utility meters and circuit breakers for each apartment, ensuring no single tenant can use all the building's water or electricity.

VMs vs. Containers #1

- **Virtual Machines (VMs)** virtualize the **hardware**. Each VM includes a full copy of a guest OS and kernel. They are heavyweight and take minutes to boot.
- **Containers** virtualize the **operating system**. They share the host system's kernel and are lightweight, booting in seconds.

VMs vs. Containers #2

Feature	Virtual Machines (VMs)	Containers
Analogy	 Houses: Fully self-contained.	 Apartments: Share building infrastructure.
Level of Abstraction	Hardware Virtualization	OS Virtualization
Size	Gigabytes (GB)	Megabytes (MB)
Startup Time	Minutes	Seconds or less
Performance Overhead	Low to Medium	Very Low (Near-native)
Resource Usage	Higher (Full OS per VM)	Lower (Shared OS Kernel)
Isolation	Strong (Hardware level)	Good (Process level)
Portability	Portable (but large)	Extremely Portable

The Container Image & Its Layers

An **image** is a read-only template built from a series of stacked **layers**. Each instruction in a `Dockerfile` creates a new layer.

This makes builds fast and disk usage efficient, as multiple images can share common base layers.

Persistent Data: Volumes

By default, a container's filesystem is **ephemeral** (deleted when the container stops).

To save data permanently, you use **volumes**, which map a *directory* inside the container to a *directory* on the host machine.

Container Networking & DNS

The container engine creates a **virtual bridge network**. Containers on the same network get a private IP and can communicate.

- **Port Mapping:** To expose a container's service to the outside world, you map a host port to a container port (e.g., `_p 8080:80`).
- **Internal DNS:** When using Docker Compose, each service can reach another using its service name as a hostname. Your webapp code can simply connect to `http://database` to reach the database container.

Introducing Docker

Docker is the platform that popularized containers. It provides a simple set of tools to build, ship, and run any application, anywhere.

- **Docker Engine:** The background service (daemon) that manages containers.
- **Docker CLI:** The command-line tool you use to interact with the Docker Engine.
- **Docker Hub:** A public registry of pre-built container images.

Common Docker Commands

Command	Description
<code>docker run [image]</code>	Creates and starts a new container from an image.
<code>docker ps</code>	Lists all running containers. <code>ps -a</code> lists all containers (running or stopped).
<code>docker stop [id/name]</code>	Stops a running container gracefully.
<code>docker rm [id/name]</code>	Removes a stopped container.
<code>docker logs [id/name]</code>	Fetches the logs (standard output) from a container.
<code>docker pull [image]</code>	Downloads an image from a registry (like Docker Hub).
<code>docker images</code>	Lists all images stored locally.
<code>docker build -t [name] .</code>	Builds a new image from a Dockerfile in the current directory.

The Dockerfile: A Deeper Look

A `Dockerfile` is a recipe for building a container image. Here are the most common instructions:

- `FROM`: Specifies the base image to build upon (e.g., `ubuntu:22.04`).
- `WORKDIR`: Sets the working directory for subsequent commands.
- `COPY`: Copies files or directories from the host into the image.

- **RUN:** Executes a command during the image build process (e.g., `RUN apt-get install -y nginx`).
- **CMD:** Provides the default command to run when a container is started from the image.
- **ENTRYPOINT:** Configures the container to run as an executable.
- **EXPOSE:** Informs Docker that the container listens on the specified network ports at runtime.
- **ENV:** Sets persistent environment variables.

Dockerfile Example: A Logging Service

This simple Dockerfile creates a service whose only job is to print a timestamp every 5 seconds. This is perfect for testing the `docker logs` command.

```
# Use a minimal base image
FROM alpine:latest

# The command to execute when the container starts.
# It's an infinite loop that prints the current
# date and sleeps for 5 seconds.
CMD ["sh", "-c", "while true; do echo\n[LOG] Server is running at $(date)\n"; sleep 5; done"]
```

To build and run it:

```
$ docker build -t logging-service .  
$ docker run -d --name logger logging-service  
$ docker logs -f logger
```

Docker Compose: A Deeper Look

A `compose.yml` file defines a multi-service application. Here are the most common keys:

- `services`: The root key where all your application services are defined.
- `image`: Specifies a pre-built image from a registry (like Docker Hub).
- `build`: Specifies the path to a `Dockerfile` to build the service's image.

- `ports`: Maps ports from the host to the container (e.g., `"8080:80"`).
- `volumes`: Mounts host paths or named volumes into the container.
- `environment`: Sets environment variables for the service.
- `depends_on`: Defines dependencies between services, controlling startup order.

Compose Example 1: Building a Custom NGINX Image

This example shows how to package your website's files directly into a custom image.

Required File Structure:

```
.
├── docker-compose.yml
├── Dockerfile
└── my-website/
    └── index.html
```

Dockerfile

```
# Use the official NGINX image as a base
FROM nginx:alpine

# Copy our custom webpage into the image's web root directory
COPY ./my-website /usr/share/nginx/html
```

docker-compose.yml

```
services:
  webserver:
    build: .
    ports:
      - "8080:80"
```

Example 1: Explanation

In this method, we create a **self-contained, portable image** that includes our application code.

1. When you run `docker-compose up`, the `build: .` directive tells Compose to look for a `Dockerfile` in the current directory.
2. The `Dockerfile` starts from a standard `nginx` base image.
3. The `COPY` instruction takes your local `./my-website` folder and copies its contents directly into the image's filesystem at `/usr/share/nginx/html`.
4. A new, custom image is created containing both `NGINX` and your webpage.
5. A container is started from this new image.

Key Concept: The application and its code are bundled together. This is ideal for **production deployments**, as the resulting image is a consistent, immutable artifact that can be run anywhere.

Compose Example 2: Using a Volume to Serve Content

This example uses a standard NGINX image and injects the website content using a volume.

Required File Structure:

```
.
├── docker-compose.yml
└── my-website/
    └── index.html
```

docker-compose.yml

```
services:
  webserver:
    image: nginx:alpine
    ports:
      - "8080:80"
    volumes:
      - ./my-website:/usr/share/nginx/html
```

(No Dockerfile is needed for this method)

Example 2: Explanation

This method keeps your code on the host machine and dynamically links it into the container.

1. When you run `docker-compose up`, the `image: nginx:alpine` directive tells Compose to pull the standard NGINX image from Docker Hub. No custom image is built.
2. A container is started from this standard image.
3. The `volumes` directive creates a live link between the `./my-website` folder on your host and the `/usr/share/nginx/html` folder inside the container.
4. When NGINX inside the container looks for files to serve, it is actually reading them directly from your host machine's disk.

Key Concept: The container is stateless, and the code lives on the host. If you change your `index.html` file on the host, the change is reflected **instantly** without rebuilding or restarting the container. This is ideal for **local development**.

Compose Example 3: NGINX with a Varnish Cache

This advanced example orchestrates two services: an NGINX web server and a Varnish cache that sits in front of it to speed up content delivery.

Required File Structure:

```
.
├── docker-compose.yml
└── varnish/
    └── default.vcl
```

varnish/default.vcl (Varnish Configuration)

```
vcl 4.1;

// Define the backend server Varnish will get content from.
// 'nginx' is the name of our other service in docker-compose.yml.
backend default {
    .host = "nginx";
    .port = "80";
}
```

docker-compose.yml

```
services:
  # The Varnish cache that is exposed to the outside world
  cache:
    image: varnish:stable
    volumes:
      # Mount our custom Varnish configuration
      - ./varnish:/etc/varnish
    ports:
      # Map host port 8080 to the cache's port 80
      - "8080:80"
    depends_on:
      - nginx

  # The NGINX web server, which is NOT exposed to the host
  nginx:
    image: nginx:alpine
    # No ports section means it's only accessible from within the Docker network
```

Example 3: Explanation

This setup demonstrates a realistic, high-performance, multi-tier architecture where services communicate internally.

1. The `docker-compose.yml` defines two services: `cache` (Varnish) and `nginx`.
2. Only the `cache` service exposes a port (8080) to the host machine. The `nginx` service is completely isolated from the outside world.
3. The custom Varnish configuration file (`default.vcl`) is mounted into the `cache` container. This file tells Varnish that its “backend” (the actual web server) is located at the hostname `nginx`.
4. Thanks to Docker’s **internal DNS**, the service name `nginx` automatically resolves to the private IP address of the `nginx` container, allowing Varnish to connect to it.

The Request Flow: User's Browser -> Host Machine (Port 8080) -> Varnish Container (Cache) -> NGINX Container (Origin Server)

The Caching Magic: On the first request for a webpage, Varnish fetches it from the nginx container and stores a copy in its memory. For all subsequent requests for that same page, Varnish serves the copy directly from its cache, which is incredibly fast and prevents the NGINX server from having to do any work.

Key Concept: This demonstrates powerful **service discovery** and the creation of a **reverse proxy**, a fundamental pattern in web architecture.

The Origin: Linux Containers (LXC)

Before Docker, there was **LXC**.

- LXC is a user-space interface for the Linux kernel's containment features (namespaces and cgroups).
- It provides a lower-level set of tools for creating and managing containers.
- LXC containers are often described as being more like very lightweight, fast-booting virtual machines than application containers. They typically run a full `init` system and are used for isolating entire operating systems.

The Standard: Docker

Docker took the underlying technology of LXC and built a high-level, user-friendly ecosystem around it.

- It introduced the concept of portable images via the `Dockerfile`.
- It created a centralized registry (Docker Hub) for sharing images.
- Its focus is on **application-centric** containers, packaging a single application or process per container. This philosophy is a cornerstone of the microservices architecture.

The Modern Alternative: Podman

Podman is a popular, modern alternative to Docker, developed by Red Hat.

- **Daemonless:** Unlike Docker, Podman does not require a central, always-running daemon, which is often cited as a security benefit.
- **Rootless:** Podman is designed to run containers as a regular user, without needing root privileges.
- **CLI Compatible:** Podman's command-line interface is intentionally identical to Docker's. In many systems, you can simply run `alias docker=podman` and use the same commands you already know.

Conclusion & Key Takeaways

- Containers solve the “it works on my machine” problem by packaging an application with its dependencies into a **portable** unit.
- They achieve isolation and resource management via Linux kernel features like **namespaces** and **cgroups**.
- **Dockerfile** provides a recipe to build images, and **Docker Compose** helps manage multi-service applications.
- Containers have revolutionized software development, forming the foundation of modern **DevOps** and **cloud-native** practices.

To continue your learning journey with containers, here are some excellent resources:

- **Docker Official Cheat Sheet:** A concise, official reference for the most common commands.
 - https://docs.docker.com/get-started/docker_cheatsheet.pdf
- **Docker Ultimate Cheat Sheet (Collabnix):** A more comprehensive cheat sheet with detailed examples and explanations.
 - <https://dockerlabs.collabnix.com/docker/cheatsheet/>

- **How to Optimize Docker Images (GeeksforGeeks):**
Learn techniques like multi-stage builds to make your images smaller, faster, and more secure.
 - <https://www.geeksforgeeks.org/devops/how-to-optimize-docker-image/>
- **Optimizing Dockerfiles for Fast Builds (WarpBuild):**
Understand how to structure your Dockerfile to take full advantage of layer caching and significantly speed up your build process.
 - <https://www.warpbuild.com/blog/optimizing-docker-builds>
- **LinuxServer.io:** A community project that provides and maintains high-quality, easy-to-use container images for many popular self-hosted applications (like media servers, download clients, and more).
 - <https://www.linuxserver.io/>