Docker Volumes

Why containers are ephemeral, named volumes, mounting a volume with docker run -v, where PostgreSQL stores its data, named volumes vs. bind mounts, and verifying persistence with a deliberate test.

Intermediate 30 min

Last updated: August 31, 2026
Read online: https://www.learnforgex.com/en/topics/docker-volumes

---

"Docker Networking" ran this project's own PostgreSQL and application containers side by side, connected by name. There's an uncomfortable fact that setup glossed over: `docker rm learning-platform-db` — a completely ordinary command, already covered back in "Cleaning Up: `docker rm`" — would have deleted every row that database ever stored, permanently, with no warning. This lesson is about why that happens, and the mechanism (a **volume**) that prevents it.

Why Containers Are Ephemeral

Everything a running container writes to its own filesystem lives in that container's own **writable layer**, sitting on top of the read-only image it was started from. That layer is part of the container itself — not the image, not anything separate — so `docker rm` (already covered in "Cleaning Up: `docker rm`") deletes it along with everything else about that specific container instance. A PostgreSQL container with no volume configured stores its actual database files inside that same writable layer, which means the data's entire lifetime is tied to that one container's lifetime — stop and restart the *same* container and the data is still there (`docker stop` / `docker start` don't touch the writable layer), but remove it, even to replace it with an identical new one, and the data is gone permanently. This is what "ephemeral" means here: not that a container's data disappears quickly, but that it disappears completely the moment the container itself does.

Named Volumes

A **named volume** is storage Docker manages itself, entirely outside of any single container's writable layer — creating one is a single command, independent of any container that will eventually use it.

```
docker volume create learning-platform-db-data
```

```
learning-platform-db-data
```

Because a volume exists independently, it survives exactly the event that destroys a container's own writable layer: `docker rm`. A new container — even a completely different one, started from a different image — can be pointed at the same existing volume and pick up right where the previous one's data left off.

Mounting a Volume: `-v`

A volume does nothing on its own until it's **mounted** into a container at a specific path, using `-v <volume-name>:<path-inside-container>` on `docker run`:

```
docker run --name learning-platform-db \
  -e POSTGRES_PASSWORD=secret \
  -v learning-platform-db-data:/var/lib/postgresql/data \
  -d postgres:16
```

From that container's point of view, `/var/lib/postgresql/data` looks like an ordinary directory — anything written there is, in reality, being written into the volume Docker manages, not into the container's own disposable writable layer.

Where PostgreSQL Stores Its Data

`/var/lib/postgresql/data` isn't an arbitrary path chosen for this lesson — it's exactly where the official `postgres` image (already used throughout "Docker CLI Fundamentals" and "Docker Networking") stores its actual database files by default, documented as such by the image itself. Mounting a volume at *that specific path* is what separates "a PostgreSQL container whose data disappears the moment it's removed" from "a PostgreSQL container whose data outlives it" — nothing about how PostgreSQL itself runs changes; only where its files physically end up does.

Verifying Persistence

The only real test of a volume actually working is destructive on purpose: write data, remove the container entirely, start a new one against the same volume, and confirm the data is still there.

```
docker exec -it learning-platform-db psql -U postgres -c "INSERT INTO proof (note) VALUES
('still here');"

docker stop learning-platform-db
docker rm learning-platform-db

docker run --name learning-platform-db -v learning-platform-db-data:/var/lib/postgresql
/data -d postgres:16

docker exec -it learning-platform-db psql -U postgres -c "SELECT * FROM proof;"
```

If the row from before the removal is still there, the volume — not the container — was where the data actually lived the whole time. "Putting It All Together," later in this lesson, walks through this exact sequence in full.

Named Volumes vs. Bind Mounts

A **bind mount** is the other way to attach host storage to a container — instead of a Docker-managed volume, it maps a specific, literal path on the host machine directly into the container (`-v /home/dev/my-project:/app` instead of `-v <volume-name>:/app`). Both use the same `-v` flag, but they solve different problems: a bind mount is for when a specific host path matters — mounting a project's own source code into a container during local development, for instance — while a named volume is for data a container manages and Docker should own the lifecycle of, exactly PostgreSQL's own data files. This course sticks to named volumes for exactly that reason; a database's data directory isn't something meant to be edited directly from the host machine's own filesystem.

Managing Volumes: `docker volume ls` / `inspect` / `rm`

```
docker volume ls
```

```
DRIVER    VOLUME NAME
local     learning-platform-db-data
```

`docker volume inspect learning-platform-db-data` shows exactly where Docker actually stores that volume's data on the host machine's own disk — worth knowing exists, rarely worth touching directly. `docker volume rm learning-platform-db-data` deletes a volume permanently, and only succeeds if no container currently has it mounted.

> `docker volume rm` is the one command in this lesson that actually does what `docker rm` looked like it might do — permanently deletes the data. It's not run by accident nearly as often as `docker rm` is, precisely because volumes exist to be the thing that survives ordinary container cleanup — but it's worth knowing this is the one Docker command that genuinely, deliberately destroys persisted data.

Putting It All Together

The complete sequence: create a volume, run PostgreSQL against it, write real data, remove the container entirely, and confirm a brand-new container still sees that data:

```
#!/bin/sh
# Proving that data written to a named volume survives even after the
# container that wrote it is completely removed -- not just stopped.

docker volume create learning-platform-db-data
# Output:
# learning-platform-db-data

docker run --name learning-platform-db \
  -e POSTGRES_PASSWORD=secret \
  -v learning-platform-db-data:/var/lib/postgresql/data \
  -d postgres:16
# Output:
# 7f8e9a0b1c2d3e4f5a6b7c8d9e0f1a2b3c4d5e6f7a8b9c0d1e2f3a4b5c6d7e8f

# Write some real data.
docker exec -it learning-platform-db psql -U postgres -c \
  "CREATE TABLE proof (id SERIAL PRIMARY KEY, note TEXT); INSERT INTO proof (note) VALUES
('still here after removal');"
# Output:
# CREATE TABLE
# INSERT 0 1

# Destroy the container entirely -- not just stop it.
docker stop learning-platform-db
docker rm learning-platform-db
# Output:
# learning-platform-db
# learning-platform-db

# The container is gone. The volume is not -- confirm it's still listed.
docker volume ls
# Output:
# DRIVER    VOLUME NAME
# local     learning-platform-db-data

# Start a brand-new container, pointed at the SAME volume.
docker run --name learning-platform-db \
  -e POSTGRES_PASSWORD=secret \
  -v learning-platform-db-data:/var/lib/postgresql/data \
  -d postgres:16
# Output:
# 3c4d5e6f7a8b9c0d1e2f3a4b5c6d7e8f9a0b1c2d3e4f5a6b7c8d9e0f1a2b3c4

docker exec -it learning-platform-db psql -U postgres -c "SELECT * FROM proof;"
# Output:
#  id |            note
# ----+----------------------------
#   1 | still here after removal
# (1 row)

# The row survived a full container removal -- because it was never stored
# inside the container's own filesystem to begin with.
```

This is the missing piece from "Docker Networking" — the same `learning-platform-db` container from that lesson, now with a named volume attached, is what makes a `docker rm`

(accidental or intentional) safe to run without losing this project's actual data. "Docker Compose," the next lesson in this category, is where a volume like this stops being typed by hand and becomes part of a declared, repeatable setup.

Common Mistakes

- Running PostgreSQL (or any stateful service) in a container with no volume mounted, and being surprised the data is gone after an ordinary `docker rm` — see "Why Containers Are Ephemeral."
- Mounting a volume at the wrong path — a volume only captures what's written to the exact path it's mounted at; `/var/lib/postgresql/data` specifically, for the official `postgres` image (see "Where PostgreSQL Stores Its Data").
- Reaching for a bind mount when a named volume is the right tool, or vice versa — a bind mount ties a container to one specific host path; a named volume is Docker-managed storage meant for exactly this kind of data.
- Running `docker volume rm` without checking whether it's the volume actually holding data still in use — unlike `docker rm` on a container, this one has no equivalent "just the running instance" safety net.

Best Practices

- Always mount a named volume for any container running a real database — PostgreSQL included — never rely on a container's own writable layer for data meant to persist.
- Verify persistence deliberately, the way "Verifying Persistence" and "Putting It All Together" did — write data, remove the container, recreate it against the same volume, confirm the data survived, rather than assuming a volume is correctly configured.
- Reach for a bind mount specifically for host source code during local development, and a named volume specifically for data a container itself manages — don't reach for either one by habit without the distinction in mind.
- Treat `docker volume rm` with the same care as any other genuinely destructive command — confirm nothing still needs that volume's data before running it.

Summary, Cheat Sheet, and Glossary

Summary

- A container's own filesystem writes live in its disposable writable layer, deleted along with everything else the moment `docker rm` runs — this is what makes a container "ephemeral."
- A named volume (`docker volume create`) is Docker-managed storage that exists independently of any single container, and survives exactly the event that destroys a container's writable layer.

- `-v <volume-name>:<path>` on `docker run` mounts a volume at a specific path inside a container — for the official `postgres` image, that path is `/var/lib/postgresql/data`.
- A bind mount maps a specific host path directly into a container (useful for source code during development); a named volume is Docker-managed storage meant for data a container's lifecycle shouldn't determine.
- `docker volume rm` is the one command that genuinely, permanently deletes persisted data — unlike removing the container itself, which a correctly configured volume is specifically meant to survive.

Cheat Sheet

```
docker volume create <name>                    # create a named volume
docker run -v <name>:<path-in-container> ...    # mount it into a container
docker volume ls                                # list all volumes
docker volume inspect <name>                    # show where it actually lives on disk
docker volume rm <name>                         # permanently delete a volume's data
```

```
postgres data path: /var/lib/postgresql/data
```

Glossary

- **Writable layer**: the disposable, per-container filesystem layer sitting on top of a read-only image; deleted along with the container by `docker rm`.
- **Named volume**: Docker-managed storage created independently of any container, which survives `docker rm` and can be mounted into a new container to resume from where the previous one left off.
- **Bind mount**: mapping a specific, literal host filesystem path directly into a container — for when the host path itself matters, unlike a named volume.
- **Mount**: attaching a volume (or bind mount) to a specific path inside a container via `-v`, making reads and writes at that path go to the volume instead of the container's own writable layer.