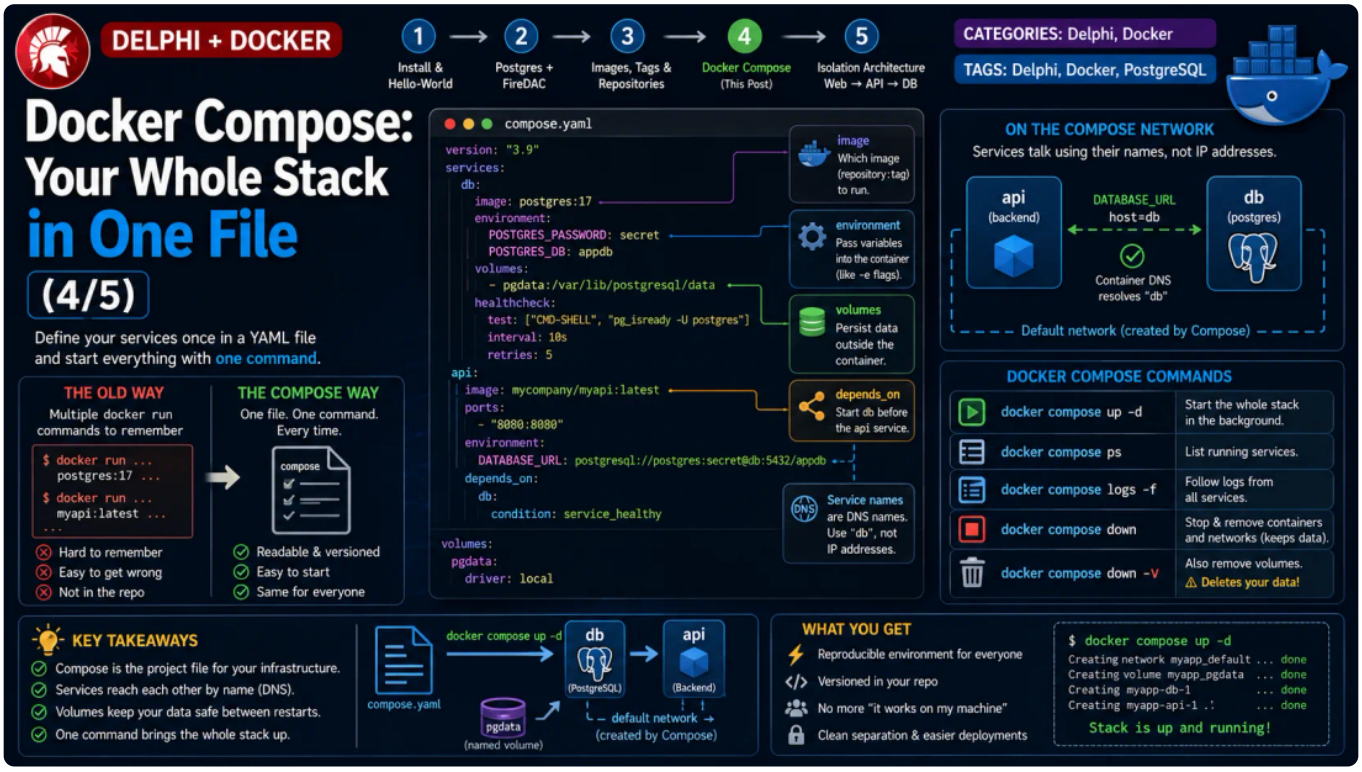
NEWS · AUGUST 28, 2026

Docker Compose: Your Whole Stack in One File (4/5)

By Dr. Holger Flick



Part 3 left you fluent in images, tags, and repositories — you can pull a specific `postgres:17` build, tag your own images, and push them somewhere your team can pull them back. You can start any single container with confidence. And that's exactly where a quiet frustration starts to creep in.

Because a real application is never *one* container. It's a database **and** a backend. Maybe a cache too. Starting each one by hand means memorizing a paragraph of `docker run` flags per service — the right image, the port map, four `-e` environment variables, a `--network`, a volume mount — and running them in the right order, every time, on every machine. Miss one flag and the backend can't find the database. That doesn't scale past a coffee break.

There's a tool built to erase that entire class of problem, and it ships inside the Docker you already installed in Part 1. It's called **Docker Compose**, and by the end of this post you'll describe your whole stack — a Postgres database and a backend that talks to it — in a single readable file, and bring it all up with one command.

This is Part 4 of a five-part journey. Here's the full map so you always know where you are.

> **This series: Docker for Delphi Developers**
> 1. Install Docker and run your first container — hello-world, the daemon, the basics 2. A Postgres container and FireDAC from the command line — a real database in seconds 3. Images, tags, and repositories —

> where images come from and how versions work 4. **Docker Compose: your whole stack in one file — this post** — a database and a backend, together 5. The isolation architecture — web ’ backend ’ database, and why only the backend touches the data

The pain point, named

Let me put the frustration on the table honestly, because naming it is half the argument for Compose. Here is roughly what it takes to start a Postgres database and a backend that connects to it, by hand, the Part 1–3 way.

```
docker network create appnet

docker run -d --name db --network appnet \
  -e POSTGRES_PASSWORD=secret -e POSTGRES_DB=appdb \
  -v pgdata:/var/lib/postgresql/data \
  postgres:17

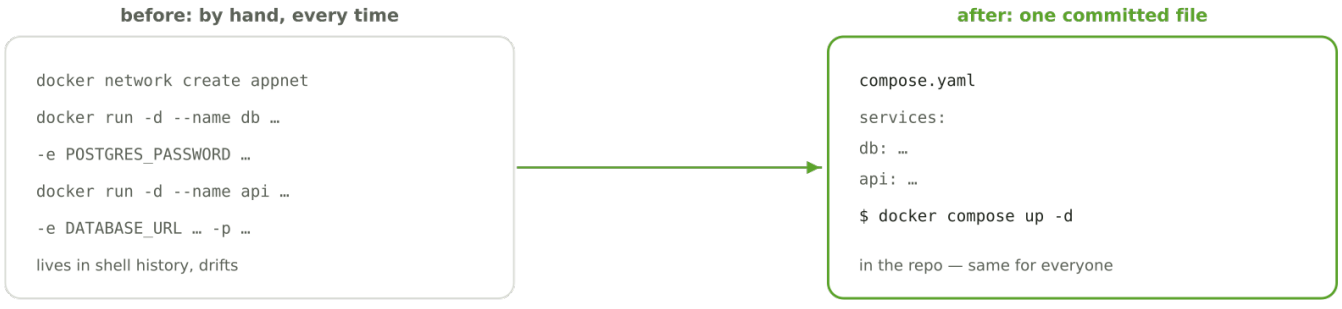
docker run -d --name api --network appnet \
  -e DATABASE_URL=postgres://postgres:secret@db:5432/appdb \
  -p 8080:8080 \
  my-backend:1.0
```

Every one of those lines is correct — but it lives in your shell history, not in your project. A teammate who clones the repo has no idea these commands exist, let alone their exact flags. Change the database password and you're editing two commands in lockstep. This is infrastructure defined by oral tradition, and oral tradition drifts.

What Compose actually is

Docker Compose is, in Docker's own words, a tool for defining and running multi-container applications — you write the whole stack down as one declarative YAML file (YAML being a plain-text format for structured configuration, all indentation and `key: value` pairs), and start it with a single command. Everything from the `docker run` soup above becomes a few lines of readable text that lives *in your repository*, next to your code.

Here is the mental model that will make this click for a Delphi developer: **Compose is the project file for your infrastructure.** You don't rebuild your `.dproj` by remembering compiler switches every morning; you open the project and everything is declared. Compose does the same for your services. Declare the stack once, commit it, and every teammate — and every server — runs the identical thing with one command.



docker run soup becomes one declared, committed file

The right-hand box is the entire goal of this post. Notice it isn't *less* capable than the left — it does everything the commands did, including creating the network and the volume for you. It's just written down where it belongs.

> **Compose is built into Docker now**
> You do not install anything extra. Per the official install docs, *"Docker Desktop includes Docker Compose along with Docker Engine and Docker CLI"* — Compose ships as the modern `docker compose` command (a v2 CLI plugin, note the space, not the older hyphenated `docker-compose`). If you finished Part 1, you already have it. Check with `docker compose version`.

Building `compose.yaml`, one service at a time

The best way to learn the file is to grow it. We'll start with the database alone, run it, then add the backend — introducing each new YAML key the first time it appears (that's the promise of this series: no unexplained magic). The file is conventionally named `compose.yaml` and sits in your project root.

Step 1 — Declare the database service

Every Compose file has a top-level `services:` block; each key under it is one service (one container, essentially) with a name you choose. Here's a `db` service running the same Postgres image from Part 2.

```
services:
  db:
    image: postgres:17
    environment:
      POSTGRES_PASSWORD: secret
      POSTGRES_DB: appdb
```

Three keys, each doing one job. `image:` names the exact image to run — `postgres:17`, the very `repository:tag` form you learned in Part 3. `environment:` passes environment variables into the container, exactly like the `-e` flags from Part 2 — here the official Postgres image reads `POSTGRES_PASSWORD` to set the superuser password and `POSTGRES_DB` to create a database named `appdb` on first start. That is a fully runnable Compose file already.

Step 2 — Add the backend service

Now we add a second service, `api`, alongside `db` under the same `services:` block. This is the backend your Delphi app (and a web frontend) will call — we'll build it for real in Part 5, so treat the image name here as a placeholder for "our backend."

```
services:
  db:
    image: postgres:17
    environment:
      POSTGRES_PASSWORD: secret
      POSTGRES_DB: appdb

  api:
    image: my-backend:1.0        # placeholder — Part 5 builds this for real
    ports:
      - "8080:8080"
    environment:
      DATABASE_URL: postgres://postgres:secret@db:5432/appdb
    depends_on:
      - db
```

Three new keys earn their introduction here. `ports:` publishes a container port to your host in the familiar `"host:container"` shape from Part 2 — `"8080:8080"` means "requests to `localhost:8080` on my machine reach port 8080 inside the `api` container." `depends_on:` tells Compose to start `db` before `api`, so the ordering you used to manage by hand is now declared. And look closely at that `DATABASE_URL` — the host is literally `db`. That one word is the payoff of this whole post, and it deserves its own section.

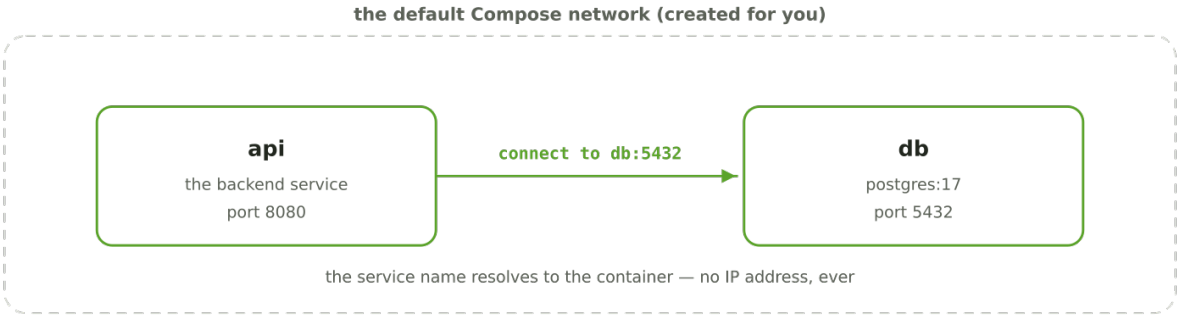
> **image: to run one, build: to build one**
> Our `api` here uses `image:` to run a pre-built image. If you want Compose to build an image from a local `Dockerfile` instead, you swap `image:` for `build: .` — Compose builds it for you on `up`. We'll lean on `build:` in Part 5 when the backend becomes real; for now, `image:` keeps the focus on the stack, not the build.

The payoff: services find each other by name

Here's the frustration Compose quietly deletes. By hand, if you wanted the backend to reach the database, you had to know the database container's IP address — an address Docker assigns dynamically and that changes every time the container is recreated. On a Compose network, you never touch an IP again.

Compose puts every service in your file onto one shared network automatically, and — this is the magic — **each service is reachable by its service name as a hostname.** Our `api` connects to the database at host `db` because that's the service's name in the YAML. Not an IP. The name.

This is not a Compose invention; it's container DNS. The docs state it plainly: *"Each service registers its name with an internal DNS server, so containers can reach each other using the service name directly"* — and, crucially, *"you should always reference services by name, not IP address"* because the name stays constant even when a container is recreated with a fresh IP.



On the Compose network, api reaches the database by the name 'db' — no IP address anywhere

If you followed the networking series, this is not a new idea at all — it's the exact DNS lesson from Part 2 of that series, where a *name* resolves to an *address* so you never hardcode the number. Docker gives each of its container networks a built-in DNS server that does precisely this for service names. You already understood the mechanism; Compose just applies it inside your stack.

> **Why this matters for your Delphi backend**
> In Part 5, when the backend connects to Postgres, its connection string will point at host `db` — a name that is stable, readable, and identical on every developer's laptop and on the production server. Nobody ever chases a changed IP address again. That's the reliability dividend of container DNS.

Volumes: the persistence answer we promised in Part 2

Back in Part 2, a nagging question hung in the air: when you stop and remove the Postgres container, where does the data go? The honest answer was *"it's gone"* — a container's own filesystem is ephemeral, wiped when the container is removed. Compose lets us fix that cleanly and permanently, and this is the right moment to do it.

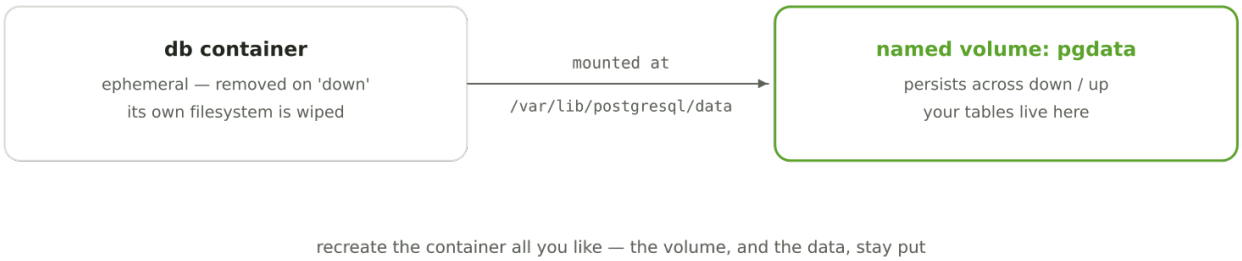
The fix is a **volume**: a piece of storage that lives *outside* any single container's lifecycle, so it survives the container being destroyed and recreated. You mount it at the exact directory where Postgres keeps its data, and the data persists across restarts. Here's our file with a named volume added.

```
services:
  db:
    image: postgres:17
    environment:
      POSTGRES_PASSWORD: secret
      POSTGRES_DB: appdb
    volumes:
      - pgdata:/var/lib/postgresql/data

  api:
    image: my-backend:1.0
    ports:
      - "8080:8080"
    environment:
      DATABASE_URL: postgres://postgres:secret@db:5432/appdb
    depends_on:
      - db

volumes:
  pgdata:
```

Two additions do the work. The service-level `volumes:` line mounts a volume named `pgdata` at `/var/lib/postgresql/data` — and that path is not arbitrary. The official Postgres image is explicit: *"Mount the data volume at `/var/lib/postgresql/data`"* for Postgres 17 and earlier, because that's the directory the database actually writes to. The separate top-level `volumes:` block at the bottom **declares** `pgdata` as a named volume that Docker manages for you. Now stop the stack, start it again, and your tables are still there.



The container is disposable; the named volume is not — data outlives the container

The takeaway from that picture: the container and its data are now *separate things*. Throw the container away and rebuild it on a newer Postgres tag — the volume, and every row in it, remains. Leave the volume out of the file, and Postgres falls back to an anonymous volume that isn't reused, which is the ephemeral behavior we saw in Part 2.

> **Named volume vs. bind mount**
> There are two ways to mount storage. A **named volume** (`pgdata:` here) is storage Docker owns and manages on your behalf — ideal for database data, portable, and the recommended default. A **bind mount** maps a specific folder on *your host* into the container (`./local/path:/container/path`) — perfect for editing source code live during development, less so for database files. For Postgres persistence, reach for the named volume; both are documented in Docker's storage guide.

The lifecycle: five commands you'll actually use

With `compose.yaml` written, running the whole stack is a handful of commands, all under `docker compose`. Here they are in the order you'll meet them day to day.

Step 1 — Start everything in the background

Run `up` with `-d` (detached) to build the network, create the volume, and start every service, returning your prompt immediately.

```
docker compose up -d
```

Compose reads `compose.yaml` in the current directory, creates the shared network and the `pgdata` volume, then starts `db` and `api` — honoring `depends_on` so the database comes up first.

Step 2 — See what's running

Check the status of the services in this stack with `ps`.

```
docker compose ps
```

This lists just *your* stack's containers and their state — a focused view, not every container on the machine.

Step 3 — Read the logs

Follow the combined output of all services with `logs -f`, which is where you'll watch the database accept its first connection.

```
docker compose logs -f
```

Both services' output is interleaved and labelled by service name; drop the `-f` for a one-time dump instead of a live follow.

Step 4 — Stop and clean up (keeping your data)

Tear the stack down with `down`, which — per the CLI reference — *"Stops and removes containers, networks."*

```
docker compose down
```

This removes the containers and the network but **leaves your named volume alone** — so `docker compose up -d` again brings everything back with your data intact. This is the everyday stop.

Step 5 — The destructive one (deletes your data)

There's one flag that changes everything, and it deserves a clear warning.

```
docker compose down -v
```

> **down -v deletes your volume — and your data**
>
> The `-v` (`--volumes`) flag tells `down` to also remove the named volumes declared in your `volumes:` block. That means `pgdata` is deleted and **every row in your database is gone.** Use it deliberately — to reset to a clean slate on purpose — never out of habit. Plain `docker compose down` (no `-v`) is the safe everyday command; `down -v` is the "start completely fresh" command.

Where Compose fits — and where teams reach for more

An honest word on scope, because Compose is excellent but not the answer to every question. Compose is superb for local development and for running a stack on a single host — which covers an enormous amount of real, valuable work, including everything in this series.

When you need to run across *many* hosts with automatic failover, rolling updates, and self-healing — production orchestration at scale — teams reach for other tools, and they're worth knowing by name. Kubernetes is the industry standard for large multi-host orchestration; Docker Swarm offers clustering with a Compose-like feel; and Podman with `podman-compose` is a strong daemonless, open-source path that reads the same Compose files. Each is a great fit for its job. For learning your stack and running it locally — the goal of this series — Compose is exactly the right tool, and the concepts carry straight over when you grow.

Takeaways

The `docker run` soup is behind you. Your database and your backend now live in one readable `compose.yaml`, committed to your repo, started with a single command, and identical on every machine that runs it. Along the way you got the two answers this series had been saving: services reach each other by **service name** thanks to container DNS — no IP addresses, ever — and a **named volume** at Postgres's data directory makes your data survive across restarts. Five commands (`up -d`, `ps`, `logs`, `down`, and the deliberate `down -v`) run the whole lifecycle.

> Compose is the project file for your infrastructure: declare the stack once, and everyone — every laptop, every server — runs the same database, the same backend, wired together by name, with the same command.

We now have a database and a backend running side by side, talking over the Compose network. But should they *both* be reachable? In Part 5 we answer the architecture question this setup raises: why only the backend should ever touch the database, and how the full web ’ backend ’ database picture — a browser and your Delphi app in front, an XData or Express/TypeScript backend in the middle, Postgres locked safely behind it — comes together into a design you can trust. See you there.

INDEPENDENT · AD-FREE · READER-SUPPORTED

Free to read. Not free to make.

Every tutorial, article, and line of example code on Holger's Code is published without a paywall and without ads. This page is about what keeps it that way.

Professional work, published free

The content here is held to the same standard as professional client work: every tutorial is built as a real project, tested against current versions, documented step by step, and revisited when the frameworks move on. A single in-depth tutorial routinely takes days — and that's before the bills behind it: servers, storage, build infrastructure, licenses, and the hardware everything is verified on.

If an article or tutorial here has saved you an afternoon of debugging, taught you a technique, or helped you ship, it created real value for you. Supporting it isn't charity — it's paying for professional work you found useful, at whatever price you choose. And if you can't, keep reading anyway. That's what the free part means.

Ways to support

KO-FI

Buy me a coffee

A one-time tip — no account, no subscription, any amount.

BOOKS & COURSES

Buy the paid work

The paid tier of everything published free here — and you get something lasting in return.

SPREAD THE WORD

Share an article

Send a tutorial to a colleague, cite an article, subscribe to the RSS feed.

Nothing here goes behind a paywall, whether you chip in or not. But if something on this site earned its keep today, this is where to say so.

Support on Ko-fi —