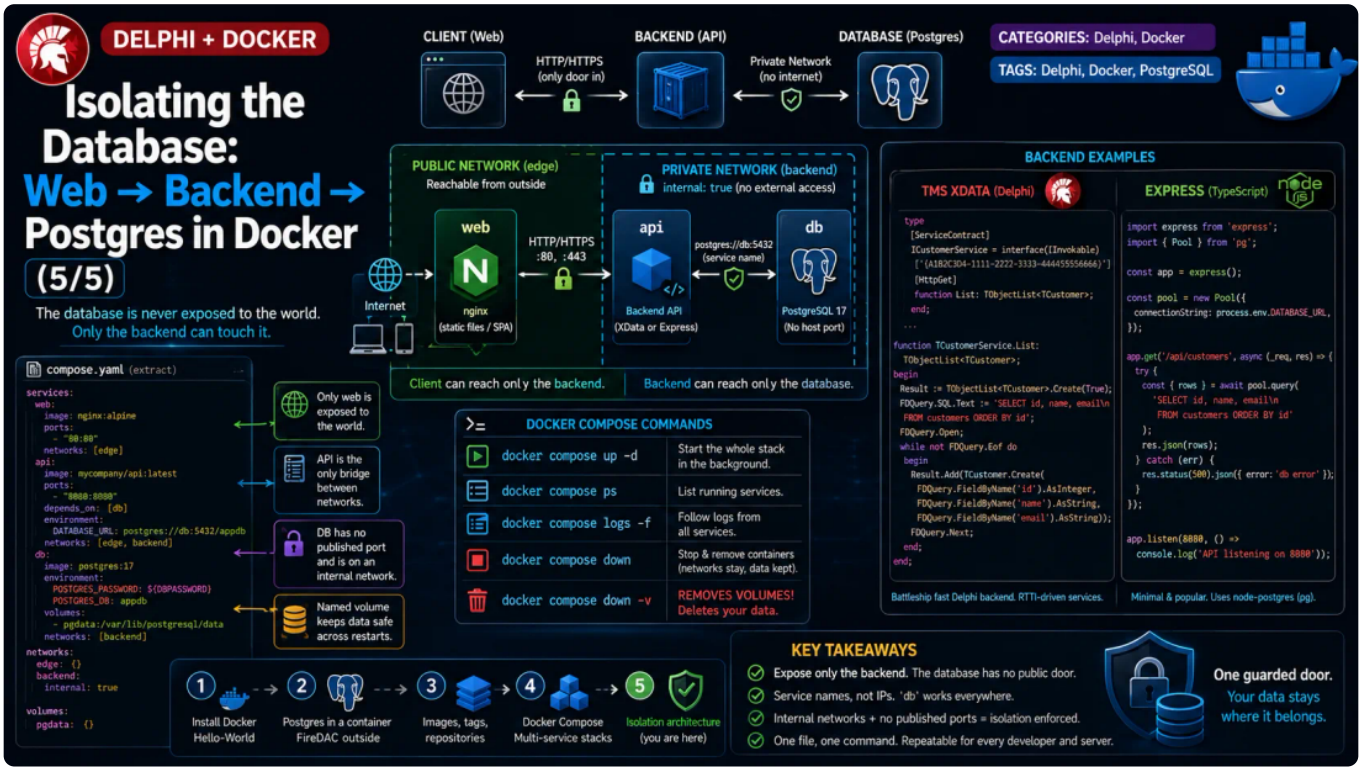
NEWS · AUGUST 31, 2026

Isolating the Database: Web ’Backend ’Postgres in Docker (5/5)

By Dr. Holger Flick



Part 4 left us with a working Docker Compose stack: several services in one `compose.yaml`, finding each other by name on a shared network, with a named volume keeping Postgres data safe across restarts. It ran. It was tidy. And it had one quiet flaw I deliberately left in place so we could fix it properly today.

The flaw: our database was reachable from the host. Convenient for poking at it with a client — and exactly the shape you must *not* ship. Here is the thesis of this finale, stated plainly:

In a real system, the database is never exposed to the world. Only the server-side backend may touch it. The web client talks only to the backend, over HTTP. The backend alone holds the database credentials and reaches Postgres over a private network no one else can see.

That is the standard, safe layering behind essentially every production web application you've ever used — three tiers, one guarded doorway. In this post we'll build it in Compose, prove the isolation is real, and show the backend in *both* stacks I reach for: TMS XData in Delphi and Express in TypeScript. Same architecture, swappable box.

This is the last stop on a five-part road, so here's the whole map before we build the finale.

> **This series: Docker for Delphi Developers**

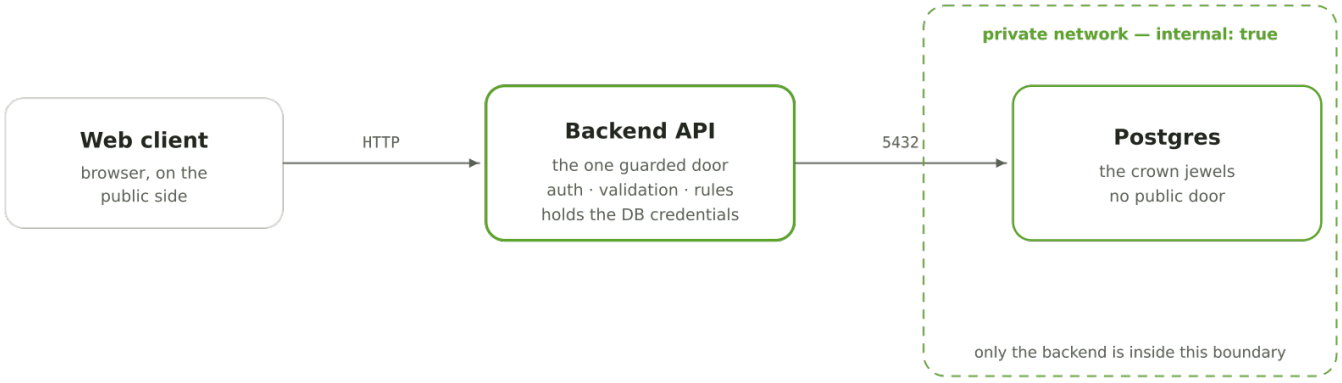
1. Install Docker and run your first container — hello-world, demystified 2. Postgres in a container, FireDAC on the outside — a real database in seconds 3. Images, tags, and repositories — where containers come from 4. Docker Compose: multi-service stacks — service-name networking and volumes 5. **The isolation architecture — this post** — web client ’ backend ’ database, done safely

Why isolation matters — the database is the crown jewels

Let's start with the intuition, framed the way I'd frame it for a client rather than as a scare story. Your database holds the single most valuable thing your business owns: its data. Customers, invoices, credentials, everything. So the question isn't "is my data important" — it obviously is — it's "how many doors lead to it?"

Every door is a thing you must guard. A database port open to the network is a door. If that port is reachable from the internet, then *anyone who can reach the port* gets to negotiate directly with your database engine — no application logic, no permission checks written in your code, nothing between them and the raw tables but the database's own login. That's a big, undifferentiated door, and it's the wrong place to be doing your defending.

The elegant move is to have **exactly one door**, and to make it a smart one: the backend. The backend is a single, guarded entrance that you control completely. It enforces authentication ("who are you?"), validation ("is this request even sensible?"), and your business rules ("this user may read their own invoices, not everyone's"). The web client never speaks to the database. It asks the backend, politely, over HTTP — and the backend decides what, if anything, the database is asked to do.



Three tiers, one guarded doorway — only the backend crosses into the database's private network

Read the diagram left to right, and notice the dashed boundary on the right. Postgres and the backend share a private network the client cannot enter; the client's only reach is the HTTP arrow into the backend. There is no line from the web client to the database — not because we politely asked it not to draw one, but because, as you'll see next, Docker makes that line physically impossible to draw.

The one-sentence version

The database is a room with a single door, and the backend is the only one holding the key. Everyone else knocks at the backend and asks nicely.

Mapping it onto Docker Compose

Here is where Docker stops being an abstraction and does the enforcing for us. We'll express the exact three-tier picture above as three services and two networks in one `compose.yaml`. This is the "show, don't tell" payoff: the architecture becomes a few lines of YAML, and the isolation is a *fact of the runtime*, not a convention we hope everyone honors.

The whole trick rests on two Compose behaviors, both straight from the official Compose networking docs:

- **Service names are hostnames.** Services on a shared network find each other by service name through Docker's built-in DNS — the backend reaches the database at `postgres://db:5432`, no IP addresses anywhere. (That's exactly the service-name networking from Part 4.)
- **No published port means no host access.** The docs are explicit: "Networked service-to-service communication uses the `CONTAINER_PORT`. The host port is only used when accessing the service from outside the network." So a service with *no* `ports:` mapping is fully reachable by other containers on its network, and completely unreachable from your host or the outside world.

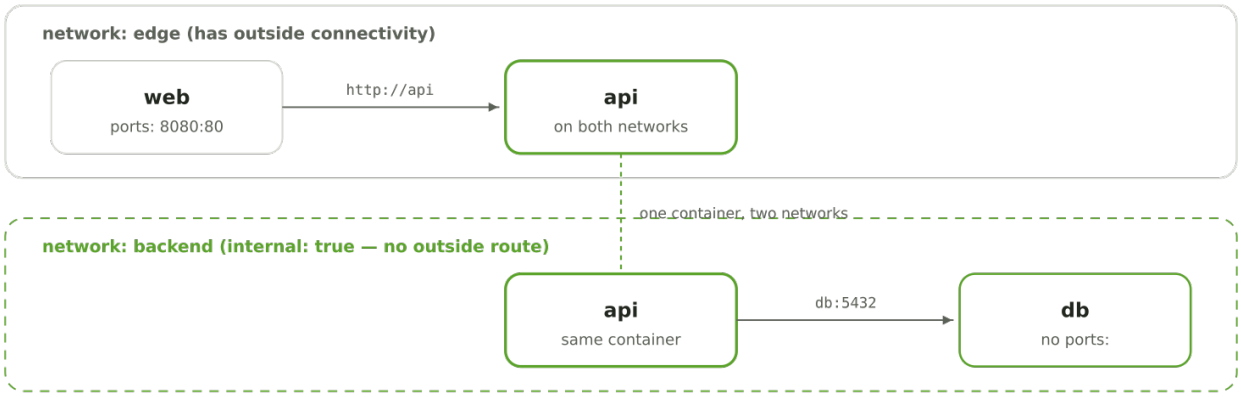
We add one more guarantee on top with an **internal network**. Per the Compose networks reference: "By default, Compose provides external connectivity to networks. `internal`, when set to `true`, lets you create an externally isolated network." An internal network has no gateway to the outside — nothing on it can reach out, and nothing outside can reach in. It's the digital equivalent of a room with no exterior windows.

Here is the stack. Read it once top to bottom; the annotations after each service explain the load-bearing keys.

```
services:
  web:
    # The public entry point. In many stacks this simply serves the browser
    # app (static files or a Next.js/SPA server). The browser is the real
    # "client" tier; this service just hands it the page.
    image: my-web:latest
    ports:
      - "8080:80"          # host 8080 -> container 80: reachable from outside
    networks:
      - edge               # shares a network with the api, and only the api

  api:
    # The backend — the single guarded door. It is the ONLY service that
    # sits on BOTH networks, so it is the only thing that can talk to db.
    image: my-api:latest
    environment:
      DATABASE_URL: "postgres://appuser:${DB_PASSWORD}@db:5432/appdb"
    networks:
      - edge               # so the web tier can reach it
      - backend            # so it (and only it) can reach the database
    depends_on:
      - db

  db:
    image: postgres:17
    environment:
      POSTGRES_USER: appuser
      POSTGRES_DB: appdb
      POSTGRES_PASSWORD: ${DB_PASSWORD}
    # NOTE: no `ports:` here. Nothing on the host can reach 5432.
    volumes:
      - dbdata:/var/lib/postgresql/data
    networks:
      - backend            # ONLY on the internal network

networks:
```

```
  edge:                    # web <-> api; has normal outside connectivity
    driver: bridge
  backend:
    driver: bridge
    internal: true         # <-- the private room: no route to/from the world

volumes:
  dbdata:
```

Walk the network topology and the security property falls right out. The `db` service is on the `backend` network only, and `backend` is `internal: true` — so the database has no route to or from the host or the internet at all. It published no ports, so even if `backend` weren't internal, the host still couldn't reach `5432`. The `api` service is the sole member of *both* `edge` and `backend`, which is precisely what makes it the only door: it's the one service that can hear the `web` tier on `edge` and speak to `db` on `backend`. And the `web` service lives only on `edge`, so it has no path to `db` whatsoever — try to open `db:5432` from `web` and Docker will simply refuse to route it.



The same three tiers as Compose networks — the api is the only service on both

The diagram shows the same `api` box straddling both bands — that's the whole architecture in one glance. Because `api` is the only member of the `backend` band, and `web` is not on it at all, there is no drawable line from `web` to `db`. Docker enforces the layering; you don't have to trust anyone to respect it.

> **Prove it to yourself in ten seconds**
> Bring the stack up and try to reach the database from your host. `psql -h localhost -p 5432` — the connection is refused, because nothing is published on 5432. Now `docker compose exec api sh` and connect from inside the backend: it works, because the `api` container is on the `backend` network. Seeing the refusal with your own eyes is what turns "isolation" from a word into a fact you trust.

The backend, in both stacks

Now for the box in the middle. I keep two go-to backends for this exact architecture, and I want to be clear up front: **neither is the "right" one.** They're the right ones for *different shops*, and both drop into the `api` slot above without changing a single line of the Compose file. The architecture is language-agnostic; that's the entire point.

Let me show the same endpoint — "give me the list of customers, read from Postgres" — in each.

TMS XData (Delphi)

If your team lives in Delphi, TMS XData — a REST/JSON server framework built on the TMS Sparkle HTTP framework — keeps your backend in the language and tooling you already know. Its central idea, which I covered in the networking series finale, is that you declare a **service contract as an ordinary Delphi interface** and the framework turns it into HTTP endpoints. Per the official service operations docs, the contract looks like this:

```
uses
  XData.Service.Common, Generics.Collections;

type
  TCustomer = class
    Id: Integer;
    Name: string;
  end;

  [ServiceContract]
  ICustomerService = interface(IInvokable)
    ['{7B1E2F0C-5A44-4E9D-9C1B-2F6A8D3B4E71}']
    [HttpGet] function List: TList<TCustomer>;
  end;

initialization
  RegisterServiceType(TypeInfo(ICustomerService));
```

One line per idea: `[ServiceContract]` marks the interface as a service, `IInvokable` plus the GUID gives the framework the RTTI it needs, `[HttpGet]` binds `List` to a GET request (by default XData operations respond to POST, so this attribute is our REST-verb dial), and `RegisterServiceType` announces the interface. The implementation is an ordinary class that reads from Postgres and returns objects — XData serializes `TList<T>` to JSON for you automatically:

```
uses
  XData.Server.Module, XData.Service.Common, Generics.Collections;

type
  [ServiceImplementation]
  TCustomerService = class(TInterfacedObject, ICustomerService)
  public
    function List: TList<TCustomer>;
  end;

function TCustomerService.List: TList<TCustomer>;
begin
  Result := TList<TCustomer>.Create;
  // Read from Postgres here and populate Result.
  // The DB connection uses the DATABASE_URL / host `db` from Compose —
  // the credentials live in this backend, nowhere else.
end;

initialization
  RegisterServiceType(TCustomerService);
```

The database access itself is FireDAC, exactly as we set it up in Part 2 — the same connection, now pointed at host `db` instead of `localhost`, since Compose gives us that name. One honest caveat worth stating: when you containerize a Delphi backend on Linux, FireDAC's Postgres driver needs the Postgres client library (`libpq`) present in the image, precisely the driver detail covered in Part 2. That's a packaging step, not a rearchitecture — and TMS documents running XData/Sparkle servers on Linux directly, so the standalone HTTP server containerizes cleanly once the driver is in the image.

> **Where the XData server runs**
> An XData server is a standalone HTTP server — it owns its port and answers requests on its own, with no external web server required — which is exactly why it drops into the `api` slot of our Compose stack. The specific Sparkle host you pick (and the Linux packaging with `libpq`) is a deployment detail; the *architecture* — one backend that owns the database — is unchanged.

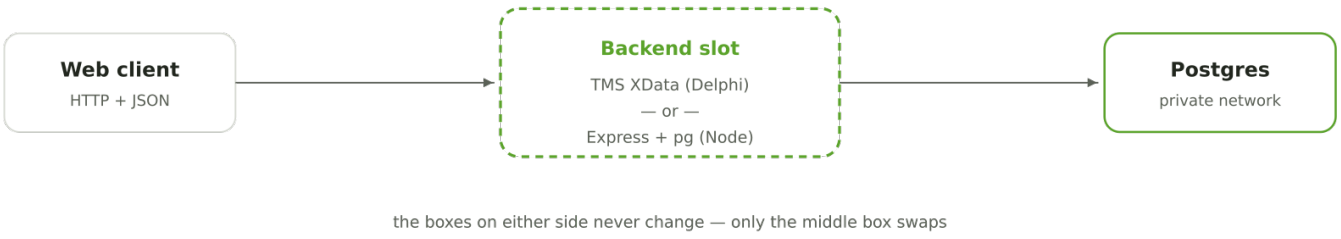
Express + TypeScript (Node)

If your team is a JavaScript/TypeScript shop, or you want the enormous npm ecosystem at your back, Express — the minimal, hugely popular Node.js web framework — gives you the identical endpoint with very little ceremony, talking to Postgres through node-postgres (the `pg` package, the standard Postgres client for Node). Per the node-postgres docs, a pooled query is `pool.query(...)` returning `.rows`:

```
import express from "express";
import { Pool } from "pg";

// The pool reads DATABASE_URL from the environment — the same connection
// string Compose injected. Credentials live here, in the backend, only.
const pool = new Pool({ connectionString: process.env.DATABASE_URL });

const app = express();

app.get("/customers", async (_req, res) => {
  const result = await pool.query("SELECT id, name FROM customers ORDER BY id");
  res.json(result.rows); // the JSON contract the web client consumes
});

app.listen(3000, () => console.log("api listening on 3000"));
```

Same shape as the Delphi version: a GET endpoint, a read from Postgres over the private network, a JSON response. The `Pool` connects to host `db` via `DATABASE_URL` — and just like the XData backend, this is the only place the database password ever lives.

Here is the part I most want to land: these two backends are *interchangeable inside the same architecture.*



Same architecture, swappable backend — XData or Express drops into one slot

The dashed box is the point: the web client and the database don't know or care what language fills the middle. Both backends containerize identically, both attach to the same two networks, both hold the credentials, both answer the same HTTP. So choose by *fit*, not by winner:

- **Delphi shop, existing FireDAC/business logic, want commercial support?** Reach for XData — you stay in one language, your contracts are RTTI-driven Delphi interfaces, and your existing data-access code from Part 2 comes along for the ride.
- **JS/TS team, or you want the npm ecosystem and a huge hiring pool?** Reach for Express — minimal, ubiquitous, and `pg` is a rock-solid Postgres client.

Both are excellent at what they do, and both leave the architecture — the thing that actually protects your data — exactly the same.

> **The wider landscape**
> Fairness demands naming the neighbors, because this pattern is universal. In Delphi, the open-source DelphiMVCFramework builds these same REST/JSON backends and is a genuinely strong choice when a commercial license doesn't fit. Beyond these two, teams fill the identical backend slot every day with Fastify (Node), FastAPI (Python), and ASP.NET Core (C#). Every one of them isolates a database behind an internal network in Docker exactly as we did above — the architecture is what transfers, unchanged.

What we're deferring — on purpose

I've kept this stack deliberately minimal so the *shape* stays visible, and honesty requires naming what a real production deployment adds on top. None of these change the architecture you just built; each is a layer you fit around it.

> **Production adds these — named, linked, and left for another day**
> - **TLS/HTTPS at the edge.** Everything here speaks plain HTTP, which is fine inside the private network but must be encrypted at the public boundary. See HTTPS/TLS. - **Real secrets management.** We used a plaintext `${DB_PASSWORD}` env var for clarity, but the Compose best-practices docs are blunt: "Don't use environment variables to pass sensitive information… Use secrets instead." Prefer Docker secrets, which mount a secret as a read-only file at `/run/secrets/…` rather than exposing it in the environment, and keep configuration in an `.env` file out of source control. - **Authentication.** The backend is the place to enforce it — an API key or token in the `Authorization` header — so the "guarded door" actually checks credentials. - **A reverse proxy at the front.** Tools like nginx or Traefik typically sit at the edge to terminate TLS, route, and load-balance across backend instances. Each of these deserves its own treatment; none of them changes the three-tier isolation we built today.

The whole Docker series, in one arc

Step all the way back. Five posts, and each one poured the floor the next stood on — from a single container to a properly isolated architecture.

The series arc: from your first container to a database no one can touch but your backend

Read it left to right and the payoff is obvious: nothing on the right threw away anything on the left. The isolated architecture in Part 5 *is* the Compose stack of Part 4, wearing the network topology of a grown-up system, running images from Part 3, holding the Postgres of Part 2, started with the commands from Part 1.

And here's the connection I most enjoy pointing out. If any of this felt legible — ports that map, names that resolve, HTTP requests carrying JSON — that's not an accident. It's the exact vocabulary from the Networking for Delphi Developers finale. A published port is Part 1's `IP:port`. Service-name networking is Part 2's DNS.

The backend's endpoint is a Part 5 web service. Docker didn't invent new concepts; it packaged the old ones. You walked through the networking doorway, and it opened straight onto this.

Takeaways

The database is the crown jewels, so it gets exactly one door — the backend — and Docker makes that door the *only* one that exists. Put the database on an `internal: true` network with no published ports, put the backend on both that private network and the public one, and the web client physically cannot reach the database. The backend alone holds the credentials, enforces auth and validation, and speaks the business rules. Fill that backend slot with TMS XData if you're a Delphi shop or Express if you're a TypeScript shop — the choice is fit, not merit, and the architecture doesn't care which you pick.

> A safe system isn't one where the database is well-defended at the door. It's one where the database has no public door at all — and the only thing that can knock is code you wrote and trust.

That's the series. You can now install Docker, run a database, build and tag images, compose a multi-service stack, and lay it out so the most valuable tier is genuinely unreachable except through your own backend. That's not a toy — that's the real shape of production.

> **This series: Docker for Delphi Developers**
> 1. Install Docker and run your first container 2. Postgres in a container, FireDAC on the outside 3. Images, tags, and repositories 4. Docker Compose: multi-service stacks 5. **The isolation architecture — this post**

If you're designing this layer for a Delphi product — deciding where the backend lives, how the database stays private, whether XData or a Node service fits your team — and you'd like a second pair of eyes from someone who has walked this exact road with legacy codebases, let's talk. Your app already works. Let's give it an architecture worthy of it.

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 —