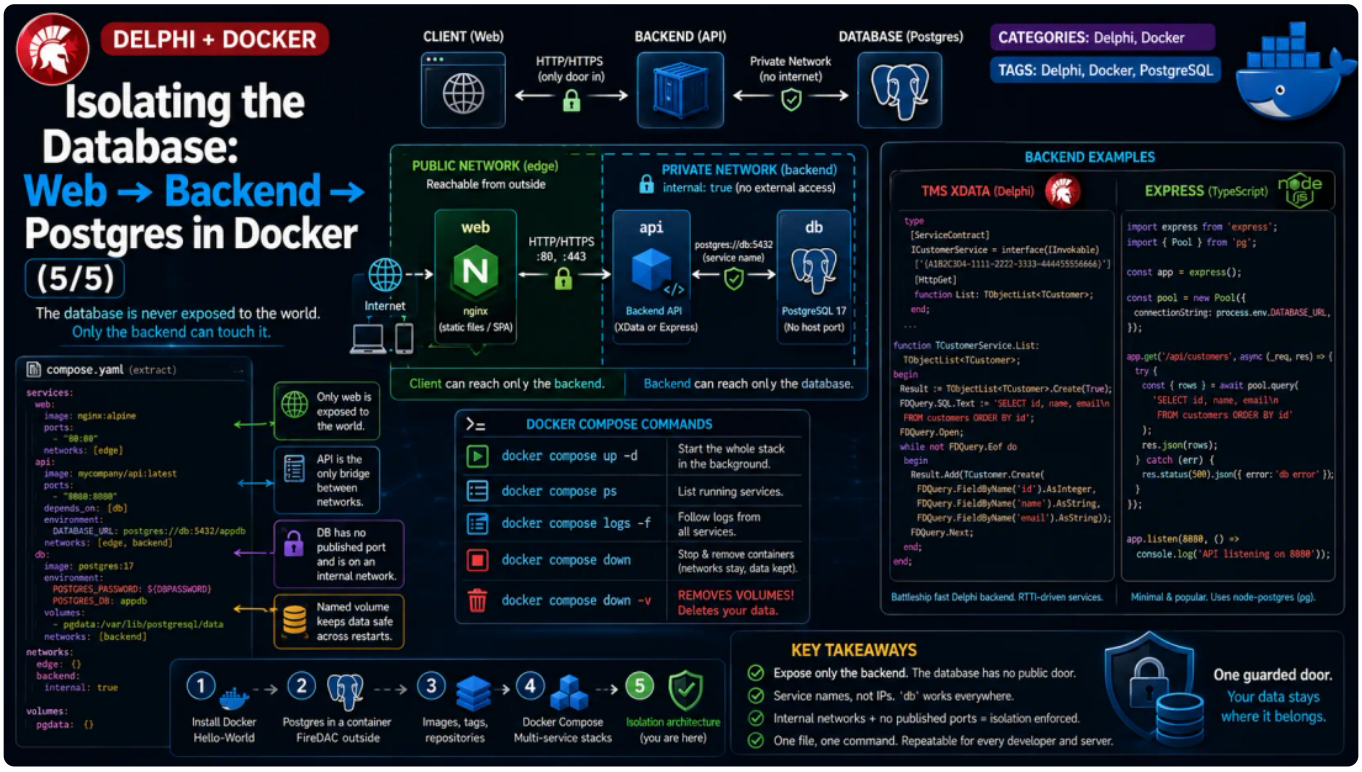
NEWS · AUGUST 31, 2026

Die Datenbank isolieren: Web ’Backend ’ Postgres in Docker (5/5)

By Dr. Holger Flick



Teil 4 hat uns einen funktionierenden Docker-Compose-Stack hinterlassen: mehrere Services in einer `compose.yaml`, die sich über ein gemeinsames Netzwerk beim Namen finden, mit einem benannten Volume, das die Postgres-Daten über Neustarts hinweg sichert. Es lief. Es war aufgeräumt. Und es hatte einen stillen Makel, den ich absichtlich stehen gelassen habe, damit wir ihn heute sauber beheben können.

Der Makel: Unsere Datenbank war vom Host aus erreichbar. Praktisch, um mit einem Client hineinzuschauen — und exakt die Form, die Sie *nicht* ausliefern dürfen. Hier ist die These dieses Finales, klar ausgesprochen:

In einem echten System ist die Datenbank niemals der Welt ausgesetzt. Nur das serverseitige Backend darf sie berühren. Der Web-Client spricht ausschließlich mit dem Backend, über HTTP. Allein das Backend hält die Datenbank-Zugangsdaten und erreicht Postgres über ein privates Netzwerk, das sonst niemand sehen kann.

Das ist die standardmäßige, sichere Schichtung hinter praktisch jeder Produktions-Webanwendung, die Sie je benutzt haben — drei Schichten, eine bewachte Tür. In diesem Beitrag bauen wir sie in Compose, beweisen, dass die Isolation real ist, und zeigen das Backend in *beiden* Stacks, zu denen ich greife: TMS XData in Delphi und Express in TypeScript. Gleiche Architektur, austauschbare Box.

Dies ist die letzte Station eines fünfteiligen Weges, deshalb hier die komplette Karte, bevor wir das Finale bauen.

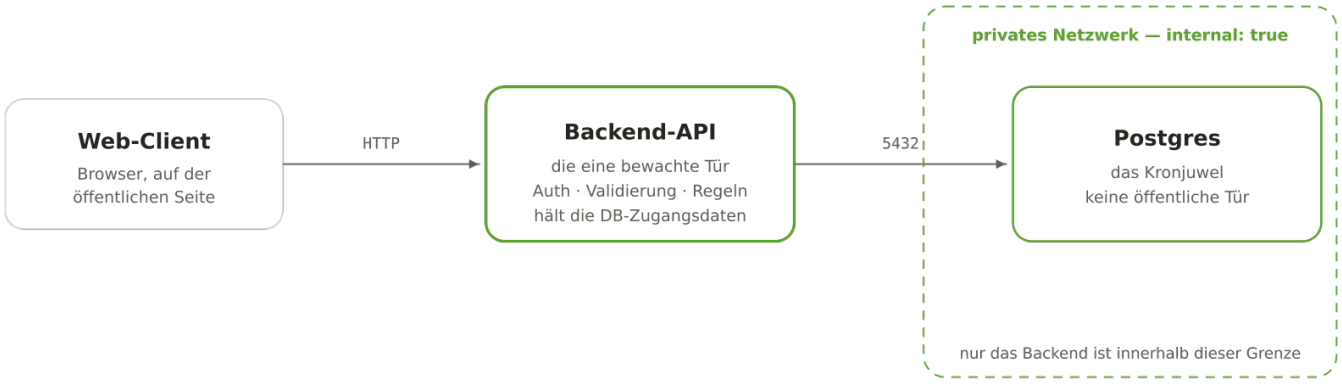
Diese Serie: Docker für Delphi-Entwickler
1. Docker installieren und den ersten Container starten — hello-world, entmystifiziert 2. Postgres im Container, FireDAC außen dran — eine echte Datenbank in Sekunden 3. Images, Tags und Repositories — woher Container kommen 4. Docker Compose: Multi-Service-Stacks — Service-Name-Networking und Volumes 5. **Die Isolationsarchitektur — dieser Beitrag** — Web-Client ’ Backend ’ Datenbank, sicher umgesetzt

Warum Isolation zählt — die Datenbank ist das Kronjuwel

Beginnen wir mit der Intuition, formuliert so, wie ich es einem Kunden erklären würde, nicht als Schauergeschichte. Ihre Datenbank enthält das wertvollste Gut, das Ihr Unternehmen besitzt: seine Daten. Kunden, Rechnungen, Zugangsdaten, alles. Die Frage lautet also nicht „sind meine Daten wichtig" — das sind sie offensichtlich —, sondern „wie viele Türen führen dorthin?"

Jede Tür ist etwas, das Sie bewachen müssen. Ein zum Netzwerk offener Datenbank-Port ist eine Tür. Ist dieser Port aus dem Internet erreichbar, dann darf *jeder, der den Port erreichen kann*, direkt mit Ihrer Datenbank-Engine verhandeln — keine Anwendungslogik, keine in Ihrem Code geschriebenen Berechtigungsprüfungen, nichts zwischen ihm und den rohen Tabellen außer dem Login der Datenbank selbst. Das ist eine große, undifferenzierte Tür, und es ist der falsche Ort, um sich zu verteidigen.

Der elegante Zug ist, **genau eine Tür** zu haben und sie klug zu machen: das Backend. Das Backend ist ein einzelner, bewachter Eingang, den Sie vollständig kontrollieren. Es erzwingt Authentifizierung („wer sind Sie?"), Validierung („ergibt diese Anfrage überhaupt Sinn?") und Ihre Geschäftsregeln („dieser Benutzer darf seine eigenen Rechnungen lesen, nicht die aller"). Der Web-Client spricht nie mit der Datenbank. Er fragt das Backend, höflich, über HTTP — und das Backend entscheidet, was die Datenbank tun soll, wenn überhaupt etwas.



Drei Schichten, eine bewachte Tür — nur das Backend betritt das private Netzwerk der Datenbank

Lesen Sie das Diagramm von links nach rechts und achten Sie auf die gestrichelte Grenze rechts. Postgres und das Backend teilen sich ein privates Netzwerk, das der Client nicht betreten kann; die einzige Reichweite des Clients ist der HTTP-Pfeil ins Backend. Es gibt keine Linie vom Web-Client zur Datenbank — nicht, weil wir höflich gebeten haben, keine zu zeichnen, sondern weil Docker, wie Sie gleich sehen werden, diese Linie physisch unmöglich macht.

> **Die Ein-Satz-Version**
> Die Datenbank ist ein Raum mit einer einzigen Tür, und das Backend ist der Einzige mit dem Schlüssel. Alle anderen klopfen beim Backend an und fragen höflich.

Die Abbildung auf Docker Compose

Hier hört Docker auf, eine Abstraktion zu sein, und übernimmt die Durchsetzung für uns. Wir drücken das exakte Drei-Schichten-Bild von oben als drei Services und zwei Netzwerke in einer `compose.yaml` aus. Das ist der „Zeigen statt behaupten"-Moment: Die Architektur wird zu ein paar Zeilen YAML, und die Isolation ist eine *Tatsache der Laufzeitumgebung*, keine Konvention, auf deren Einhaltung wir hoffen.

Der ganze Trick beruht auf zwei Compose-Verhaltensweisen, beide direkt aus der offiziellen Compose-Networking-Dokumentation:

- **Service-Namen sind Hostnamen.** Services in einem gemeinsamen Netzwerk finden einander per Service-Name über Dockers eingebautes DNS — das Backend erreicht die Datenbank unter `postgres://db:5432`, nirgends IP-Adressen. (Das ist genau das Service-Name-Networking aus Teil 4.)
- **Kein veröffentlichter Port bedeutet kein Host-Zugriff.** Die Dokumentation ist explizit: "Networked service-to-service communication uses the `CONTAINER_PORT`. The host port is only used when accessing the service from outside the network." Ein Service *ohne* `ports:`-Mapping ist also für andere Container in seinem Netzwerk voll erreichbar — und von Ihrem Host oder der Außenwelt aus vollkommen unerreichbar.

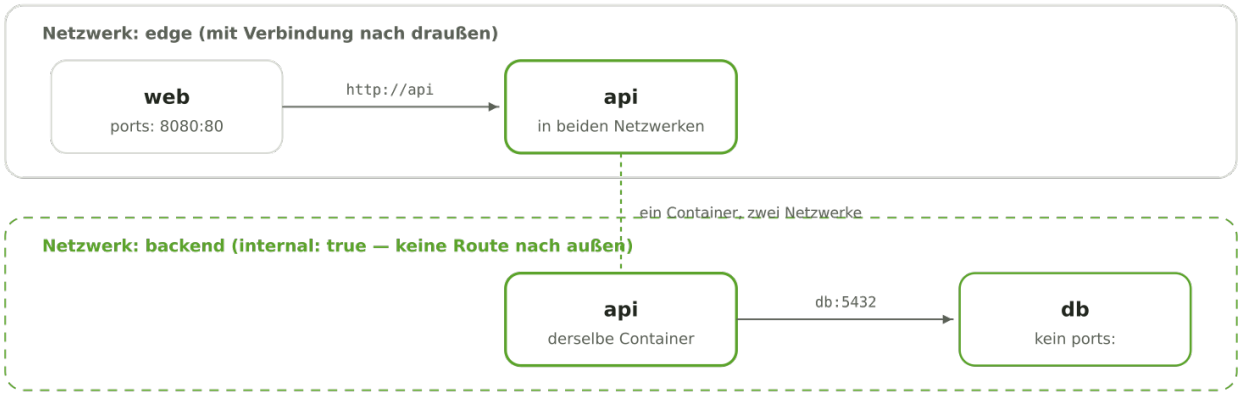
Obendrauf setzen wir noch eine Garantie mit einem **internen Netzwerk**. Laut der Compose-Networks-Referenz: "By default, Compose provides external connectivity to networks. `internal`, when set to `true`, lets you create an externally isolated network." Ein internes Netzwerk hat kein Gateway nach draußen — nichts darin kann hinausgreifen, und nichts von außen kann hinein. Es ist das digitale Äquivalent eines Raums ohne Außenfenster.

Hier ist der Stack. Lesen Sie ihn einmal von oben nach unten; die Anmerkungen nach jedem Service erklären die tragenden Schlüssel.

```
services:
  web:
    # Der öffentliche Einstiegspunkt. In vielen Stacks liefert dieser Service
    # einfach die Browser-App aus (statische Dateien oder ein Next.js/SPA-
    # Server). Der Browser ist die eigentliche "Client"-Schicht; dieser
    # Service reicht ihm nur die Seite.
    image: my-web:latest
    ports:
      - "8080:80"          # Host 8080 -> Container 80: von außen erreichbar
    networks:
      - edge               # teilt ein Netzwerk mit der api, und nur mit ihr

  api:
    # Das Backend — die eine bewachte Tür. Es ist der EINZIGE Service, der
    # auf BEIDEN Netzwerken sitzt, und damit das Einzige, was mit db reden kann.
    image: my-api:latest
    environment:
      DATABASE_URL: "postgres://appuser:${DB_PASSWORD}@db:5432/appdb"
    networks:
      - edge               # damit die Web-Schicht es erreichen kann
      - backend            # damit es (und nur es) die Datenbank erreicht
    depends_on:
```

```
      - db

  db:
    image: postgres:17
    environment:
      POSTGRES_USER: appuser
      POSTGRES_DB: appdb
      POSTGRES_PASSWORD: ${DB_PASSWORD}
    # HINWEIS: kein `ports:` hier. Nichts auf dem Host erreicht 5432.
    volumes:
      - dbdata:/var/lib/postgresql/data
    networks:
      - backend            # NUR im internen Netzwerk

networks:
  edge:                    # web <-> api; hat normale Verbindung nach draußen
    driver: bridge
  backend:
    driver: bridge
    internal: true         # <-- der private Raum: keine Route zur/von der Welt

volumes:
  dbdata:
```

Gehen Sie die Netzwerktopologie durch, und die Sicherheitseigenschaft fällt einfach heraus. Der Service `db` liegt nur im Netzwerk `backend`, und `backend` ist `internal: true` — die Datenbank hat also überhaupt keine Route zum oder vom Host oder Internet. Sie hat keine Ports veröffentlicht, sodass der Host `5432` selbst dann nicht erreichen könnte, wenn `backend` nicht intern wäre. Der Service `api` ist das einzige Mitglied *beider* Netzwerke `edge` und `backend`, und genau das macht ihn zur einzigen Tür: Er ist der eine Service, der die `web`-Schicht auf `edge` hören und mit `db` auf `backend` sprechen kann. Und der Service `web` lebt nur auf `edge`, hat also keinerlei Pfad zu `db` — versuchen Sie, von `web` aus `db:5432` zu öffnen, und Docker weigert sich schlicht, das zu routen.



Dieselben drei Schichten als Compose-Netzwerke — die api ist der einzige Service auf beiden

Das Diagramm zeigt dieselbe `api`-Box über beide Bänder gespannt — das ist die ganze Architektur auf einen Blick. Weil `api` das einzige Mitglied des `backend`-Bandes ist und `web` gar nicht darauf liegt, gibt es keine zeichenbare Linie von `web` zu `db`. Docker erzwingt die Schichtung; Sie müssen niemandem vertrauen, dass er sie respektiert.

> **Beweisen Sie es sich selbst in zehn Sekunden**
>
> Fahren Sie den Stack hoch und versuchen Sie, die Datenbank von Ihrem Host aus zu erreichen. `psql -h localhost -p 5432` — die Verbindung wird abgelehnt, weil auf 5432 nichts veröffentlicht ist. Nun `docker compose exec api sh` und von innerhalb des Backends verbinden: Es funktioniert, weil der `api`-Container

> im `backend`-Netzwerk liegt. Die Ablehnung mit eigenen Augen zu sehen, macht aus „Isolation" ein Wort, dem Sie als Tatsache vertrauen.

Das Backend, in beiden Stacks

Nun zur Box in der Mitte. Ich habe zwei Standard-Backends für genau diese Architektur, und ich möchte vorab klarstellen: **Keines ist das „richtige".** Sie sind die richtigen für *unterschiedliche Teams*, und beide passen in den `api`-Slot oben, ohne eine einzige Zeile der Compose-Datei zu ändern. Die Architektur ist sprachagnostisch; das ist der ganze Punkt.

Ich zeige Ihnen denselben Endpunkt — „gib mir die Kundenliste, gelesen aus Postgres" — in beiden.

TMS XData (Delphi)

Wenn Ihr Team in Delphi zu Hause ist, hält TMS XData — ein REST/JSON-Server-Framework auf Basis des HTTP-Frameworks TMS Sparkle — Ihr Backend in der Sprache und den Werkzeugen, die Sie bereits kennen. Seine zentrale Idee, die ich im Finale der Networking-Serie behandelt habe, ist: Sie deklarieren einen **Service-Vertrag als gewöhnliches Delphi-Interface**, und das Framework macht daraus HTTP-Endpunkte.

Laut der offiziellen Dokumentation zu Service-Operationen sieht der Vertrag so aus:

```
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));
```

Eine Zeile pro Idee: `[ServiceContract]` markiert das Interface als Service, `IInvokable` plus die GUID gibt dem Framework die nötige RTTI, `[HttpGet]` bindet `List` an einen GET-Request (standardmäßig antworten XData-Operationen auf POST, dieses Attribut ist also unser REST-Verb-Regler), und `RegisterServiceType` macht das Interface bekannt. Die Implementierung ist eine gewöhnliche Klasse, die aus Postgres liest und Objekte zurückgibt — XData serialisiert `TList<T>` automatisch nach JSON:

```
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;
  // Hier aus Postgres lesen und Result befüllen.
  // Die DB-Verbindung nutzt die DATABASE_URL / den Host `db` aus Compose –
  // die Zugangsdaten leben in diesem Backend, nirgendwo sonst.
end;

initialization
  RegisterServiceType(TCustomerService);
```

Der Datenbankzugriff selbst ist FireDAC, genau wie in Teil 2 eingerichtet — dieselbe Verbindung, jetzt auf den Host `db` statt `localhost` zeigend, da Compose uns diesen Namen schenkt. Ein ehrlicher Vorbehalt gehört ausgesprochen: Wenn Sie ein Delphi-Backend unter Linux containerisieren, braucht FireDACs Postgres-Treiber die Postgres-Client-Bibliothek (`libpq`) im Image — genau das Treiberdetail aus Teil 2. Das ist ein Verpackungsschritt, keine Neuarchitektur — und TMS dokumentiert den Betrieb von XData/Sparkle-Servern unter Linux direkt, sodass sich der eigenständige HTTP-Server sauber containerisieren lässt, sobald der Treiber im Image steckt.

> **Wo der XData-Server läuft**
> Ein XData-Server ist ein eigenständiger HTTP-Server — er besitzt seinen Port und beantwortet Anfragen selbst, ganz ohne externen Webserver — und genau deshalb passt er in den `api`-Slot unseres Compose-Stacks. Welchen Sparkle-Host Sie konkret wählen (und die Linux-Verpackung mit `libpq`) ist ein Deployment-Detail; die *Architektur* — ein Backend, dem die Datenbank gehört — bleibt unverändert.

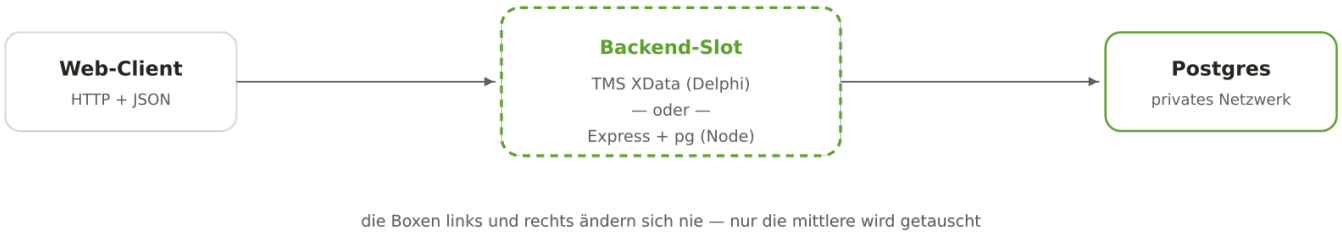
Express + TypeScript (Node)

Wenn Ihr Team ein JavaScript/TypeScript-Haus ist oder Sie das riesige npm-Ökosystem im Rücken haben wollen, liefert Express — das minimale, enorm populäre Node.js-Web-Framework — den identischen Endpunkt mit sehr wenig Zeremonie und spricht mit Postgres über node-postgres (das Paket `pg`, der Standard-Postgres-Client für Node). Laut der node-postgres-Dokumentation ist eine gepoolte Abfrage `pool.query(...)` mit `.rows` als Ergebnis:

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

// Der Pool liest DATABASE_URL aus der Umgebung — derselbe Connection-String,
// den Compose injiziert hat. Zugangsdaten leben hier, im Backend, nur hier.
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); // der JSON-Vertrag, den der Web-Client konsumiert
});

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

Dieselbe Form wie die Delphi-Version: ein GET-Endpunkt, ein Lesen aus Postgres über das private Netzwerk, eine JSON-Antwort. Der `Pool` verbindet sich über `DATABASE_URL` mit dem Host `db` — und genau wie beim XData-Backend ist dies der einzige Ort, an dem das Datenbank-Passwort je existiert.

Und hier der Punkt, der mir am wichtigsten ist: Diese beiden Backends sind *innerhalb derselben Architektur austauschbar.*



Gleiche Architektur, austauschbares Backend — XData oder Express passt in einen Slot

Die gestrichelte Box ist der Punkt: Web-Client und Datenbank wissen nicht und kümmern sich nicht darum, welche Sprache die Mitte füllt. Beide Backends containerisieren identisch, beide hängen an denselben zwei Netzwerken, beide halten die Zugangsdaten, beide beantworten dasselbe HTTP. Wählen Sie also nach *Passung*, nicht nach Sieger:

- **Delphi-Haus, bestehende FireDAC-/Geschäftslogik, kommerzieller Support gewünscht?** Greifen Sie zu XData — Sie bleiben in einer Sprache, Ihre Verträge sind RTTI-getriebene Delphi-Interfaces, und Ihr bestehender Datenzugriffscode aus Teil 2 kommt einfach mit.
- **JS/TS-Team, oder Sie wollen das npm-Ökosystem und einen großen Bewerberpool?** Greifen Sie zu Express — minimal, allgegenwärtig, und `pg` ist ein grundsolider Postgres-Client.

Beide sind exzellent in dem, was sie tun, und beide lassen die Architektur — das, was Ihre Daten tatsächlich schützt — exakt gleich.

> **Der weitere Horizont**

> Fairness verlangt, die Nachbarn zu nennen, denn dieses Muster ist universell. In Delphi baut das quelloffene DelphiMVCFramework dieselben REST/JSON-Backends und ist eine wirklich starke Wahl, wenn eine kommerzielle Lizenz nicht passt. Darüber hinaus füllen Teams denselben Backend-Slot täglich mit Fastify (Node), FastAPI (Python) und ASP.NET Core (C#). Jedes davon isoliert eine Datenbank hinter einem internen Netzwerk in Docker genau so wie wir oben — die Architektur ist das, was unverändert übertragbar bleibt.

Was wir aufschieben — mit Absicht

Ich habe diesen Stack bewusst minimal gehalten, damit die *Form* sichtbar bleibt, und Ehrlichkeit verlangt zu benennen, was ein echtes Produktions-Deployment obendrauf setzt. Nichts davon ändert die Architektur, die Sie gerade gebaut haben; jedes ist eine Schicht, die Sie darum herumlegen.

> **Die Produktion ergänzt dies — benannt, verlinkt und für einen anderen Tag aufgehoben**
> - **TLS/HTTPS am Rand.** Alles hier spricht reines HTTP, was innerhalb des privaten Netzwerks in Ordnung ist, an der öffentlichen Grenze aber verschlüsselt werden muss. Siehe HTTPS/TLS. - **Echtes Secrets-Management.** Wir haben der Klarheit halber eine Klartext-Umgebungsvariable `${DB_PASSWORD}` benutzt, aber die Compose-Best-Practices-Dokumentation ist unmissverständlich: "Don't use environment variables to pass sensitive information… Use secrets instead." Bevorzugen Sie Docker Secrets, die ein Secret als schreibgeschützte Datei unter `/run/secrets/…` einbinden, statt es in der Umgebung offenzulegen, und halten Sie Konfiguration in einer `.env`-Datei außerhalb der Versionsverwaltung. - **Authentifizierung.** Das Backend ist der Ort, sie durchzusetzen — ein API-Key oder Token im `Authorization`-Header —, damit die „bewachte Tür" tatsächlich Ausweise prüft. - **Ein Reverse-Proxy an der Front.** Werkzeuge wie nginx oder Traefik sitzen typischerweise am Rand, um TLS zu terminieren, zu routen und Last über Backend-Instanzen zu verteilen. Jedes davon verdient eine eigene Behandlung; keines ändert die Drei-Schichten-Isolation, die wir heute gebaut haben.

Die ganze Docker-Serie, in einem Bogen

Treten Sie ganz zurück. Fünf Beiträge, und jeder hat das Fundament gegossen, auf dem der nächste stand — vom einzelnen Container bis zur sauber isolierten Architektur.

Der Serienbogen: vom ersten Container zu einer Datenbank, die nur Ihr Backend berühren kann

Lesen Sie es von links nach rechts, und der Gewinn ist offensichtlich: Nichts rechts hat irgendetwas links weggeworfen. Die isolierte Architektur aus Teil 5 *ist* der Compose-Stack aus Teil 4, in der Netzwerktopologie

eines erwachsenen Systems, mit Images aus Teil 3, dem Postgres aus Teil 2, gestartet mit den Befehlen aus Teil 1.

Und hier die Verbindung, auf die ich am liebsten hinweise. Wenn sich irgendetwas davon lesbar anfühlte — Ports, die gemappt werden, Namen, die aufgelöst werden, HTTP-Requests, die JSON transportieren —, ist das kein Zufall. Es ist exakt das Vokabular aus dem Finale von Networking für Delphi-Entwickler. Ein veröffentlichter

Port ist das `IP:Port` aus Teil 1. Service-Name-Networking ist das DNS aus Teil 2. Der Endpunkt des Backends ist ein Web-Service aus Teil 5. Docker hat keine neuen Konzepte erfunden; es hat die alten verpackt. Sie sind durch die Networking-Tür gegangen, und sie öffnete sich direkt hierhin.

Fazit

Die Datenbank ist das Kronjuwel, also bekommt sie genau eine Tür — das Backend —, und Docker macht diese Tür zur *einzigen*, die überhaupt existiert. Legen Sie die Datenbank in ein Netzwerk mit `internal: true` ohne

veröffentlichte Ports, legen Sie das Backend sowohl in dieses private als auch ins öffentliche Netzwerk, und der Web-Client kann die Datenbank physisch nicht erreichen. Allein das Backend hält die Zugangsdaten, erzwingt Auth und Validierung und spricht die Geschäftsregeln. Füllen Sie den Backend-Slot mit TMS XData, wenn Sie ein Delphi-Haus sind, oder mit Express, wenn Sie ein TypeScript-Haus sind — die Wahl ist eine Frage der Passung, nicht des Verdienstes, und der Architektur ist es egal, wofür Sie sich entscheiden.

> Ein sicheres System ist nicht eines, bei dem die Datenbank an der Tür gut verteidigt wird. Es ist eines, bei dem die Datenbank gar keine öffentliche Tür hat — und das Einzige, was anklopfen kann, ist Code, den Sie geschrieben haben und dem Sie vertrauen.

Das war die Serie. Sie können jetzt Docker installieren, eine Datenbank betreiben, Images bauen und taggen, einen Multi-Service-Stack komponieren und ihn so auslegen, dass die wertvollste Schicht wirklich nur über Ihr eigenes Backend erreichbar ist. Das ist kein Spielzeug — das ist die echte Form von Produktion.

> **Diese Serie: Docker für Delphi-Entwickler**
> 1. Docker installieren und den ersten Container starten 2. Postgres im Container, FireDAC außen dran 3. Images, Tags und Repositories 4. Docker Compose: Multi-Service-Stacks 5. **Die Isolationsarchitektur — dieser Beitrag**

Wenn Sie diese Schicht für ein Delphi-Produkt entwerfen — entscheiden, wo das Backend lebt, wie die Datenbank privat bleibt, ob XData oder ein Node-Service zu Ihrem Team passt — und ein zweites Paar Augen von jemandem möchten, der genau diesen Weg mit Legacy-Codebasen gegangen ist: Sprechen wir. Ihre

Anwendung funktioniert bereits. Geben wir ihr eine Architektur, die ihrer würdig ist.

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 —