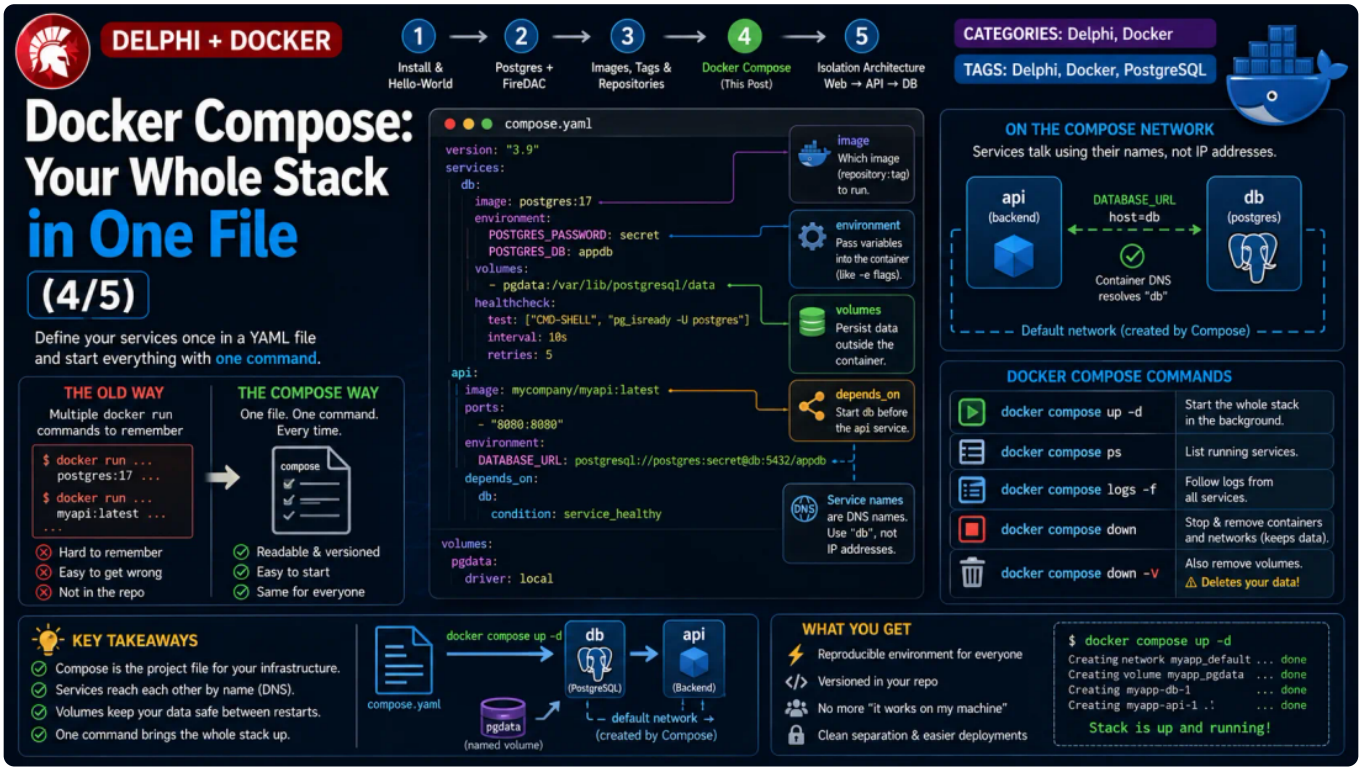
NEWS · AUGUST 28, 2026

Docker Compose: Ihr gesamter Stack in einer Datei (4/5)

By Dr. Holger Flick



Teil 3 hat Sie fit gemacht in Images, Tags und Repositories — Sie können einen bestimmten `postgres:17`-Build pullen, eigene Images taggen und sie dorthin pushen, wo Ihr Team sie wieder abholen kann. Sie können jeden einzelnen Container souverän starten. Und genau dort beginnt sich eine leise Frustration einzuschleichen.

Denn eine echte Anwendung ist nie *ein* Container. Sie ist eine Datenbank **und** ein Backend. Vielleicht auch noch ein Cache. Jeden davon von Hand zu starten bedeutet, pro Service einen ganzen Absatz an `docker run`-Flags auswendig zu kennen — das richtige Image, das Port-Mapping, vier `-e`-Umgebungsvariablen, ein `--network`, ein Volume-Mount — und sie in der richtigen Reihenfolge auszuführen, jedes Mal, auf jeder Maschine. Fehlt ein Flag, findet das Backend die Datenbank nicht. Das skaliert nicht über eine Kaffeepause hinaus.

Es gibt ein Werkzeug, das genau diese Klasse von Problemen aus der Welt schafft, und es steckt bereits in dem Docker, das Sie in Teil 1 installiert haben. Es heißt **Docker Compose**, und am Ende dieses Beitrags beschreiben Sie Ihren gesamten Stack — eine Postgres-Datenbank und ein Backend, das mit ihr spricht — in einer einzigen, gut lesbaren Datei und starten alles mit einem einzigen Befehl.

Dies ist Teil 4 einer fünfteiligen Reise. Hier ist die vollständige Karte, damit Sie immer wissen, wo Sie stehen.

> **Diese Serie: Docker für Delphi-Entwickler**

1. Docker installieren und den ersten Container starten — hello-world, der Daemon, die Grundlagen 2. Ein Postgres-Container und FireDAC von der Kommandozeile — eine echte Datenbank in Sekunden 3. Images, Tags und Repositories — woher Images kommen und wie Versionen funktionieren 4. **Docker Compose: Ihr gesamter Stack in einer Datei — dieser Beitrag** — eine Datenbank und ein Backend, gemeinsam 5. Die Isolationsarchitektur — Web ’ Backend ’ Datenbank, und warum nur das Backend die Daten anfasst

Das Problem, beim Namen genannt

Lassen Sie mich die Frustration ehrlich auf den Tisch legen, denn sie zu benennen ist schon das halbe Argument für Compose. So ungefähr sieht es aus, eine Postgres-Datenbank und ein Backend, das sich mit ihr verbindet, von Hand zu starten — auf die Art aus Teil 1–3.

```
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

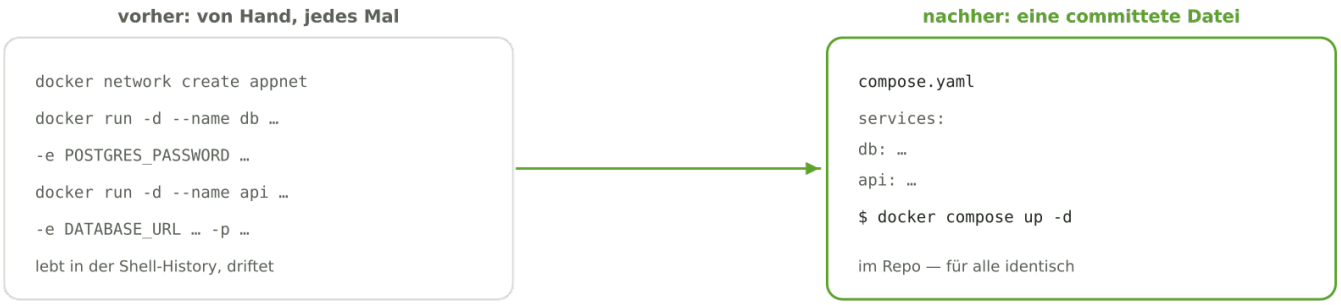
docker run -d --name api --network appnet \
  -e DATABASE_URL=postgres://postgres:secret@db:5432/appdb \
  -p 8080:8080 \
  my-backend:1.0
```

Jede einzelne dieser Zeilen ist korrekt — aber sie lebt in Ihrer Shell-History, nicht in Ihrem Projekt. Ein Teamkollege, der das Repository klont, hat keine Ahnung, dass diese Befehle existieren, geschweige denn, wie ihre Flags genau lauten. Ändern Sie das Datenbankpasswort, müssen Sie zwei Befehle im Gleichschritt anpassen. Das ist Infrastruktur per mündlicher Überlieferung — und mündliche Überlieferung driftet.

Was Compose eigentlich ist

Docker Compose ist, in Dockers eigenen Worten, ein Werkzeug zum Definieren und Ausführen von Multi-Container-Anwendungen — Sie schreiben den gesamten Stack als eine deklarative YAML-Datei nieder (YAML ist ein Klartextformat für strukturierte Konfiguration, ganz aus Einrückung und `key: value`-Paaren aufgebaut) und starten ihn mit einem einzigen Befehl. Alles aus der `docker run`-Suppe oben wird zu ein paar Zeilen lesbarem Text, der *in Ihrem Repository* lebt, direkt neben Ihrem Code.

Hier ist das mentale Modell, mit dem es für einen Delphi-Entwickler klick macht: **Compose ist die Projektdatei für Ihre Infrastruktur.** Sie bauen Ihre `.dproj` auch nicht jeden Morgen neu auf, indem Sie sich an Compiler-Schalter erinnern; Sie öffnen das Projekt, und alles ist deklariert. Compose macht dasselbe für Ihre Services. Deklarieren Sie den Stack einmal, committen Sie ihn, und jeder Teamkollege — und jeder Server — führt mit einem Befehl exakt dasselbe aus.



Aus docker-run-Suppe wird eine deklarierte, committete Datei

Der rechte Kasten ist das gesamte Ziel dieses Beitrags. Beachten Sie: Er kann nicht *weniger* als die linke Seite — er tut alles, was die Befehle taten, einschließlich des Anlegens von Netzwerk und Volume für Sie. Es steht nur dort geschrieben, wo es hingehört.

> **Compose ist heute in Docker eingebaut**
> Sie installieren nichts Zusätzliches. Laut den offiziellen Installationsdocs gilt: *"Docker Desktop includes Docker Compose along with Docker Engine and Docker CLI"* — Compose kommt als modernes `docker compose`-Kommando (ein v2-CLI-Plugin; beachten Sie das Leerzeichen, nicht das ältere `docker-compose` mit Bindestrich). Wenn Sie Teil 1 abgeschlossen haben, haben Sie es bereits. Prüfen Sie es mit `docker compose version`.

`compose.yaml` aufbauen, Service für Service

Der beste Weg, die Datei zu lernen, ist, sie wachsen zu lassen. Wir beginnen mit der Datenbank allein, starten sie und fügen dann das Backend hinzu — wobei jeder neue YAML-Schlüssel bei seinem ersten Auftreten erklärt wird (das ist das Versprechen dieser Serie: keine unerklärte Magie). Die Datei heißt per Konvention `compose.yaml` und liegt im Wurzelverzeichnis Ihres Projekts.

Schritt 1 — Den Datenbank-Service deklarieren

Jede Compose-Datei hat einen `services:`-Block auf oberster Ebene; jeder Schlüssel darunter ist ein Service (im Wesentlichen ein Container) mit einem Namen Ihrer Wahl. Hier ist ein `db`-Service, der dasselbe Postgres-Image wie in Teil 2 ausführt.

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

Drei Schlüssel, jeder mit genau einer Aufgabe. `image:` benennt das exakte Image, das laufen soll — `postgres:17`, genau die `repository:tag`-Form, die Sie in Teil 3 gelernt haben. `environment:` reicht Umgebungsvariablen in den Container hinein, exakt wie die `-e`-Flags aus Teil 2 — hier liest das offizielle Postgres-Image `POSTGRES_PASSWORD`, um das Superuser-Passwort zu setzen, und `POSTGRES_DB`, um beim ersten Start eine Datenbank namens `appdb` anzulegen. Das ist bereits eine vollständig lauffähige Compose-Datei.

Schritt 2 — Den Backend-Service hinzufügen

Nun fügen wir einen zweiten Service hinzu, `api`, neben `db` im selben `services:`-Block. Das ist das Backend, das Ihre Delphi-Anwendung (und ein Web-Frontend) aufrufen wird — in Teil 5 bauen wir es richtig, betrachten Sie den Image-Namen hier also als Platzhalter für „unser Backend".

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

  api:
    image: my-backend:1.0        # Platzhalter — Teil 5 baut das richtig
    ports:
      - "8080:8080"
    environment:
      DATABASE_URL: postgres://postgres:secret@db:5432/appdb
    depends_on:
      - db
```

Drei neue Schlüssel verdienen hier ihre Vorstellung. `ports:` veröffentlicht einen Container-Port auf Ihrem Host in der vertrauten `"host:container"`-Form aus Teil 2 — `"8080:8080"` bedeutet: „Anfragen an `localhost:8080` auf meiner Maschine erreichen Port 8080 im `api`-Container." `depends_on:` weist Compose an, `db` vor `api` zu starten — die Reihenfolge, die Sie bisher von Hand verwaltet haben, ist nun deklariert. Und schauen Sie sich diese `DATABASE_URL` genau an — der Host ist buchstäblich `db`. Dieses eine Wort ist der Lohn dieses ganzen Beitrags, und es verdient einen eigenen Abschnitt.

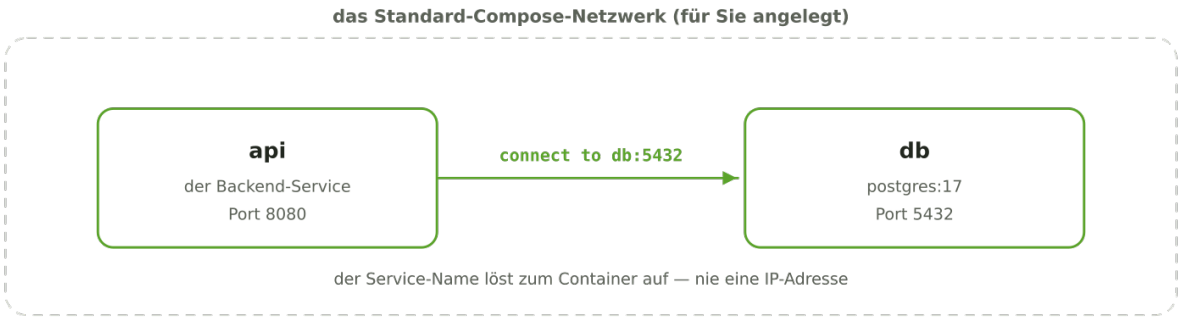
> **image: zum Ausführen, build: zum Bauen**
>
> Unser `api` verwendet hier `image:`, um ein fertig gebautes Image auszuführen. Wenn Compose stattdessen ein Image aus einem lokalen `Dockerfile` bauen soll, tauschen Sie `image:` gegen `build: .` — Compose baut es beim `up` für Sie. In Teil 5, wenn das Backend real wird, stützen wir uns auf `build:`; für den Moment hält `image:` den Fokus auf dem Stack statt auf dem Build.

Der Lohn: Services finden sich gegenseitig über ihren Namen

Hier ist die Frustration, die Compose stillschweigend beseitigt. Von Hand mussten Sie, damit das Backend die Datenbank erreicht, die IP-Adresse des Datenbank-Containers kennen — eine Adresse, die Docker dynamisch vergibt und die sich bei jeder Neuerstellung des Containers ändert. In einem Compose-Netzwerk fassen Sie nie wieder eine IP an.

Compose legt automatisch jeden Service Ihrer Datei in ein gemeinsames Netzwerk, und — das ist die Magie — **jeder Service ist unter seinem Service-Namen als Hostname erreichbar.** Unser `api` verbindet sich mit der Datenbank unter dem Host `db`, weil das der Name des Service im YAML ist. Keine IP. Der Name.

Das ist keine Erfindung von Compose; es ist Container-DNS. Die Docs sagen es klar: *"Each service registers its name with an internal DNS server, so containers can reach each other using the service name directly"* — und, entscheidend: *"you should always reference services by name, not IP address"*, weil der Name konstant bleibt, selbst wenn ein Container mit frischer IP neu erstellt wird.



Im Compose-Netzwerk erreicht api die Datenbank über den Namen 'db' — nirgendwo eine IP-Adresse

Wenn Sie der Netzwerk-Serie gefolgt sind, ist das überhaupt keine neue Idee — es ist exakt die DNS-Lektion aus Teil 2 jener Serie, wo ein *Name* zu einer *Adresse* aufgelöst wird, damit Sie die Nummer nie hartkodieren.

Docker gibt jedem seiner Container-Netzwerke einen eingebauten DNS-Server, der genau das für Service-Namen erledigt. Den Mechanismus haben Sie bereits verstanden; Compose wendet ihn nur innerhalb Ihres Stacks an.

> **Warum das für Ihr Delphi-Backend wichtig ist**
> Wenn sich das Backend in Teil 5 mit Postgres verbindet, zeigt sein Connection-String auf den Host `db` — einen Namen, der stabil, lesbar und auf jedem Entwickler-Laptop wie auf dem Produktionsserver identisch ist. Niemand jagt je wieder einer geänderten IP-Adresse hinterher. Das ist die Zuverlässigkeitsdividende von Container-DNS.

Volumes: die Persistenz-Antwort, die wir in Teil 2 versprochen haben

Schon in Teil 2 hing eine bohrende Frage im Raum: Wenn Sie den Postgres-Container stoppen und entfernen — wo gehen die Daten hin? Die ehrliche Antwort lautete *„sie sind weg"* — das eigene Dateisystem eines Containers ist flüchtig und wird beim Entfernen des Containers gelöscht. Mit Compose können wir das sauber und dauerhaft beheben, und jetzt ist der richtige Moment dafür.

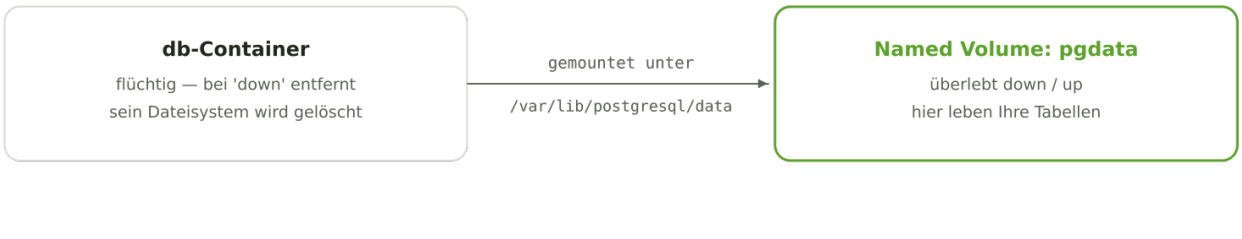
Die Lösung ist ein **Volume**: ein Stück Speicher, das *außerhalb* des Lebenszyklus eines einzelnen Containers lebt und daher überlebt, wenn der Container zerstört und neu erstellt wird. Sie mounten es genau in das Verzeichnis, in dem Postgres seine Daten hält, und die Daten bleiben über Neustarts hinweg erhalten. Hier ist unsere Datei mit einem hinzugefügten Named Volume.

```
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:
```

Zwei Ergänzungen leisten die Arbeit. Die `volumes:`-Zeile auf Service-Ebene mountet ein Volume namens `pgdata` unter `/var/lib/postgresql/data` — und dieser Pfad ist nicht willkürlich. Das offizielle Postgres-Image ist eindeutig: *"Mount the data volume at `/var/lib/postgresql/data`"* für Postgres 17 und früher, denn das ist das Verzeichnis, in das die Datenbank tatsächlich schreibt. Der separate `volumes:`-Block auf oberster Ebene ganz unten **deklariert** `pgdata` als Named Volume, das Docker für Sie verwaltet. Stoppen Sie nun den Stack, starten Sie ihn neu — und Ihre Tabellen sind noch da.



Der Container ist wegwerfbar; das Named Volume nicht — die Daten überleben den Container

Die Kernaussage dieses Bildes: Der Container und seine Daten sind jetzt *zwei getrennte Dinge*. Werfen Sie den Container weg und bauen Sie ihn auf einem neueren Postgres-Tag neu — das Volume und jede Zeile darin bleiben erhalten. Lassen Sie das Volume aus der Datei weg, fällt Postgres auf ein anonymes Volume zurück, das nicht wiederverwendet wird — das ist das flüchtige Verhalten, das wir in Teil 2 gesehen haben.

> **Named Volume vs. Bind Mount**
> Es gibt zwei Wege, Speicher zu mounten. Ein **Named Volume** (hier `pgdata:`) ist Speicher, den Docker besitzt und für Sie verwaltet — ideal für Datenbankdaten, portabel und der empfohlene Standard. Ein **Bind Mount** bildet einen bestimmten Ordner *Ihres Hosts* in den Container ab (`./local/path:/container/path`) — perfekt, um während der Entwicklung Quellcode live zu bearbeiten, weniger geeignet für Datenbank-

> dateien. Für Postgres-Persistenz greifen Sie zum Named Volume; beide sind in Dockers Storage-Leitfaden dokumentiert.

Der Lebenszyklus: fünf Befehle, die Sie wirklich verwenden werden

Mit fertig geschriebener `compose.yaml` ist der Betrieb des gesamten Stacks eine Handvoll Befehle, alle unter `docker compose`. Hier sind sie in der Reihenfolge, in der sie Ihnen im Alltag begegnen.

Schritt 1 — Alles im Hintergrund starten

Führen Sie `up` mit `-d` (detached) aus, um das Netzwerk aufzubauen, das Volume anzulegen und jeden Service zu starten — Ihr Prompt kommt sofort zurück.

```
docker compose up -d
```

Compose liest die `compose.yaml` im aktuellen Verzeichnis, erstellt das gemeinsame Netzwerk und das `pgdata`-Volume und startet dann `db` und `api` — unter Beachtung von `depends_on`, sodass die Datenbank zuerst hochkommt.

Schritt 2 — Sehen, was läuft

Prüfen Sie den Status der Services dieses Stacks mit `ps`.

```
docker compose ps
```

Das listet nur die Container *Ihres* Stacks und ihren Zustand — eine fokussierte Ansicht, nicht jeden Container auf der Maschine.

Schritt 3 — Die Logs lesen

Folgen Sie der kombinierten Ausgabe aller Services mit `logs -f` — hier beobachten Sie, wie die Datenbank ihre erste Verbindung annimmt.

```
docker compose logs -f
```

Die Ausgaben beider Services sind verschachtelt und nach Service-Namen gekennzeichnet; lassen Sie das `-f` weg, um statt eines Live-Follows eine einmalige Ausgabe zu bekommen.

Schritt 4 — Stoppen und aufräumen (Daten bleiben erhalten)

Fahren Sie den Stack mit `down` herunter, das — laut der CLI-Referenz — *"Stops and removes containers, networks."*

```
docker compose down
```

Das entfernt die Container und das Netzwerk, **lässt Ihr Named Volume aber unangetastet** — ein erneutes `docker compose up -d` bringt alles mit intakten Daten zurück. Das ist das alltägliche Stoppen.

Schritt 5 — Der destruktive Befehl (löscht Ihre Daten)

Es gibt ein Flag, das alles ändert, und es verdient eine klare Warnung.

```
docker compose down -v
```

> **down -v löscht Ihr Volume — und Ihre Daten**
>
> Das Flag `-v` (`--volumes`) weist `down` an, auch die in Ihrem `volumes:`-Block deklarierten Named Volumes zu entfernen. Das bedeutet: `pgdata` wird gelöscht, und **jede Zeile in Ihrer Datenbank ist weg.** Verwenden
>
> Sie es bewusst — um absichtlich auf einen sauberen Stand zurückzusetzen — niemals aus Gewohnheit. Das schlichte `docker compose down` (ohne `-v`) ist der sichere Alltagsbefehl; `down -v` ist der Befehl für „komplett von vorn anfangen".

Wo Compose hinpasst — und wo Teams zu mehr greifen

Ein ehrliches Wort zum Geltungsbereich, denn Compose ist ausgezeichnet, aber nicht die Antwort auf jede Frage. Compose ist hervorragend für die lokale Entwicklung und für den Betrieb eines Stacks auf einem einzelnen Host — was einen enormen Anteil echter, wertvoller Arbeit abdeckt, einschließlich allem in dieser Serie.

Wenn Sie über *viele* Hosts hinweg laufen müssen, mit automatischem Failover, Rolling Updates und Self-Heal-

ing — Produktionsorchestrierung im großen Maßstab —, greifen Teams zu anderen Werkzeugen, und die sollte man beim Namen kennen. Kubernetes ist der Industriestandard für große Multi-Host-Orchestrierung;

Docker Swarm bietet Clustering mit einem Compose-ähnlichen Gefühl; und Podman mit `podman-compose` ist

ein starker daemonloser Open-Source-Weg, der dieselben Compose-Dateien liest. Jedes ist für seine Aufgabe eine großartige Wahl. Um Ihren Stack zu lernen und lokal zu betreiben — das Ziel dieser Serie — ist Compose genau das richtige Werkzeug, und die Konzepte lassen sich direkt übertragen, wenn Sie wachsen.

Fazit

Die `docker run`-Suppe liegt hinter Ihnen. Ihre Datenbank und Ihr Backend leben nun in einer lesbaren `compose.yaml`, committet in Ihrem Repo, gestartet mit einem einzigen Befehl und identisch auf jeder Maschine, die sie ausführt. Unterwegs haben Sie die zwei Antworten bekommen, die sich diese Serie aufgespart hatte: Services erreichen einander über den **Service-Namen** dank Container-DNS — nie über IP-Adressen — und

ein **Named Volume** am Datenverzeichnis von Postgres lässt Ihre Daten Neustarts überleben. Fünf Befehle (`up -d`, `ps`, `logs`, `down` und das bewusste `down -v`) decken den gesamten Lebenszyklus ab.

> Compose ist die Projektdatei für Ihre Infrastruktur: Deklarieren Sie den Stack einmal, und alle — jeder Laptop, jeder Server — betreiben dieselbe Datenbank, dasselbe Backend, verbunden über Namen, mit demselben Befehl.

Wir haben jetzt eine Datenbank und ein Backend, die Seite an Seite laufen und über das Compose-Netzwerk miteinander sprechen. Aber sollten *beide* erreichbar sein? In Teil 5 beantworten wir die Architekturfrage, die

dieses Setup aufwirft: warum nur das Backend jemals die Datenbank anfassen sollte, und wie das vollständige Bild Web ’ Backend ’ Datenbank — ein Browser und Ihre Delphi-App vorn, einXData- oder Express/Type-

Script-Backend in der Mitte, Postgres sicher dahinter verwahrt — zu einem Design zusammenkommt, dem Sie vertrauen können. Bis dann.

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 —