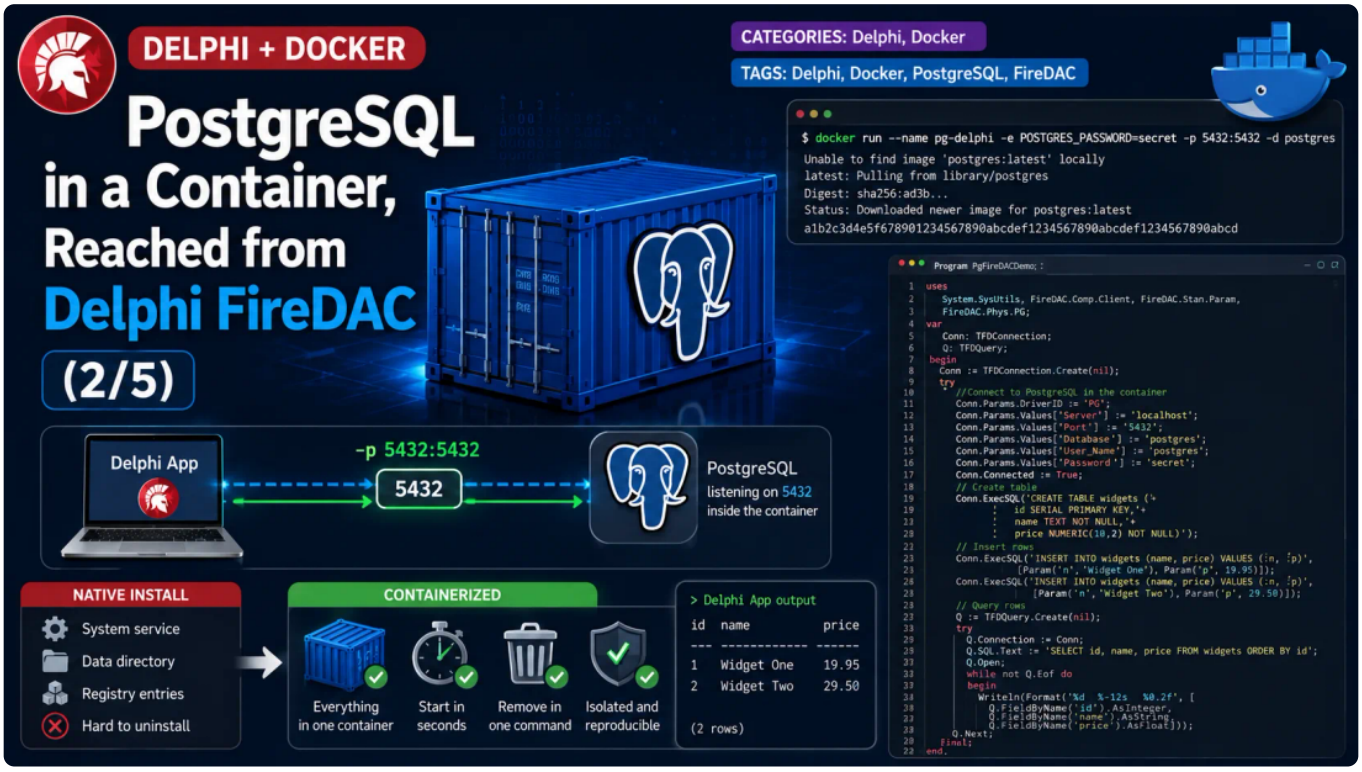
NEWS · AUGUST 24, 2026

PostgreSQL im Container, erreicht mit Delphi FireDAC (2/5)

By Dr. Holger Flick



Teil 1 hat Docker Desktop installiert, `hello-world` ausgeführt und drei Begriffe festgenagelt — **Image**, **Container**, **Registry** —, damit sie nicht länger einschüchtern. Das war das Aufwärmen. Sie können jetzt Container ausführen. Zeit, etwas laufen zu lassen, das Sie tatsächlich auf Ihrem Rechner haben wollen.

Hier ist das Versprechen dieses Beitrags: ein voll funktionsfähiger **PostgreSQL**-Datenbankserver — dieselbe relationale Datenbank, auf der ein großer Teil der Softwareindustrie aufbaut — gestartet mit **einem Befehl**, ohne nativen Installer, ohne „Weiter-Weiter-Fertigstellen"-Assistenten, ohne Dienst, der sich klammheimlich in Ihren Windows-Autostart einnistet. Und am Ende steht ein echtes Delphi-Konsolenprogramm, das mit **FireDAC** eine Tabelle anlegt, Zeilen einfügt und sie aus diesem Container wieder ausliest.

Der Clou ist, was passiert, wenn Sie mit dem Experimentieren fertig sind: `docker rm` auf den Container, und Ihr Rechner ist **blitzsauber**. Kein zurückgebliebener PostgreSQL-Dienst, kein verwaistes Datenverzeichnis, keine Registry-Einträge. Die Datenbank existierte, tat ihre Arbeit und verschwand spurlos — genau die Ordnung, die Teil 1 Ihnen von Containern versprochen hat.

Dies ist die zweite Station einer fünfteiligen Reise, hier also die ganze Landkarte.

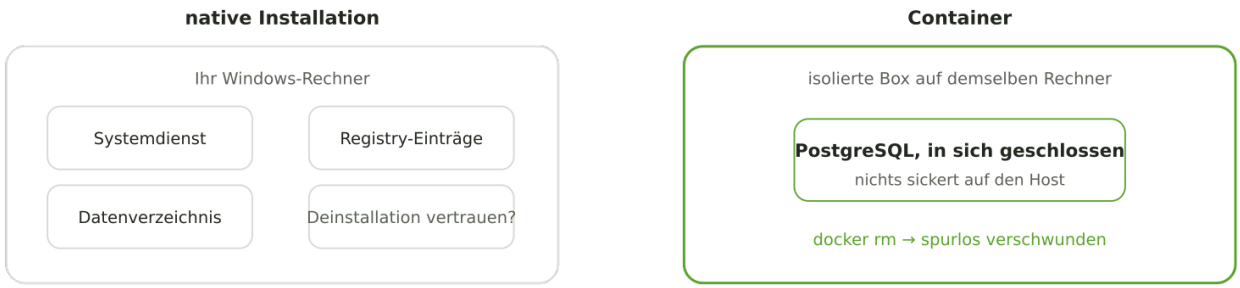
> **Diese Serie: Docker für Delphi-Entwickler**

> 1. Docker Desktop installieren und den ersten Container ausführen — das Werkzeug und die drei Begriffe, auf die es ankommt 2. **PostgreSQL im Container, erreicht mit Delphi FireDAC — dieser Beitrag** — eine echte Datenbank betreiben und aus Pascal mit ihr sprechen 3. Images, Tags, Layer und Registries — woher dieses `postgres`-Image eigentlich kam 4. Docker Compose: Datenbank und Backend zusammen — einen ganzen Stack in einer Datei beschreiben 5. Das große Ganze: Web ’ Backend ’ Datenbank, isoliert — wie die Teile in echten Deployments zusammenspielen

Warum eine Datenbank der perfekte erste echte Container ist

Eine Datenbank ist das ideale erste Objekt zum Containerisieren, und der Grund ist ebenso emotional wie technisch. PostgreSQL nativ zu installieren war schon immer eine kleine Verpflichtung — ein Systemdienst, ein Datenverzeichnis, ein Superuser-Passwort, das Sie vergessen werden, eine Deinstallation, der Sie nicht ganz trauen. Genau diese Reibung ist der Grund, warum „probier Postgres doch einfach mal einen Nachmittag lang aus" so selten passiert.

Ein Container löscht diese Verpflichtung aus. Die Datenbank läuft in einer isolierten Box, die mit Ihrem Host nichts teilt außer dem einen Port, den Sie ihr geben — und sie wegzuwerfen ist ein einziger Befehl. Ein Schema ausprobieren, es kaputtmachen, den Container löschen, neu anfangen: Die Kosten des Experimentierens sinken auf fast null.



Eine native Installation lebt auf Ihrem Rechner; ein Container ist eine Wegwerf-Box

Lesen Sie die beiden Boxen nebeneinander: Links verstreut sich eine native Installation über Ihr ganzes System; rechts lebt alles, was die Datenbank braucht, in einem Container, den Sie sauber löschen können. Dieser Kontrast ist der ganze Grund, warum wir hier beginnen.

Schritt 1 — PostgreSQL mit einem Befehl starten

Alles beginnt mit einem einzigen `docker run`. Wir verwenden das offizielle `postgres`-Image auf Docker Hub — das von der PostgreSQL-Docker-Community gepflegte Image und der kanonische Weg, PostgreSQL (die relationale Open-Source-Datenbank) in einem Container zu betreiben. Hier ist der Befehl, und danach nehmen wir jedes Flag auseinander.

```
docker run --name pg-delphi -e POSTGRES_PASSWORD=secret -p 5432:5432 -d postgres
```

Jedes Teil hat sich seinen Platz verdient, also benennen wir, was jedes einzelne tut:

- `docker run` — erzeugt und startet einen Container aus einem Image, genau wie in Teil 1.
-

`--name pg-delphi` — gibt dem Container einen sprechenden Namen, damit Sie ihn später ansprechen können (`docker stop pg-delphi`) statt über einen zufälligen Hash, den Docker sonst vergeben würde.

- `-e POSTGRES_PASSWORD=secret` — setzt eine *Umgebungsvariable* im Container. Laut der offiziellen Image-Dokumentation ist `POSTGRES_PASSWORD` die eine Variable, die das Image *verlangt*: das Passwort für den Standard-Superuser. (Zwei optionale Geschwister gibt es: `POSTGRES_USER` ändert den Superuser-Namen von seinem Standard `postgres`, und `POSTGRES_DB` setzt den Namen der initialen Datenbank, der sonst dem Benutzernamen entspricht. Wir bleiben bei den Standardwerten und halten die beweglichen Teile klein.)
- `-p 5432:5432` — veröffentlicht einen Port: Port **5432 auf Ihrem Host** wird auf Port **5432 im Container** abgebildet. `5432` ist der Standardport von PostgreSQL. Dieses Flag ist die gesamte Brücke zwischen Ihrer Delphi-Anwendung und der Datenbank und bekommt in Schritt 3 sein eigenes Diagramm.
- `-d` — läuft *detached*, also im Hintergrund, sodass Ihr Terminal sofort wieder frei ist, statt am Log des Servers zu hängen.
- `postgres` — das Image, das ausgeführt werden soll. Da es noch nicht auf Ihrem Rechner liegt, zieht Docker es bei der ersten Verwendung aus der Registry (das dritte Wort aus Teil 1).

Diese Portnummer ist Ihnen nicht neu, wenn Sie die Netzwerk-Serie verfolgt haben — es ist dieselbe „Programme lauschen auf einem nummerierten Port"-Idee aus Ports und localhost, jetzt auf einen Container gerichtet. Führen Sie den Befehl aus, und in wenigen Sekunden haben Sie einen PostgreSQL-Server, der Verbindungen auf `localhost:5432` annimmt. Kein Installer ist gelaufen. Nichts hat Ihre Windows-Dienste angerührt.

Schritt 2 — Prüfen, ob er lebt

Bevor wir eine Zeile Delphi schreiben, beweisen wir, dass der Server läuft. Der Alltagsbefehl für „Was läuft gerade?" ist `docker ps`, der Ihre laufenden Container auflistet.

```
docker ps
```

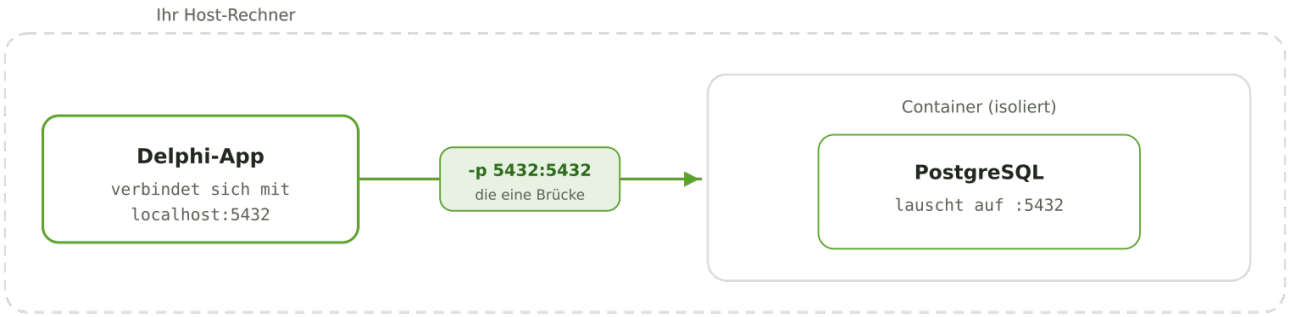
Sie sehen eine Zeile für `pg-delphi` mit dem Image `postgres`, einem Status wie `Up 12 seconds` und einer Ports-Spalte mit `0.0.0.0:5432->5432/tcp` — Docker bestätigt das Mapping, das Sie angefordert haben. Wenn Sie in die laufende Datenbank *hineinschauen* möchten, können Sie eine Shell im Container öffnen und mit PostgreSQL über dessen eigenen Kommandozeilen-Client `psql` sprechen:

```
docker exec -it pg-delphi psql -U postgres
```

Dieser Befehl verdient eine kurze Erläuterung: `docker exec` führt einen Befehl *innerhalb eines bereits laufenden Containers* aus (im Gegensatz zu `docker run`, das einen neuen startet). Die Flags `-it` machen ihn interaktiv mit angeschlossenem Terminal, und `psql -U postgres` startet den eingebauten Client von PostgreSQL als Superuser `postgres`. Sie landen an einem `postgres=#`-Prompt — das ist die Datenbank selbst, die aus dem Inneren der Box antwortet. Mit `\q` verlassen Sie sie wieder. Dieser Schritt ist völlig optional; er steht hier, damit Sie *sehen* können, dass der Server echt ist, bevor Delphi sich je verbindet.

Schritt 3 — Das mentale Modell: ein Port, in die Box gebrückt

Das mit Abstand wichtigste Konzept dieses gesamten Beitrags ist das Flag `-p 5432:5432`, also machen wir ein Bild daraus. Ein Container ist isoliert — standardmäßig kann nichts von außen ein Programm darin erreichen. Einen Port zu veröffentlichen bohrt genau ein vorhersagbares Loch durch diese Isolation.



Port-Mapping: localhost:5432 auf dem Host wird zu 5432 im Container gebrückt

Lesen Sie es von links nach rechts: Ihre Delphi-App verbindet sich mit `localhost:5432`, als wäre eine Datenbank direkt auf Ihrem Rechner installiert. Sie hat keine Ahnung, dass ein Container existiert. Das Mapping `-p 5432:5432` leitet diesen Verkehr still über die Container-Grenze weiter zu PostgreSQL, das *innerhalb* der

Box auf `5432` lauscht. Die Zahl links vom Doppelpunkt ist der Host-Port, den Sie anwählen; die Zahl rechts ist der Port im Container. Hier stimmen sie überein, was die Sache einfach hält — müssen sie aber nicht: `-p 6000:5432` würde Ihnen dieselbe Datenbank stattdessen unter `localhost:6000` zugänglich machen, praktisch, wenn Host-Port 5432 bereits belegt ist.

Das ist die ganze Abstraktion. Für jedes Werkzeug auf Ihrem Host — Delphi, `psql`, eine GUI, was auch immer — sieht die containerisierte Datenbank exakt aus wie eine lokale auf `localhost:5432`.

Jetzt der Delphi-Teil: FireDAC spricht mit dem Container

Hier lächelt der erfahrene Delphi-Entwickler, denn von Pascals Seite aus ist hier überhaupt nichts Exotisches. Die Datenbank ist *einfach ein PostgreSQL-Server auf* `localhost:5432`. Sie verbinden sich mit FireDAC —

Delphis eingebauter, hochperformanter Datenbankzugriffsschicht — genau so, wie Sie es mit jedem PostgreSQL-Server täten, containerisiert oder nicht.

FireDAC identifiziert jede Datenbank-Engine über eine kurze **DriverID**, und die von PostgreSQL ist `PG`. Laut

Embarcaderos Leitfaden Connect to PostgreSQL (FireDAC)) wird eine PostgreSQL-Verbindung durch eine

Handvoll Parameter beschrieben — `DriverID`, `Server`, `Port`, `Database`, `User_Name` und `Password` —, und der Treiber lebt in der Unit `FireDAC.Phys.PG`. Hier ist ein vollständiges, fokussiertes Konsolenprogramm, das sich verbindet, eine Tabelle anlegt, zwei Zeilen einfügt und sie wieder ausliest.

```
program PgDemo;

{$APPTYPE CONSOLE}

uses
  System.SysUtils,
  FireDAC.Comp.Client,   // TFDConnection, TFDQuery
  FireDAC.Phys.PG,       // der PostgreSQL-Treiber (DriverID = PG)
  FireDAC.Phys.PGDef,    // seine Design-/Registrierungsdefinitionen
  FireDAC.Stan.Def,      // Treiber-/Verbindungsdefinitions-Manager
  FireDAC.Stan.Async,    // Unterstützung für asynchrone Ausführung
  FireDAC.DApt;          // der Datenadapter, der TFDQuery befüllt

var
  Conn: TFDConnection;
  Qry: TFDQuery;
begin
  try
```

```
    Conn := TFDConnection.Create(nil);
    try
      // Die Verbindung zu unserem containerisierten PostgreSQL beschreiben.
      Conn.Params.DriverID  := 'PG';
      Conn.Params.Values['Server']    := 'localhost';
      Conn.Params.Values['Port']      := '5432';
      Conn.Params.Values['Database']  := 'postgres';   // die Standard-DB
      Conn.Params.Values['User_Name'] := 'postgres';   // Standard-Superuser
      Conn.Params.Values['Password']  := 'secret';     // entspricht POSTGRES_PASSWORD
      Conn.Connected := True;
      Writeln('Connected to PostgreSQL in the container.');

      // CREATE + INSERT sind Anweisungen ohne Ergebnismenge: ExecSQL.
      Conn.ExecSQL(
        'CREATE TABLE IF NOT EXISTS widgets (' +
        '  id     SERIAL PRIMARY KEY,' +
        '  name   TEXT NOT NULL,' +
        '  price  NUMERIC(10,2) NOT NULL)');

      Conn.ExecSQL(
        'INSERT INTO widgets (name, price) VALUES (:n, :p)',
        ['Rocket Skates', 20.00]);
      Conn.ExecSQL(
        'INSERT INTO widgets (name, price) VALUES (:n, :p)',
        ['Portable Hole', 99.00]);
      Writeln('Inserted 2 rows.');

      // SELECT liefert Zeilen, also TFDQuery verwenden und die Ergebnismenge durchlaufen.
      Qry := TFDQuery.Create(nil);
      try
        Qry.Connection := Conn;
        Qry.Open('SELECT id, name, price FROM widgets ORDER BY id');
        while not Qry.Eof do
        begin
          Writeln(Format('#%d  %-16s  $%.2f',
            [Qry.FieldByName('id').AsInteger,
             Qry.FieldByName('name').AsString,
             Qry.FieldByName('price').AsFloat]));
          Qry.Next;
        end;
      finally
        Qry.Free;
      end;
    finally
      Conn.Free;
    end;
  except
    on E: Exception do
      Writeln('ERROR: ', E.ClassName, ': ', E.Message);
  end;
  Readln;
end.
```

Ein paar Dinge sind hervorzuheben. Beachten Sie die Aufteilung zwischen `TFDConnection.ExecSQL` und `TFDQuery`: `ExecSQL` ist für Anweisungen, die Daten *verändern*, aber keine Ergebnismenge liefern — `CREATE TABLE`, `INSERT`, `UPDATE`, `DELETE` —, während `TFDQuery.Open` ein `SELECT` ausführt und Ihnen eine navigierbare Ergebnismenge übergibt, die Sie mit der klassischen Schleife `while not Eof do … Next` durchlaufen, die jeder Delphi-Entwickler längst im Muskelgedächtnis hat. Die `:n` und `:p` sind FireDAC-*Parameter* — Platzhalter, die an die übergebenen Werte gebunden werden, sodass PostgreSQL korrekt typisierte, sicher maskierte Daten sieht statt zusammengestückeltes SQL. Führen Sie es aus, und die Konsole gibt Ihre zwei Widgets aus, direkt aus dem Container zurückgelesen:

```
Connected to PostgreSQL in the container.
Inserted 2 rows.
#1  Rocket Skates     $20.00
#2  Portable Hole     $99.00
```

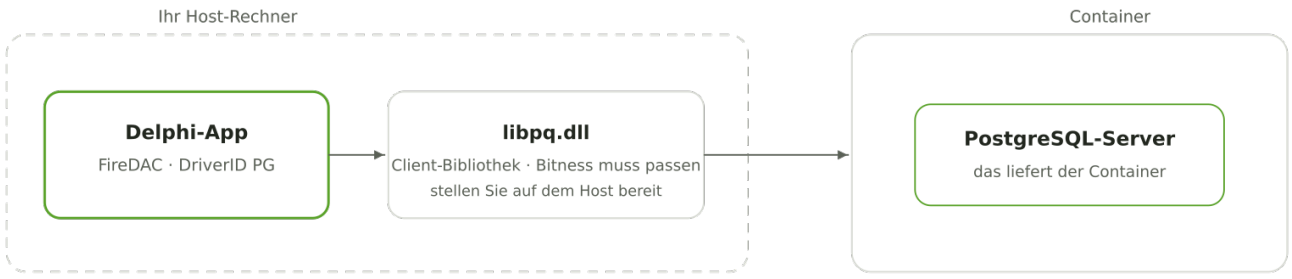
Diese Ausgabe ist die komplette Rundreise: ein modernes Delphi-Programm, das mit nichts als der eingebauten RTL und FireDAC einen echten PostgreSQL-Server anlegt und abfragt, den Sie mit einem einzigen `docker run` herbeigezaubert haben.

Der ehrliche Stolperstein: FireDAC braucht die Client-Bibliothek

Es gibt einen echten Haken zwischen Ihnen und dieser Ausgabe, und es ist besser, jetzt davon zu hören, als später gegen eine kryptische Fehlermeldung zu kämpfen. FireDACs nativer PostgreSQL-Treiber spricht das PostgreSQL-Wire-Protokoll nicht selbst — er delegiert an **libpq**, die offizielle C-Client-Bibliothek von PostgreSQL, unter Windows ausgeliefert als `libpq.dll`. Der Container hat Ihnen den *Server* gegeben; die *Client-Bibliothek* hat er **nicht** auf Ihren Host gelegt — und Ihre Delphi-Anwendung läuft auf Ihrem Host.

> **FireDAC braucht die PostgreSQL-Client-Bibliothek (libpq)**
> Laut Embarcaderos Dokumentation Connect to PostgreSQL (FireDAC)) benötigt der native PG-Treiber `libpq.dll` — plus die DLLs, von denen sie abhängt — zur Laufzeit erreichbar für Ihre Anwendung (im `PATH` oder direkt neben Ihrer `.exe`). Der Container ist der **Server**; libpq ist der **Client**, und er lebt bei *Ihrer App*, nicht in der Box. Zwei Regeln halten das schmerzfrei: Die **Bitness der DLL muss zu Ihrer App passen** — ein 64-Bit-Delphi-Build braucht die 64-Bit-`libpq.dll`, ein 32-Bit-Build die 32-Bit-Variante —, und idealerweise sollte die Client-Version nahe an der Server-Version liegen.

Woher bekommen Sie sie? libpq steckt im standardmäßigen PostgreSQL-Windows-Download (von EDB) — Sie können den Installer ausführen und nur die *Command Line Tools* auswählen, um `libpq.dll` und ihre Begleiter zu erhalten, ohne den Server selbst zu installieren, und dann diesen Satz DLLs neben Ihre `.exe` oder in Ihren `PATH` kopieren. Das ist eine einmalige Einrichtung auf jedem Rechner, auf dem die App läuft, und sobald die DLLs an Ort und Stelle sind, verbindet sich FireDAC ohne einen weiteren Gedanken. Es ist eine kleine Steuer, und sie ist vollständig reproduzierbar — kein Raten.



Server im Container, Client-Bibliothek auf dem Host — beide werden gebraucht

Lesen Sie die drei Boxen als Kette: Ihre Delphi-App ruft in `libpq.dll` auf dem Host hinein, und `libpq` ist das, was tatsächlich über den Port mit dem PostgreSQL-Server im Container spricht. Fehlt die mittlere Box, hat FireDAC keinen Weg zum Server — das ist der Stolperstein in einem Bild.

Dieselbe Datenbank heißt jeden Client willkommen

Weil dies ein Standard-PostgreSQL-Server ist, ist nichts daran Delphi-spezifisch — und das ist eine Stärke, die man ehrlich benennen sollte. Der Container spricht das gewöhnliche PostgreSQL-Protokoll, also verbindet sich *jeder* Client auf diesem Planeten mit exakt demselben `localhost:5432` und exakt denselben Zugangsdaten.

> **Denselben Container aus allem Möglichen erreichen**
>
> Einen Client kennen Sie bereits: `psql`, PostgreSQLs eigene Kommandozeile, per `docker exec` in Schritt 2. Lieber eine GUI? DBeaver und pgAdmin sind ausgezeichnete freie Werkzeuge, die sich mit `localhost:5432` verbinden und Tabellen visuell durchstöbern lassen. Sie schreiben ein Backend in einem anderen Stack? Ein Node.js-Dienst mit dem `pg`-Paket oder eine Python-App mit `psycopg` spricht mit diesem identischen Container. Der Datenbank ist es egal, wer anruft — genau deshalb ist ihre Containerisierung so befreiend. Ihre Delphi-App ist ein willkommener Client unter vielen, kein eingesperrter.

Diese Interoperabilität ist der stille Gewinn einer standardisierten, quelloffenen Datenbank im Container: Ihr Delphi-Frontend und, sagen wir, ein Web-Backend können sich eine Datenbank teilen, ohne dass eines wissen oder sich darum kümmern muss, worin das andere geschrieben ist.

Ein Haken zum Merken: diese Daten sind vergänglich

In allem oben steckt eine bewusste Vereinfachung, und Sie sollten davon wissen, bevor Sie sich für irgendetwas auf diesen Container verlassen, das Sie vermissen würden. Im Moment leben die Tabelle `widgets` und ihre Zeilen *innerhalb* des Containers.

> **Container weg, Daten weg**
>
> Löschen Sie diesen Container mit `docker rm`, und die Daten von PostgreSQL gehen mit — was für Wegwerf-Experimente *perfekt* ist (das ist das Versprechen vom blitzsauberen Rechner), aber offensichtlich nicht das, was Sie für Daten wollen, die Sie behalten möchten. Die Lösung ist ein **Volume**: ein Stück Speicher, das auf Ihrem Host lebt und jeden einzelnen Container überdauert, sodass Sie den Datenbank-Container zerstören und neu erstellen können, während die Daten bleiben. Wir brauchen es noch nicht, und es hier zu lehren würde das mentale Modell verwässern — deshalb führt Teil 4 Volumes sauber ein, wenn wir mit Docker Compose einen echten Stack zusammenbauen.

Für diesen Beitrag ist Vergänglichkeit ein Feature: Experimentieren Sie frei, und wenn Sie fertig sind, hinterlässt `docker stop pg-delphi && docker rm pg-delphi` Ihren Rechner exakt so, wie er vor dem Start war.

Fazit

Sie haben soeben eine produktionsreife Datenbank mit einem Befehl gestartet, mit den Werkzeugen, die Sie bereits kennen, aus Delphi mit ihr gesprochen und die eine Abstraktion gelernt — den veröffentlichten Port —, die einen containerisierten Dienst lokal wirken lässt.

> Ein PostgreSQL-Server ist jetzt nur noch ein `docker run` entfernt, und ihn aus Delphi zu erreichen ist gewöhnliches FireDAC gegen `localhost:5432` — vorausgesetzt, libpq fährt auf dem Host mit. Wenn Sie fertig sind: `docker rm`, und Ihr Rechner ist blitzsauber.

Drei Dinge lohnt es mitzunehmen. Erstens: `-p host:container` ist *die* Brücke — verinnerlichen Sie dieses eine

Flag, und Container hören auf, rätselhaft zu sein. Zweitens: Von Delphis Seite gibt es nichts Neues zu lernen — DriverID `PG`, Standard-FireDAC, eine ehrliche Abhängigkeit (libpq, passend zur Bitness Ihrer App). Drittens: Die Datenbank ist standardisiert und offen, also empfängt sie Delphi, `psql`, DBeaver oder ein Node-Backend mit derselben Gleichgültigkeit.

Eine Frage bleibt absichtlich offen: Woher kam dieses `postgres`-Image eigentlich, und woraus besteht es? Das ist das Thema von Teil 3 — **Images, Tags, Layer und Repositories** — die Anatomie des Dings, das Sie gerade

ohne einen zweiten Gedanken heruntergeladen haben. 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 —