NEWS · SEPTEMBER 2, 2026

Delphi und PostgreSQL von Grund auf: eine FireDAC-CLI in vier Units

Eine vollständige, kompilierende Delphi-Kommandozeilenanwendung, die einen PostgreSQL-Container startet, ihre eigene Datenbank anlegt, zwei Tabellen erzeugt, Zeilen einfügt und wieder ausliest — jede Unit, jeder Windows-Befehl und die eine DLL, über die fast jeder stolpert.

By Dr. Holger Flick

Die meisten Datenbankbeispiele hören genau da auf, wo es interessant wird. Man bekommt ein Formular mit einer daraufgelegten `TFDConnection`, ein fest verdrahtetes Passwort, ein Grid und einen Screenshot — und für alles, was daraus ein echtes Programm macht, ist man dann allein: wo die Einstellungen leben, wer die Datenbank anlegt, was passiert, wenn der Server nicht läuft, und wie man das Ganze baut, ohne in der IDE herumzuklicken.

Machen wir es also einmal vollständig. Am Ende dieses Beitrags läuft ein PostgreSQL-Server in einem Container, und eine Delphi-Kommandozeilenanwendung aus vier Quelldateien legt ihre eigene Datenbank an, erzeugt zwei Tabellen, fügt Zeilen innerhalb einer Transaktion ein, liest sie mit einem Join wieder aus und gibt einen formatierten Bericht aus. Jede Datei steht hier vollständig. Jeder Befehl lässt sich direkt in ein Windows-Terminal einfügen.

Zwischen Ihnen und dieser Ausgabe steht ein ehrlicher Haken — eine einzige DLL, die der Container nicht mitliefert — und er bekommt einen eigenen Abschnitt statt einer Fußnote, denn er ist es, der die meisten beim ersten Versuch stoppt.

Das Versprechen: rund 480 Zeilen Object Pascal, eine `compose.yaml` und vier Befehle, und Sie beherrschen den gesamten Stack ab der Portnummer.

Was wir bauen

Das fertige Programm ist ein kleines Kommandozeilenwerkzeug namens `WidgetShop`, und es tut vier Dinge, eines pro Befehl. Wer die Form vorher kennt, liest den folgenden Code deutlich schneller.

```
WidgetShop setup    create the database and the tables
WidgetShop seed     insert sample customers and orders
WidgetShop list     print every order, biggest first
WidgetShop reset    drop the tables again
```

Vier bewegliche Teile müssen dafür zusammenpassen, und es lohnt sich, sie zu benennen, bevor wir eines davon bauen.

Die vier beweglichen Teile: eine Compose-Datei startet den Server, libpq schlägt die Brücke, der Port ist die Tür

Von links nach rechts gelesen: Ihr Programm ruft FireDAC, FireDAC ruft `libpq.dll`, und `libpq` spricht über den veröffentlichten Port mit PostgreSQL im Container. Die Compose-Datei über dem Container zaubert die gesamte rechte Seite herbei. Drei dieser vier Kästen bekommen Sie geschenkt. Der mittlere — die Client-Bibliothek — ist das Stück, das Sie selbst dorthin legen müssen, und genau deshalb bekommt es unten einen eigenen Schritt.

Falls Container für Sie neu sind: die Docker-Serie auf dieser Seite führt durch die Installation von Docker Desktop und den ersten Container, und Teil 2 zeigt den kürzestmöglichen Postgres-und-FireDAC-Rundlauf mit einem einzigen `docker run`. Dieser Beitrag ist die Version, die Sie tatsächlich in ein Repository einchecken würden.

Schritt 1 — PostgreSQL mit einer Compose-Datei starten

Statt eines Absatzes voller `docker run`-Flags, der nur in Ihrer Shell-History lebt, wird der Server einmal in einer Datei beschrieben, die neben dem Quelltext liegt. Legen Sie einen Ordner für das Projekt an und speichern Sie darin Folgendes als `compose.yaml`.

```
# PostgreSQL for the WidgetShop example.
#
#   docker compose up -d      start the server in the background
#   docker compose ps         see whether it is healthy
#   docker compose down       stop it, keep the data
#   docker compose down -v    stop it and delete the data as well
#
# The application database ("widgetshop") is NOT created here on purpose --
# "WidgetShop.exe setup" creates it, so you can watch it happen.

services:
  db:
    image: postgres:17
    container_name: widgetshop-db
    environment:
      # The only variable the official image requires.
      POSTGRES_PASSWORD: secret
    ports:
      # host:container -- this one line is what lets Delphi reach the server.
      - "5432:5432"
    volumes:
      # Named volume: the data survives "docker compose down".
      - pgdata:/var/lib/postgresql/data
    healthcheck:
      # pg_isready ships inside the image; "healthy" means "accepting connections".
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 5s
      timeout: 5s
      retries: 10
    restart: unless-stopped

volumes:
  pgdata:
```

Das meiste davon kommt Ihnen bekannt vor, wenn Sie Compose schon einmal begegnet sind, aber drei Zeilen verdienen eine ordentliche Einführung. `POSTGRES_PASSWORD` ist die eine Einstellung, auf der das offizielle `postgres`-Image besteht — seine Dokumentation wird da deutlich: *"This environment variable is required for you to use the PostgreSQL image. It must not be empty or undefined."* Der `ports`-Eintrag veröffentlicht Container-Port 5432 auf Port 5432 Ihres Rechners, und genau diese eine Zuordnung ist die gesamte Brücke zwischen Delphi und der Datenbank. Das benannte Volume `pgdata` hält Ihre Zeilen über ein `docker compose down` hinweg am Leben, und es wird an genau dem Pfad eingehängt, den das Image verlangt.

Über diesen Pfad lohnt sich eine Anmerkung, denn er hat sich kürzlich geändert, und ein veraltetes Copy-and-paste beißt Sie dann.

> **Das Datenverzeichnis ist in PostgreSQL 18 umgezogen**
> Für **Postgres 17 und darunter** sagt die Image-Dokumentation, man solle *"Mount the data volume at `/var/lib/postgresql/data` and not at `/var/lib/postgresql` because mounts at the latter path WILL NOT PERSIST database data when the container is re-created."* Ab **18 aufwärts** ist das Image auf ein versionsspezifisches Datenverzeichnis umgestiegen und hat den Volume-Ort auf `/var/lib/postgresql` hochgezogen. Die Compose-Datei oben pinnt `postgres:17`, hier ist `/var/lib/postgresql/data` also richtig — wenn Sie dieses Tag aber anheben, lesen Sie die Image-Dokumentation noch einmal, bevor Sie annehmen, Ihr Volume decke die Daten weiterhin ab.

Der `healthcheck`-Block ist der kleine Luxus, der diese Datei erst richtig wertvoll macht. Er führt alle fünf Sekunden `pg_isready` aus — ein Werkzeug, das im Image mitgeliefert wird und laut PostgreSQL selbst

"is a utility for checking the connection status of a PostgreSQL database server". Sein Exit-Status ist die ganze Antwort: `0` heißt, der Server nimmt Verbindungen an, `1` heißt, er weist sie noch ab — genau das, was eine Datenbank in den ersten ein bis zwei Sekunden nach dem Start tut. Dockers Compose-Dateireferenz beschreibt `healthcheck` als Deklaration *"a check that's run to determine whether or not the service containers are 'healthy'"*, und dieser Status macht aus „der Container läuft" das viel nützlichere „die Datenbank wird Ihnen auch antworten".

Hochfahren

Ist die Datei gespeichert, startet ein einziger Befehl alles und legt dabei Netzwerk und Volume gleich mit an.

```
docker compose up -d
```

Prüfen, ob er gesund ist

Statt zu raten, fragen Sie Compose nach dem Status der Dienste in diesem Stack.

```
docker compose ps
```

Sie suchen eine `STATUS`-Spalte, in der `Up ... (healthy)` steht und nicht `Up ... (health: starting)`. Dieses Wort `healthy` ist der Healthcheck von oben, der zurückmeldet, und es ist Ihr Signal, dass ein Verbindungsversuch gelingen und nicht in einen Timeout laufen wird.

!!! tip "Auf „healthy" warten, in einer Zeile" Wenn Sie das skripten, ist `depends_on` der eingebaute Weg zu warten: Compose *"guarantees dependency services marked with* `service_healthy` *are 'healthy' before starting a dependent service."* Das wird in dem Moment wichtig, in dem ein zweiter Dienst dazukommt, der mit dieser Datenbank spricht — und genau das ist das Terrain von Teil 4 der Docker-Serie.

Schritt 2 — Das eine, was der Container nicht mitliefert

Hier ist der Haken, und es ist besser, ihm jetzt zu begegnen als als kryptischer Fehler um drei Uhr nachts. FireDACs nativer PostgreSQL-Treiber spricht das PostgreSQL-Protokoll nicht selbst. Er delegiert an **libpq**, PostgreSQLs offizielle C-Client-Bibliothek, die unter Windows als `libpq.dll` plus eine Handvoll Abhängigkeiten daherkommt. Der Container hat Ihnen den *Server* gegeben. Ihr Delphi-Programm läuft auf dem *Host*, und es braucht den *Client*.

> **FireDAC braucht libpq.dll auf Ihrem Rechner, passend zur Bitness Ihrer App**
> Laut Embarcaderos Anleitung Connect to PostgreSQL (FireDAC)) benötigt der native `PG`-Treiber `libpq.dll` samt abhängiger DLLs zur Laufzeit erreichbar — im `PATH` oder direkt neben Ihrer `.exe`. Zwei Regeln halten das schmerzfrei. Die **Bitness muss zu Ihrem Build passen**: eine Win64-Anwendung braucht die 64-Bit-`libpq.dll`, eine Win32-Anwendung die 32-Bit-Variante, und eine Verwechslung erzeugt einen Fehler, der so klingt, als fehle die DLL komplett. Und die Client-Version sollte einigermaßen nah an der Server-Version liegen.

Die DLLs zu besorgen ist eine einmalige Pflichtübung. Sie stecken im normalen PostgreSQL-Download für Windows, und Sie müssen dafür keinen Server installieren: Starten Sie das Installationsprogramm, wählen Sie nur die Komponente **command-line tools** aus und kopieren Sie dann `libpq.dll` und ihre Begleiter neben Ihre ausführbare Datei oder in den `PATH`. Sind sie einmal am Platz, denken Sie nie wieder daran.

Ob Ihr Rechner sie schon hat, fragen Sie Windows direkt:

```
where libpq.dll
```

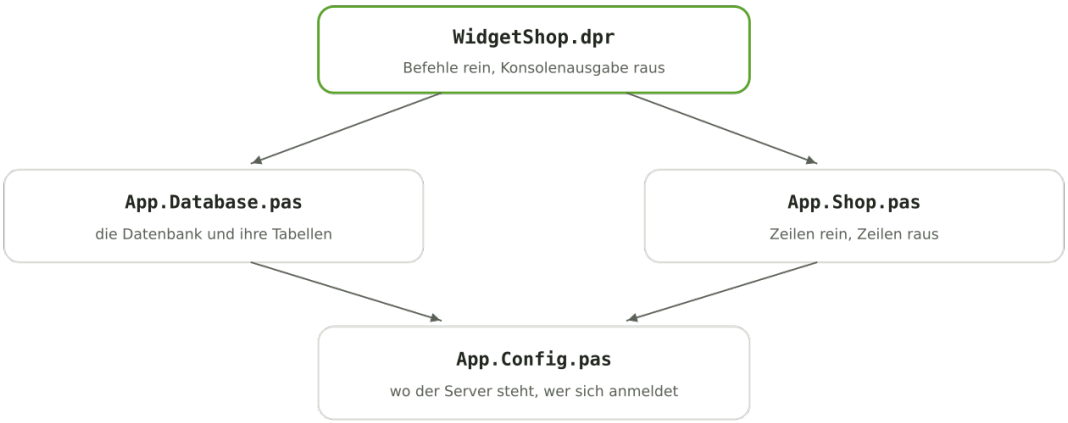
Keine Ausgabe heißt, sie sind noch nicht im `PATH`. Das ist ein behebbarer, einmaliger Zustand — kein Rätsel.

Schritt 3 — Das Projekt, vier Dateien nacheinander

Mit laufendem Server und vorhandener Client-Bibliothek ist die Delphi-Seite ganz gewöhnliches FireDAC. So ist der Code aufgeteilt, und das aus gutem Grund.

Datei	Wofür sie zuständig ist
`App.Config.pas`	Wo der Server steht und als wer wir uns anmelden
`App.Database.pas`	Datenbank anlegen, Verbindungen öffnen, das Schema
`App.Shop.pas`	Zeilen schreiben und wieder auslesen
`WidgetShop.dpr`	Befehlsauswertung und jede Zeile Konsolenausgabe

Diese Aufteilung ist kein Selbstzweck. Sie gibt dem Programm eine einzige Abhängigkeitsrichtung — und das ist die Eigenschaft, die Code später leicht änderbar macht.



Jede Unit hängt nur nach unten — das Programm kennt SQL, das SQL kennt das Programm nicht

Folgen Sie den Pfeilen, und eine Regel fällt heraus: nichts zeigt jemals nach oben. `App.Shop` weiß nicht, dass es eine Konsole gibt — an dem Tag, an dem daraus ein VCL-Formular oder ein REST-Dienst wird, ändert sich diese Unit überhaupt nicht. `App.Config` sitzt unten, weil alle wissen müssen, wo der Server steht, und niemand irgendetwas über `App.Config` wissen muss.

App.Config.pas — die Einstellungen, an genau einer Stelle

Die erste Unit beantwortet eine einzige Frage: Wo ist der Server, und wer sind wir? Sie in eine eigene Datei zu legen bedeutet, dass es genau eine Stelle gibt, an der man nachsieht, wenn eine Verbindung scheitert.

```
unit App.Config;

{ Connection settings for the PostgreSQL server started by compose.yaml.

  Everything the application needs to find the database lives here and nowhere
  else, so there is exactly one place to look when a connection fails. }

interface

uses
  FireDAC.Comp.Client;

type
  TAppConfig = record
    Host: string;
    Port: string;
    UserName: string;
    Password: string;
    /// The database this application owns and creates.
    Database: string;
    /// The database we log in to in order to create the one above.
    MaintenanceDatabase: string;
    /// Read the settings, letting environment variables override the defaults.
    class function Load: TAppConfig; static;
  end;

/// Point AConnection at ADatabase on the configured server.
procedure ConfigureConnection(AConnection: TFDConnection;
  const AConfig: TAppConfig; const ADatabase: string);

implementation

uses
  System.SysUtils;

function EnvOrDefault(const AName, ADefault: string): string;
begin
  Result := GetEnvironmentVariable(AName);
  if Result = '' then
    Result := ADefault;
end;

class function TAppConfig.Load: TAppConfig;
begin
  Result.Host := EnvOrDefault('WIDGETSHOP_HOST', 'localhost');
  Result.Port := EnvOrDefault('WIDGETSHOP_PORT', '5432');
  Result.UserName := EnvOrDefault('WIDGETSHOP_USER', 'postgres');
  Result.Password := EnvOrDefault('WIDGETSHOP_PASSWORD', 'secret');
  Result.Database := EnvOrDefault('WIDGETSHOP_DB', 'widgetshop');
  // Always present on a fresh PostgreSQL server, and never dropped.
  Result.MaintenanceDatabase := 'postgres';
end;

procedure ConfigureConnection(AConnection: TFDConnection;
  const AConfig: TAppConfig; const ADatabase: string);
begin
  AConnection.Params.Clear;
  AConnection.Params.DriverID := 'PG';
  AConnection.Params.Values['Server'] := AConfig.Host;
  AConnection.Params.Values['Port'] := AConfig.Port;
  AConnection.Params.Values['Database'] := ADatabase;
  AConnection.Params.Values['User_Name'] := AConfig.UserName;
  AConnection.Params.Values['Password'] := AConfig.Password;
  // A console application must never pop up a login dialog: fail loudly instead.
  AConnection.LoginPrompt := False;
```

```
end;

end.
```

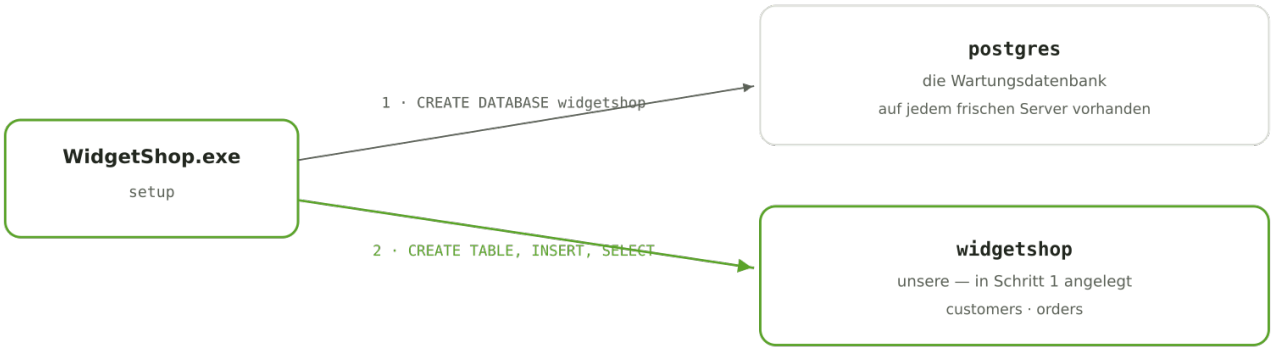
Drei Details leisten hier echte Arbeit. `DriverID := 'PG'` ist die Art, wie FireDAC den PostgreSQL-Treiber auswählt — jede Engine hat eine kurze Kennung, und die Parameternamen drumherum (`Server`, `Port`, `Database`, `User_Name`, `Password`) sind die, die Embarcadero für die Verbindung zu PostgreSQL) dokumentiert.

`LoginPrompt := False` ist wichtiger, als es aussieht: Eine Konsolenanwendung, die einen Anmeldedialog zeigen will, hängt entweder oder stürzt ab, und mit einer klaren Exception zu scheitern ist deutlich freundlicher. Und `EnvOrDefault` bedeutet, dass dieselbe kompilierte Anwendung mit `set WIDGETSHOP_HOST=...` auf den Server eines Kollegen zeigen kann, statt neu gebaut zu werden.

Beachten Sie außerdem, dass `TAppConfig` *zwei* Datenbanknamen trägt. Der zweite ist der Schlüssel zur nächsten Unit.

App.Database.pas — eine Datenbank anlegen, mit der man nicht verbunden ist

Jetzt das Henne-Ei-Problem: Um `CREATE DATABASE widgetshop` auszuführen, muss man mit PostgreSQL verbunden sein — aber man kann sich nicht mit `widgetshop` verbinden, weil es die noch nicht gibt. Der Ausweg ist eine Datenbank, die immer da ist.



Zwei Verbindungen, zwei Aufgaben: sich an der Wartungsdatenbank anmelden, um die eigene anzulegen

Das Bild zeigt den ganzen Trick: Die erste Verbindung geht an `postgres`, eine Datenbank, die jeder Server hat und niemand löscht, nur damit wir irgendwo stehen können, während wir `CREATE DATABASE` absetzen. Die zweite Verbindung geht an die gerade angelegte Datenbank, und alles Weitere passiert dort. Hier ist die Unit, die das erledigt.

```
unit App.Database;

{ Everything that brings the database itself into existence: creating it,
  opening a connection to it, and creating or dropping its tables. }

interface

uses
  FireDAC.Comp.Client,
  App.Config;

/// Create the application database unless it is already there.
/// Returns True when this call is the one that created it.
function EnsureDatabase(const AConfig: TAppConfig): Boolean;

/// Open a connection to the application database. The caller owns the result.
function OpenAppConnection(const AConfig: TAppConfig): TFDConnection;
```

```
/// Create the customers and orders tables. Safe to run repeatedly.
procedure CreateSchema(AConnection: TFDConnection);

/// Drop both tables: orders first, because it references customers.
procedure DropSchema(AConnection: TFDConnection);

implementation

uses
  System.SysUtils;

{ A database or table name cannot be passed as a query parameter, so it has to
  be pasted into the SQL text. That is exactly the shape SQL injection takes,
  so we refuse anything that is not a plain lower-case identifier. }
procedure ValidateIdentifier(const AName: string);
var
  C: Char;
begin
  if AName = '' then
    raise Exception.Create('Database name must not be empty.');
  for C in AName do
    if not CharInSet(C, ['a'..'z', '0'..'9', '_']) then
      raise Exception.CreateFmt(
        'Refusing to build SQL from %s: lower-case letters, digits and ' +
        'underscores only.', [QuotedStr(AName)]);
end;

function DatabaseExists(AConnection: TFDConnection; const AName: string): Boolean;
begin
  // pg_database is PostgreSQL's own catalog of every database on the server.
  Result := AConnection.ExecSQLScalar(
    'SELECT count(*) FROM pg_database WHERE datname = :name', [AName]) > 0;
end;

function EnsureDatabase(const AConfig: TAppConfig): Boolean;
var
  Conn: TFDConnection;
begin
  ValidateIdentifier(AConfig.Database);

  Conn := TFDConnection.Create(nil);
  try
    // You cannot connect to a database that does not exist yet, so we log in
    // to the maintenance database and create ours from there.
    ConfigureConnection(Conn, AConfig, AConfig.MaintenanceDatabase);
    Conn.Connected := True;

    Result := not DatabaseExists(Conn, AConfig.Database);
    if Result then
      Conn.ExecSQL('CREATE DATABASE ' + AConfig.Database);
  finally
    Conn.Free;
  end;
end;

function OpenAppConnection(const AConfig: TAppConfig): TFDConnection;
begin
  Result := TFDConnection.Create(nil);
  try
    ConfigureConnection(Result, AConfig, AConfig.Database);
    Result.Connected := True;
  except
    Result.Free;
    raise;
  end;
end;

procedure CreateSchema(AConnection: TFDConnection);
begin
```

```
  AConnection.ExecSQL(
    'CREATE TABLE IF NOT EXISTS customers ('    +
    '  id         SERIAL      PRIMARY KEY,'     +
    '  name       TEXT        NOT NULL,'        +
    '  email      TEXT        NOT NULL UNIQUE,' +
    '  created_at TIMESTAMPTZ NOT NULL DEFAULT now())');

  AConnection.ExecSQL(
    'CREATE TABLE IF NOT EXISTS orders ('  +
    '  id          SERIAL        PRIMARY KEY,' +
    '  customer_id INTEGER       NOT NULL'     +
    '                REFERENCES customers(id) ON DELETE CASCADE,' +
    '  product     TEXT          NOT NULL,'    +
    '  quantity    INTEGER       NOT NULL CHECK (quantity > 0),' +
    '  unit_price  NUMERIC(10,2) NOT NULL,'    +
    '  ordered_at  TIMESTAMPTZ   NOT NULL DEFAULT now())');
end;

procedure DropSchema(AConnection: TFDConnection);
begin
  AConnection.ExecSQL('DROP TABLE IF EXISTS orders');
  AConnection.ExecSQL('DROP TABLE IF EXISTS customers');
end;

end.
```

Ein paar Dinge darin lohnen einen zweiten Blick. `DatabaseExists` fragt `pg_database` ab, PostgreSQLs eigenen Katalog aller Datenbanken auf dem Server — ein deutlich besserer Test, als sich zu verbinden und den Fehlschlag abzufangen. `ValidateIdentifier` existiert, weil ein Datenbankname zu den wenigen Dingen gehört, die SQL nicht als Parameter annimmt: Er muss in den Anweisungstext hineinkonkateniert werden, und Konkatenation ist genau der Ort, an dem Injection lebt — also wird der Name zuerst gegen einen bewusst langweiligen Zeichenvorrat geprüft. Und `OpenAppConnection` gibt die Verbindung frei, falls `Connected := True` eine Exception wirft, damit ein nicht erreichbarer Server nicht auch noch ein Objekt leckt.

Das Schema selbst ist klein, aber kein Spielzeug. `SERIAL` gibt jeder Tabelle einen automatisch hochzählenden ganzzahligen Schlüssel. `REFERENCES customers(id) ON DELETE CASCADE` lässt die Datenbank selbst durchsetzen, dass eine Bestellung zu einem echten Kunden gehört, und räumt Bestellungen auf, wenn ein Kunde verschwindet. `CHECK (quantity > 0)` weist Unsinn ab, bevor er überhaupt gespeichert wird. `NUMERIC(10,2)` ist der richtige Typ für Geld — es speichert Dezimalstellen exakt, was ein Fließkommatyp nicht versprechen kann.

Unter `EnsureDatabase` liegt eine PostgreSQL-Regel, die eine Fehlermeldung erklärt, der Sie irgendwann begegnen werden.

> **CREATE DATABASE läuft nicht innerhalb einer Transaktion**
> PostgreSQLs Dokumentation sagt es unmissverständlich: *"*`CREATE DATABASE` *cannot be executed inside a transaction block."* Die Anweisung muss also über eine Verbindung hinausgehen, auf der keine Transaktion offen ist — deshalb ruft `EnsureDatabase` oben nie `StartTransaction` auf, und deshalb benutzt es eine eigene, kurzlebige Verbindung, statt sich eine zu borgen, die vielleicht mitten in einer Transaktion steckt. Wenn Ihnen jemals *"CREATE DATABASE cannot run inside a transaction block"* durch FireDAC entgegenkommt, dann spricht diese Regel. Dieselbe Seite hält fest, dass Sie außerdem Superuser sein oder das `CREATEDB`-Recht besitzen müssen — was der `postgres`-Benutzer aus unserer Compose-Datei ist.

App.Shop.pas — Zeilen schreiben und wieder auslesen

Diese Unit fasst die eigentlichen Daten an, und sie enthält mit Absicht überhaupt keine Ausgabe. Sie gibt Records zurück; das Programm entscheidet, was damit geschieht.

```
unit App.Shop;
```

```
{ The data the application actually cares about: writing sample rows and
  reading them back. No Writeln in here - this unit returns data, the program
  decides how to show it. }

interface

uses
  FireDAC.Comp.Client;

type
  /// One order joined to the customer who placed it.
  TOrderLine = record
    OrderId: Integer;
    Customer: string;
    Product: string;
    Quantity: Integer;
    UnitPrice: Currency;
    LineTotal: Currency;
  end;

/// Insert two customers and their orders. Returns the number of orders written.
function SeedSampleData(AConnection: TFDConnection): Integer;

/// Read every order together with its customer, most valuable line first.
function FetchOrderLines(AConnection: TFDConnection): TArray<TOrderLine>;

implementation

uses
  System.Generics.Collections,
  FireDAC.Stan.Param;

function InsertCustomer(AConnection: TFDConnection;
  const AName, AEmail: string): Integer;
var
  Qry: TFDQuery;
begin
  Qry := TFDQuery.Create(nil);
  try
    Qry.Connection := AConnection;
    // RETURNING hands the generated id straight back, so there is no second
    // round trip to ask "what number did you just give me?".
    Qry.SQL.Text :=
      'INSERT INTO customers (name, email) VALUES (:name, :email) RETURNING id';
    Qry.ParamByName('name').AsString := AName;
    Qry.ParamByName('email').AsString := AEmail;
    Qry.Open;
    Result := Qry.Fields[0].AsInteger;
  finally
    Qry.Free;
  end;
end;

procedure InsertOrder(AConnection: TFDConnection; ACustomerId: Integer;
  const AProduct: string; AQuantity: Integer; AUnitPrice: Currency);
begin
  AConnection.ExecSQL(
    'INSERT INTO orders (customer_id, product, quantity, unit_price) ' +
    'VALUES (:customer_id, :product, :quantity, :unit_price)',
    [ACustomerId, AProduct, AQuantity, AUnitPrice]);
end;

function SeedSampleData(AConnection: TFDConnection): Integer;
var
  AdaId, GraceId: Integer;
begin
  // Six inserts that belong together: either all of them land, or none do.
  AConnection.StartTransaction;
```

```
  try
    AdaId := InsertCustomer(AConnection, 'Ada Lovelace', 'ada@example.com');
    GraceId := InsertCustomer(AConnection, 'Grace Hopper', 'grace@example.com');

    InsertOrder(AConnection, AdaId, 'Analytical Engine Gear', 4, 129.50);
    InsertOrder(AConnection, AdaId, 'Punch Card Set', 12, 3.75);
    InsertOrder(AConnection, GraceId, 'Nanosecond Wire', 1, 11.80);
    InsertOrder(AConnection, GraceId, 'COBOL Manual', 2, 42.00);

    AConnection.Commit;
    Result := 4;
  except
    AConnection.Rollback;
    raise;
  end;
end;

function FetchOrderLines(AConnection: TFDConnection): TArray<TOrderLine>;
var
  Qry: TFDQuery;
  Lines: TList<TOrderLine>;
  Line: TOrderLine;
begin
  Lines := TList<TOrderLine>.Create;
  try
    Qry := TFDQuery.Create(nil);
    try
      Qry.Connection := AConnection;
      Qry.Open(
        'SELECT o.id, c.name AS customer, o.product, o.quantity, ' +
        '       o.unit_price, o.quantity * o.unit_price AS line_total ' +
        'FROM orders o ' +
        'JOIN customers c ON c.id = o.customer_id ' +
        'ORDER BY line_total DESC, o.id');

      while not Qry.Eof do
      begin
        Line.OrderId   := Qry.FieldByName('id').AsInteger;
        Line.Customer  := Qry.FieldByName('customer').AsString;
        Line.Product   := Qry.FieldByName('product').AsString;
        Line.Quantity  := Qry.FieldByName('quantity').AsInteger;
        Line.UnitPrice := Qry.FieldByName('unit_price').AsCurrency;
        Line.LineTotal := Qry.FieldByName('line_total').AsCurrency;
        Lines.Add(Line);
        Qry.Next;
      end;
    finally
      Qry.Free;
    end;
    Result := Lines.ToArray;
  finally
    Lines.Free;
  end;
end;

end.
```

Vier Techniken aus dieser Unit lohnen sich für Ihren eigenen Code. **Jeder Wert geht als Parameter hinein** — die Platzhalter `:name` und `:customer_id` — damit der Treiber sauber typisierte Werte schickt, statt dass Sie Zeichenketten zusammenkleben und hoffen, dass niemand O'Brien heißt. `RETURNING id` bittet PostgreSQL, den gerade erzeugten Schlüssel als Teil derselben Anweisung zurückzugeben, und genau deshalb ruft `InsertCustomer` `Open` statt `ExecSQL` auf: Eine Anweisung mit `RETURNING` liefert eine Ergebnismenge. **Das gesamte Befüllen läuft in einer Transaktion**, damit ein Fehlschlag beim fünften Insert Sie nicht mit zwei Kunden und drei Bestellungen zurücklässt; `Rollback` und danach `raise` macht die Arbeit rückgängig und lässt den Aufrufer trotzdem sehen, was schiefging. Und das `SELECT` erledigt seine Rechnerei und seine Sortierung **in der**

Datenbank — `o.quantity * o.unit_price AS line_total` berechnet PostgreSQL, und `ORDER BY line_total DESC` sortiert nach dieser berechneten Spalte, Arbeit, die Delphi nie wiederholen muss.

Die Trennung zwischen `ExecSQL` und `Open` bringt Einsteiger am häufigsten ins Straucheln, deshalb klar gesagt: `ExecSQL` ist für Anweisungen, die etwas ändern und keine Zeilen liefern, und `TFDQuery.Open` für Anweisungen, die Ihnen eine Ergebnismenge zum Durchlaufen geben.

WidgetShop.dpr — das Programm selbst

Die letzte Datei ist die einzige, die mit einem Menschen spricht. Sie wertet den Befehl aus, ruft in die Units oben hinein und gibt aus.

```
program WidgetShop;

{ A tiny PostgreSQL client for the command line.

    WidgetShop setup    create the database and the tables
    WidgetShop seed     insert sample customers and orders
    WidgetShop list     print every order, biggest first
    WidgetShop reset    drop the tables again

  The database it talks to is the one started by compose.yaml next to this file. }

{$APPTYPE CONSOLE}

uses
  System.SysUtils,
  FireDAC.Stan.Intf,
  FireDAC.Stan.Option,
  FireDAC.Stan.Error,
  FireDAC.Stan.Def,
  FireDAC.Stan.Pool,
  FireDAC.Stan.Async,
  FireDAC.Stan.Param,
  FireDAC.DatS,
  FireDAC.DApt,
  FireDAC.DApt.Intf,
  FireDAC.Phys,
  FireDAC.Phys.Intf,
  FireDAC.Phys.PG,
  FireDAC.Phys.PGDef,
  FireDAC.UI.Intf,
  FireDAC.ConsoleUI.Wait,
  FireDAC.Comp.Client,
  FireDAC.Comp.DataSet,
  App.Config in 'App.Config.pas',
  App.Database in 'App.Database.pas',
  App.Shop in 'App.Shop.pas';

const
  LineWidth = 71;

procedure PrintUsage;
begin
  Writeln('WidgetShop - a tiny PostgreSQL client written in Delphi');
  Writeln;
  Writeln('Usage: WidgetShop <command>');
  Writeln;
  Writeln('  setup   create the database and the tables');
  Writeln('  seed    insert sample customers and orders');
  Writeln('  list    print every order, biggest first');
  Writeln('  reset   drop the tables again');
end;

procedure RunSetup(const AConfig: TAppConfig);
var
```

```
  Conn: TFDConnection;
begin
  if EnsureDatabase(AConfig) then
    Writeln('Created database "', AConfig.Database, '".')
  else
    Writeln('Database "', AConfig.Database, '" already exists.');

  Conn := OpenAppConnection(AConfig);
  try
    CreateSchema(Conn);
    Writeln('Tables "customers" and "orders" are ready.');
  finally
    Conn.Free;
  end;
end;

procedure RunSeed(const AConfig: TAppConfig);
var
  Conn: TFDConnection;
begin
  Conn := OpenAppConnection(AConfig);
  try
    Writeln(Format('Inserted %d orders for 2 customers.',
      [SeedSampleData(Conn)]));
  finally
    Conn.Free;
  end;
end;

procedure RunList(const AConfig: TAppConfig);
var
  Conn: TFDConnection;
  Lines: TArray<TOrderLine>;
  Line: TOrderLine;
  Total: Currency;
begin
  Conn := OpenAppConnection(AConfig);
  try
    Lines := FetchOrderLines(Conn);
  finally
    Conn.Free;
  end;

  if Length(Lines) = 0 then
  begin
    Writeln('No orders yet - run "WidgetShop seed" first.');
    Exit;
  end;

  Writeln(Format('%-4s %-14s %-24s %4s %10s %10s',
    ['#', 'CUSTOMER', 'PRODUCT', 'QTY', 'PRICE', 'TOTAL']));
  Writeln(StringOfChar('-', LineWidth));

  Total := 0;
  for Line in Lines do
  begin
    Writeln(Format('%-4d %-14s %-24s %4d %10.2f %10.2f',
      [Line.OrderId, Line.Customer, Line.Product,
       Line.Quantity, Line.UnitPrice, Line.LineTotal]));
    Total := Total + Line.LineTotal;
  end;

  Writeln(StringOfChar('-', LineWidth));
  Writeln(Format('%-60s %10.2f', ['TOTAL ORDER VALUE', Total]));
end;

procedure RunReset(const AConfig: TAppConfig);
var
  Conn: TFDConnection;
```

```
begin
  Conn := OpenAppConnection(AConfig);
  try
    DropSchema(Conn);
    Writeln('Tables dropped. The database itself is still there.');
  finally
    Conn.Free;
  end;
end;

var
  Config: TAppConfig;
  Command: string;
begin
  try
    Config := TAppConfig.Load;

    if ParamCount = 0 then
      PrintUsage
    else
    begin
      Command := LowerCase(ParamStr(1));
      if Command = 'setup' then
        RunSetup(Config)
      else if Command = 'seed' then
        RunSeed(Config)
      else if Command = 'list' then
        RunList(Config)
      else if Command = 'reset' then
        RunReset(Config)
      else
      begin
        Writeln(ErrOutput, 'Unknown command: ', Command);
        Writeln;
        PrintUsage;
        ExitCode := 2;
      end;
    end;
  except
    on E: Exception do
    begin
      // A CLI reports failure on stderr and with a non-zero exit code, so a
      // build script or a scheduled task can tell that something went wrong.
      Writeln(ErrOutput, E.ClassName, ': ', E.Message);
      ExitCode := 1;
    end;
  end;
end.
```

Diese `uses`-Klausel ist lang, und sie ist aus einem Grund lang, den man kennen sollte. Wenn Sie Komponenten auf ein Formular ziehen, fügt die IDE diese Units still für Sie hinzu; in einem handgeschriebenen Konsolenprogramm fügen Sie sie selbst hinzu. Zwei davon vergisst man besonders gern. `FireDAC.Phys.PG` und `FireDAC.Phys.PGDef` sind es, die den PostgreSQL-Treiber überhaupt registrieren — ohne sie scheitert `DriverID := 'PG'` zur Laufzeit mit einem „Treiber nicht gefunden", obwohl alles kompiliert. Und `FireDAC.ConsoleUI.Wait` ist der konsolentaugliche Ersatz für die dialogbasierte Warte-UI, die eine VCL-Anwendung verwenden würde; lassen Sie sie weg, beschwert sich FireDAC, dass ihm eine UI fehlt.

Der letzte Block ist klein, aber er ist es, der ein Kommandozeilen*werkzeug* von einem Programm unterscheidet, das zufällig etwas ausgibt. Fehler gehen an `ErrOutput` statt an die Standardausgabe, damit ein Aufrufer den Bericht von der Beschwerde trennen kann. `ExitCode` ist `1` bei einem Fehlschlag und `2` bei einem unbekannten Befehl, sodass eine Batchdatei, eine geplante Aufgabe oder ein CI-Job auf das Ergebnis verzweigen kann, statt Text zu durchsuchen.

Eine Meinung: Lassen Sie die Anwendung ihre Datenbank selbst anlegen

Hier ist eine Entscheidung, die ich bewusst getroffen habe, und ich sage offen dazu, dass sie eine Vorliebe ist und keine Tatsache. Die Compose-Datei hätte die Datenbank für uns anlegen können — das offizielle Image liest eine Variable `POSTGRES_DB`, die *"can be used to define a different name for the default database that is created when the image is first started."* Eine Zeile, und `widgetshop` hätte existiert, bevor Delphi je gelaufen wäre. Ich habe das nicht getan, und für ein Programm wie dieses halte ich die Variante oben für besser.

Meine Begründung: Ein Werkzeug, das sich selbst hochziehen kann, ist ein Werkzeug, das Sie jemand anderem in die Hand drücken können. `WidgetShop setup` funktioniert gegen einen Container, gegen den Server eines Kollegen, gegen eine frische Cloud-Instanz — überall dort, wo die Zugangsdaten es hineinlassen. In dem Moment, in dem die Existenz der Datenbank zu einer Eigenschaft Ihrer Compose-Datei wird, läuft die Anwendung nur noch in der einen Umgebung, die diese Compose-Datei beschreibt, und „bei mir läuft es" hat ein neues Versteck.

Dieses Argument hat allerdings echte Grenzen, und sie zu verschweigen wäre Ihnen gegenüber unfair.

> **Wo das nicht gilt**
> Eine Datenbank anzulegen erfordert erhöhte Rechte: PostgreSQLs Dokumentation ist eindeutig, *"to create a database, you must be a superuser or have the special* `CREATEDB` *privilege."* In der Produktion wollen Sie ganz sicher **nicht**, dass sich Ihre Anwendung mit solcher Macht anmeldet — eine kompromittierte Anwendung soll keine Datenbanken anlegen oder löschen können. Das erwachsene Muster ist das Gegenteil dieses hier: Ein Administrator oder ein Infrastrukturwerkzeug legt die Datenbank und eine eingeschränkte Rolle an, und die Anwendung meldet sich als diese eingeschränkte Rolle an und fasst nur ihre eigenen Tabellen an. Schemaänderungen gehören dann in versionierte Migrationsskripte statt in ein `CREATE TABLE IF NOT EXISTS` beim Start, damit Upgrades wiederholbar und prüfbar sind. Selbst-Bootstrapping ist hervorragend für Entwicklung, Demos und Einzelplatz-Desktopwerkzeuge. Für einen Mehrbenutzerdienst in Produktion ist es die falsche Voreinstellung.

Nutzen Sie das Muster also dort, wo es passt — und wissen Sie genau, wann Sie es aufgeben. Diese Grenze ist der nützliche Teil.

Schritt 4 — Bauen

Sie können `WidgetShop.dproj` in der IDE öffnen und wie jedes andere Projekt F9 drücken, das ist der kürzeste Weg. Aber der Bau aus dem Terminal lohnt sich ebenfalls, denn genau das tut ein Build-Server, und er macht aus „kompiliert das noch?" eine Fünf-Sekunden-Frage.

Aus dem Terminal bauen

RAD Studio liefert eine Batchdatei mit, die Compiler und Suchpfade in Ihre Umgebung legt, und danach erledigt MSBuild die Arbeit. Führen Sie diese zwei Zeilen im Projektordner in `cmd.exe` aus:

```
call "C:\Program Files (x86)\Embarcadero\Studio\37.0\bin\rsvars.bat"
msbuild WidgetShop.dproj /t:Build /p:Config=Release /p:Platform=Win64
```

Passen Sie `37.0` an Ihre RAD-Studio-Version an und tauschen Sie `Win64` gegen `Win32`, wenn das Ihr Ziel ist — denken Sie nur daran, dass die Wahl zur Bitness der `libpq.dll` in Ihrem `PATH` passen muss. Die ausführbare Datei landet in `.\Win64\Release\WidgetShop.exe`.

Die Projektdatei selbst ist ganz gewöhnliches MSBuild-XML. Der Teil, den Sie von Hand bearbeiten würden, ist die Liste der Quelldateien — dort werden die drei Units hereingeholt:

```
<ItemGroup>
    <DelphiCompile Include="$(MainSource)">
        <MainSource>MainSource</MainSource>
    </DelphiCompile>
    <DCCReference Include="App.Config.pas"/>
    <DCCReference Include="App.Database.pas"/>
    <DCCReference Include="App.Shop.pas"/>
    <BuildConfiguration Include="Base">
        <Key>Base</Key>
    </BuildConfiguration>
    <BuildConfiguration Include="Debug">
        <Key>Cfg_2</Key>
        <CfgParent>Base</CfgParent>
    </BuildConfiguration>
    <BuildConfiguration Include="Release">
        <Key>Cfg_1</Key>
        <CfgParent>Base</CfgParent>
    </BuildConfiguration>
</ItemGroup>
```

Die vollständige `.dproj` — Plattformen Win32 und Win64, Konfigurationen Debug und Release — liegt neben den Quelldateien, statt hier noch einmal abgedruckt zu werden, denn hundert Zeilen Build-XML würden die Teile dieses Beitrags ertränken, die tatsächlich etwas beibringen.

Schritt 5 — Ausführen

Mit gesundem Container und gebauter Anwendung laufen die drei Befehle in der erwarteten Reihenfolge. Beginnen Sie mit `setup`.

```
WidgetShop.exe setup
```

```
Created database "widgetshop".
Tables "customers" and "orders" are ready.
```

Führen Sie ihn ein zweites Mal aus, ändert sich die erste Zeile zu `Database "widgetshop" already exists.` — das ist `DatabaseExists` bei der Arbeit, und es macht den Befehl sicher genug für ein Skript. Als Nächstes ein paar Zeilen hinein.

```
WidgetShop.exe seed
```

```
Inserted 4 orders for 2 customers.
```

Und schließlich wieder auslesen, verbunden mit ihren Kunden und nach Wert sortiert.

```
WidgetShop.exe list
```

```
#    CUSTOMER       PRODUCT                   QTY      PRICE      TOTAL
-------------------------------------------------------------------------
1    Ada Lovelace   Analytical Engine Gear      4     129.50     518.00
4    Grace Hopper   COBOL Manual                2      42.00      84.00
2    Ada Lovelace   Punch Card Set             12       3.75      45.00
3    Grace Hopper   Nanosecond Wire             1      11.80      11.80
-------------------------------------------------------------------------
TOTAL ORDER VALUE                                                658.80
```

Beachten Sie die Reihenfolge: Zeile `4` steht über Zeile `2`, weil nach Zeilensumme sortiert wird und nicht nach Id. PostgreSQL hat `quantity * unit_price` berechnet und danach sortiert, und Delphi hat schlicht ausgegeben, was zurückkam. Ein kleiner Hinweis zu dieser Ausgabe: `Format` verwendet bei `%.2f` das Dezimaltrennzeichen aus den Regionseinstellungen des Rechners, auf einem deutschen oder französischen Windows erscheinen diese Zahlen also mit Komma. Wenn die Ausgabe überall zeichengleich sein muss, weil etwas nachgelagert sie auswertet, übergeben Sie `Format` ein explizites `TFormatSettings`, statt es die Locale lesen zu lassen.

Zum Schluss ist der Abbau symmetrisch. `WidgetShop.exe reset` löscht die Tabellen, behält aber die Datenbank; `docker compose down` stoppt den Server, behält aber das Volume; und `docker compose down -v` löscht auch das Volume und damit jede Zeile.

Wenn es schiefgeht

Vier Fehler machen fast jeden misslungenen ersten Versuch aus, und alle vier sind schnell erledigt, sobald man sie erkennt. Das steckt jeweils wirklich dahinter.

Was Sie sehen	Was es tatsächlich bedeutet
Der `PG`-Treiber lässt sich nicht laden, oder `libpq.dll` wird nicht gefunden	`libpq.dll` fehlt im `PATH`, oder ihre Bitness passt nicht zu Ihrem Build
Verbindung auf `localhost:5432` abgelehnt	Der Container läuft nicht oder ist noch nicht gesund — `docker compose ps` prüfen
Passwortauthentifizierung für Benutzer `postgres` fehlgeschlagen	Das Passwort in `App.Config` passt nicht zu `POSTGRES_PASSWORD` in `compose.yaml`
`CREATE DATABASE cannot run inside a transaction block`	Die Anweisung ging über eine Verbindung mit offener Transaktion hinaus

Der zweite Fall hat eine besonders häufige Spielart: Der Container läuft, aber die Datenbank startet noch, also wird die Verbindung ein, zwei Sekunden lang abgelehnt. Genau davon soll Ihnen der Healthcheck in der Compose-Datei erzählen — warten Sie auf `(healthy)`, nicht bloß auf `Up`.

Dieselbe Datenbank empfängt auch andere Werkzeuge

Nichts von dem, was der Server hier tut, ist Delphi-spezifisch, und das ist eine echte Stärke und kein Vorbehalt. Es ist ein normaler PostgreSQL-Server auf einem normalen Port, also ist alles willkommen, was das Protokoll spricht — was zugleich bedeutet, dass Sie nirgends festsitzen.

> **Weitere gute Wege zu genau diesem Container**
>
> PostgreSQLs eigener Client ist einen Befehl entfernt, im Container: `docker exec -it widgetshop-db psql -U postgres -d widgetshop` setzt Sie an eine SQL-Eingabeaufforderung. Für einen grafischen Rundgang durch die Tabellen sind DBeaver und pgAdmin beide kostenlos und verbinden sich mit `localhost:5432` und den obigen Zugangsdaten. Und in der Pascal-Welt ist FireDAC nicht der einzige Weg: ZeosLib ist ein seit

> langem gepflegtes quelloffenes Komponentenpaket für PostgreSQL, Firebird, MySQL, SQLite und mehr — für Delphi, C++Builder und Free Pascal gleichermaßen; und wer mit Lazarus arbeitet, erreicht PostgreSQL über das freie und quelloffene SQLdb-Paket mit dessen Komponente `TPQConnection`. Andere Werkzeuge, dieselbe Datenbank — und jede Zeile SQL von oben lässt sich unverändert übertragen.

FireDAC verdient seinen Platz, weil es direkt mitgeliefert wird, mit starker PostgreSQL-Unterstützung und einem Komponentenmodell, das die meisten Delphi-Entwickler im Schlaf beherrschen. Die quelloffenen Alternativen verdienen ihren Platz über die Lizenz und darüber, dass sie Free Pascal erreichen. Was passt, hängt davon ab, was Sie bauen — nicht davon, was besser ist.

Fazit

Sie haben jetzt eine Datenbankanwendung statt eines Datenbank-Schnipsels: einen Server, der in einer eincheckbaren Datei beschrieben ist, eine Anwendung, die ihr eigenes Schema anlegt, parametrisiertes SQL, eine Transaktion, die sauber zurückrollt, einen Join, der dort berechnet wird, wo Joins hingehören, und einen Build, den Sie aus dem Terminal starten können, ohne die Maus anzufassen.

> Ein PostgreSQL-Server ist ein `docker compose up -d` entfernt, und ein vollständiger Delphi-Client sind vier kleine Dateien — solange `libpq.dll` auf dem Host liegt und zur Bitness Ihres Builds passt.

Drei Dinge lohnen sich zum Mitnehmen. Erstens: Die Portzuordnung und die Client-Bibliothek sind die einzigen zwei Installationsteile, die je wirklich Ärger machen; alles darüber ist gewöhnliches FireDAC. Zweitens: Eine Unit für die Einstellungen, eine für das Schema, eine für die Daten und eine für den Menschen hält SQL von der ersten Zeile an aus Ihrem UI-Code heraus — was heute nichts kostet und Ihnen später eine Neufassung spart. Drittens: Selbst-Bootstrapping ist eine Entwicklungsbequemlichkeit und keine Produktionsarchitektur — wissen Sie, wann diese Aufgabe an einen Administrator und ein Migrationsskript gehört.

Jede Unit oben kompiliert sauber mit dem Kommandozeilencompiler von RAD Studio 13, sowohl für Win32 als auch für Win64, und das vollständige Projekt — Quellen, `.dproj` und `compose.yaml` — liegt im Ordner `WidgetShop` neben diesem Beitrag, bereit zum Bauen.

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 —