NEWS · OCTOBER 2, 2026

WebStencils und SQLite in Delphi: Eine Datenbank, die der Server selbst anlegt

Eine WebStencils-Seite, gespeist aus einer SQLite-Datenbank, die das Delphi-Programm beim Start selbst anlegt -- Datei, Tabelle und Beispieldaten --, direkt aus einer FireDAC-Query in einem WebBroker-Server gerendert.

By Dr. Holger Flick

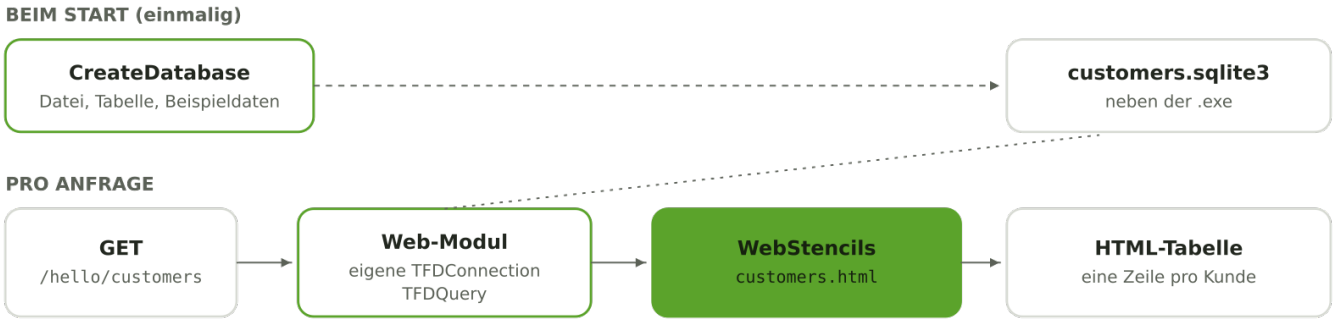
Im vorherigen Beitrag hat unsere erste WebStencils-Seite eine Liste von Objekten gerendert, die wir im Code erzeugt haben. Um die Syntax kennenzulernen, war das in Ordnung, aber so sieht keine echte Seite aus. Echte Seiten zeigen Daten aus einer Datenbank, und genau das machen wir heute.

Eine Anforderung habe ich mir selbst gestellt: Die Datenbank darf nichts sein, was Sie herunterladen, wiederherstellen oder von Hand einrichten müssen. Der Server legt sie an. Beim ersten Start erzeugt er die SQLite-Datei, die Tabelle und eine Handvoll Beispieldatensätze. Bei jedem weiteren Start findet er alles vor und lässt es in Ruhe. Ich habe aufgehört zu zählen, wie viele Beispielprojekte ich nicht starten konnte, weil eine Datenbankdatei fehlte oder ein Skript auf einen Pfad auf dem Rechner von jemand anderem verwies -- fügen wir dieser Liste kein weiteres hinzu.

Wir verwenden SQLite, das eine komplette Datenbank in einer einzigen Datei hält und keinen Server braucht, sowie FireDAC, die Datenbankbibliothek von Delphi. Beide werden mit Delphi ausgeliefert, der Stack bleibt also derselbe wie in der gesamten Serie: Über Delphi hinaus muss nichts installiert werden.

Was wir bauen

Bevor wir loslegen, schauen wir uns an, wann welcher Teil des Codes läuft, denn es gibt zwei klar getrennte Zeitpunkte. Das Programm legt die Datenbank genau einmal an, beim Start, bevor der HTTP-Server die erste Anfrage annimmt. Danach öffnet jede Anfrage an die Kundenseite eine Query über die Verbindung ihres Web-Moduls und übergibt sie an WebStencils.



Die Datenbank entsteht einmal beim Start; jede Anfrage fragt sie über die Verbindung ihres eigenen Web-Moduls ab

Die gepunktete Linie vom Web-Modul zur Datei ist das entscheidende Detail: Jedes Web-Modul spricht über seine eigene Verbindung mit derselben Datenbankdatei. Erinnern Sie sich an den ersten Beitrag dieser Serie: WebBroker erzeugt ein Web-Modul pro gleichzeitiger Anfrage. Genau deshalb kann jedes Modul eine eigene Verbindung besitzen, ohne dass wir selbst irgendetwas sperren müssen -- und genau deshalb muss SQLite mehreren offenen Verbindungen erlauben, sich die Datei zum Lesen und Schreiben zu teilen, was die Standardeinstellung von FireDAC, wie wir gleich sehen werden, nicht tut.

Falls FireDAC neu für Sie ist: Mein früherer Beitrag zu Delphi und PostgreSQL von Grund auf baut Schritt für Schritt eine FireDAC-Anwendung auf und legt dabei auch eine Datenbank im Code an. Mit SQLite fällt dieser Teil heute deutlich kürzer aus.

Schritt 1: Die Datenbank im Code anlegen

Alles, was die Datenbank betrifft, kommt in eine Unit. Legen Sie `CustomerData.pas` an:

```
unit CustomerData;

interface

uses
  FireDAC.Comp.Client;

function DatabaseFileName: string;
function CreateConnection: TFDConnection;
procedure CreateDatabase;

implementation

uses
  System.SysUtils,
  System.IOUtils,
  FireDAC.Stan.Def,
  FireDAC.Stan.Async,
  FireDAC.DApt,
  FireDAC.Phys.SQLite,
  FireDAC.Phys.SQLiteWrapper.Stat,
  FireDAC.ConsoleUI.Wait;
```

```
const
  SEED: array[0..5, 0..1] of string = (
    ('Los Pollos Hermanos', 'Albuquerque'),
    ('Wayne Enterprises', 'Gotham City'),
    ('Dunder Mifflin', 'Scranton'),
    ('Cyberdyne Systems', 'Sunnyvale'),
    ('Oscorp', 'New York'),
    ('Stark Industries', 'New York'));

function DatabaseFileName: string;
begin
  // the database file sits next to the executable
  Result := TPath.Combine(ExtractFilePath(ParamStr(0)), 'customers.sqlite3');
end;

function CreateConnection: TFDConnection;
begin
  Result := TFDConnection.Create(nil);
  Result.LoginPrompt := False;
  Result.Params.DriverID := 'SQLite';
  Result.Params.Database := DatabaseFileName;

  // (1) create the file if it does not exist yet
  Result.Params.Values['OpenMode'] := 'CreateUTF8';

  // (2) FireDAC opens SQLite exclusively by default; several web
  // modules with their own connections need shared access
  Result.Params.Values['LockingMode'] := 'Normal';
  Result.Params.Values['SharedCache'] := 'False';
end;

procedure CreateDatabase;
var
  Connection: TFDConnection;
  I: Integer;
begin
  Connection := CreateConnection;
  try
    Connection.Connected := True;

    // (3) the schema -- safe to run on every start
    Connection.ExecSQL(
      'CREATE TABLE IF NOT EXISTS customers (' +
      '  id INTEGER PRIMARY KEY AUTOINCREMENT,' +
      '  name TEXT NOT NULL,' +
      '  city TEXT NOT NULL)');

    // (4) sample data, but only into an empty table
    if Connection.ExecSQLScalar('SELECT COUNT(*) FROM customers') = 0 then
    begin
      Connection.StartTransaction;
      try
        for I := Low(SEED) to High(SEED) do
          Connection.ExecSQL(
            'INSERT INTO customers (name, city) VALUES (:name, :city)',
            [SEED[I, 0], SEED[I, 1]]);
        Connection.Commit;
      except
        Connection.Rollback;
        raise;
      end;
    end;
  finally
    Connection.Free;
  end;
end;

end.
```

Die nummerierten Kommentare der Reihe nach:

1. `OpenMode=CreateUTF8` weist den FireDAC-SQLite-Treiber an, die Datenbankdatei anzulegen, falls sie noch nicht existiert. Das ist zwar der Standard, aber in einer Unit, deren einziger Zweck das Anlegen einer Datenbank ist, schreibe ich es lieber ausdrücklich hin.
2. `LockingMode=Normal` ist die Zeile, die Ihnen einen Nachmittag erspart. FireDAC öffnet SQLite-Datenbanken standardmäßig im exklusiven Sperrmodus. Das ist die schnellste Variante, solange es nur eine einzige Verbindung gibt. Im exklusiven Modus behält eine Verbindung, die einmal aus der Datei gelesen hat, ihre Sperre, bis sie geschlossen wird. Unsere Seite, die nur liest, bemerkt davon nichts: Ich habe die Zeile entfernt, 20 parallele Anfragen an den Server geschickt, und jede lieferte alle sechs Zeilen. Die Falle schnappt beim ersten Schreibzugriff zu. Unsere Web-Module halten ihre Verbindungen offen, solange sie leben, und so scheitert jede andere Verbindung, die schreiben will -- ein weiteres Web-Modul oder ein Datenbankwerkzeug auf Ihrem Desktop --, mit `database is locked`. Im normalen Modus werden die Sperren nach jeder Transaktion freigegeben, und `SharedCache=False` gibt jeder Verbindung ihren eigenen Cache. Die SQLite-Datenbank in Embarcaderos eigenen WebStencils-Demos ist mit genau diesen beiden Einstellungen konfiguriert.
3. `CREATE TABLE IF NOT EXISTS` kann bei jedem Start gefahrlos ausgeführt werden. Beim ersten Start legt es die Tabelle an, bei jedem weiteren tut es nichts.
4. Die Beispieldaten werden nur eingefügt, wenn die Tabelle leer ist, damit ein Neustart des Servers sie nicht verdoppelt. Sie werden in einer einzigen Transaktion eingefügt, sodass nichts halb fertig zurückbleibt, wenn ein Insert fehlschlägt. Außerdem ist es schneller: Ohne explizite Transaktion ist jedes einzelne `INSERT` eine eigene Transaktion, und genau das nennt die SQLite-FAQ als klassischen Grund für langsame Inserts.

Beachten Sie die Units im Implementation-Abschnitt. `FireDAC.Phys.SQLite` registriert den Treiber, und `FireDAC.Phys.SQLiteWrapper.Stat` linkt die SQLite-Engine statisch in die ausführbare Datei, sodass keine `sqlite3.dll` ausgeliefert werden muss. `FireDAC.ConsoleUI.Wait` stellt den Wartecursor von FireDAC für eine Konsolenanwendung bereit. In meinen Tests mit RAD Studio 13 lief der Server auch ohne diese Unit, denn FireDAC behandelt jeden Thread außer dem Hauptthread als "silent" und fragt dort nie nach einem Wartecursor. Jeder Codepfad, der doch danach fragt, bricht ohne eingebundene Wait-Unit allerdings mit einem "factory missing"-Fehler ab, und die Unit kostet nichts, also behalte ich sie. Das WebBroker-Projekt in den WebStencils-Demos verwendet für seine SQLite-Datenbank dieselben Units.

Beachten Sie außerdem, dass es sich bei den Beispieldaten um erfundene Firmen aus Film und Fernsehen handelt. Falls Sie Post von den Anwälten von Wayne Enterprises bekommen: Ich habe Ihnen nicht gesagt, dass Sie diese Zeile einfügen sollen.

Schritt 2: Die Datenbank beim Start anlegen

Das Programm ruft `CreateDatabase` auf, bevor es mit dem Lauschen beginnt. Fügen Sie `CustomerData` der `uses`-Klausel von `HelloServer.dpr` hinzu und erweitern Sie den Anfang des Hauptblocks:

```
    // (0) create the database file, table, and sample data if needed
    CreateDatabase;
    WriteLn('Database: ', DatabaseFileName);

    // (1) tell WebBroker which class answers the requests
    WebRequestHandler.WebModuleClass := THelloModule;
```

Schlägt das Anlegen der Datenbank fehl -- etwa weil der Ordner schreibgeschützt ist --, landet die Exception im vorhandenen `except`-Block, und der Server startet gar nicht erst. Das ist Absicht. Ein Webserver, der ohne seine Datenbank startet, verschiebt den Fehler nur auf die erste Anfrage, wo er deutlich schwerer zu entdecken ist.

Schritt 3: Die Datenbank im Web-Modul abfragen

Das Web-Modul bekommt eine eigene Verbindung. Fügen Sie `FireDAC.Comp.Client` der `uses`-Klausel im Interface-Abschnitt und `CustomerData` der im Implementation-Abschnitt hinzu, und ergänzen Sie die Klasse um ein Feld und einen Destruktor:

```
  THelloModule = class(TWebModule)
  private
    FConnection: TFDConnection;
    procedure AddRoute(const APathInfo: string; AMethod: TMethodType;
      AHandler: THTTPMethodEvent; ADefault: Boolean = False);
    procedure SendJson(Response: TWebResponse; AStatusCode: Integer;
      AJson: TJSONObject);
    procedure Cors(Sender: TObject; Request: TWebRequest;
      Response: TWebResponse; var Handled: Boolean);
    procedure HelloAction(Sender: TObject; Request: TWebRequest;
      Response: TWebResponse; var Handled: Boolean);
    procedure AddAction(Sender: TObject; Request: TWebRequest;
      Response: TWebResponse; var Handled: Boolean);
    procedure HomeAction(Sender: TObject; Request: TWebRequest;
      Response: TWebResponse; var Handled: Boolean);
    procedure CustomersAction(Sender: TObject; Request: TWebRequest;
      Response: TWebResponse; var Handled: Boolean);
    procedure NotFoundAction(Sender: TObject; Request: TWebRequest;
      Response: TWebResponse; var Handled: Boolean);
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  end;
```

Der Konstruktor erzeugt die Verbindung und registriert die neue Route; der Destruktor gibt die Verbindung wieder frei:

```
constructor THelloModule.Create(AOwner: TComponent);
begin
  inherited;

  // (10) every web module instance gets its own connection
  FConnection := CreateConnection;

  // (1) runs before any action -- adds the CORS headers
  BeforeDispatch := Cors;

  // (2) one action per URL
  AddRoute('/hello/HelloService/Hello', mtGet, HelloAction);
  AddRoute('/hello/HelloService/Add', mtGet, AddAction);
  AddRoute('/hello', mtGet, HomeAction);
  AddRoute('/hello/customers', mtGet, CustomersAction);

  // (3) the default action answers everything else
  AddRoute('', mtAny, NotFoundAction, True);
end;

destructor THelloModule.Destroy;
begin
  FConnection.Free;
  inherited;
end;
```

Beachten Sie, dass wir die Verbindung hier nur erzeugen. Geöffnet wird sie erst bei der ersten Query, ein Web-Modul, das immer nur JSON beantwortet, berührt die Datenbank also nie. Der Handler für die neue Seite sieht so aus:

```
procedure THelloModule.CustomersAction(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
var
  Query: TFDQuery;
  Processor: TWebStencilsProcessor;
  City: string;
begin
  City := Request.QueryFields.Values['city'];

  Query := TFDQuery.Create(nil);
  try
    Query.Connection := FConnection;

    // (11) user input only ever travels as a parameter
    if City = '' then
      Query.Open('SELECT id, name, city FROM customers ORDER BY name')
    else
      Query.Open('SELECT id, name, city FROM customers ' +
        'WHERE city = :city ORDER BY name', [City]);

    Processor := TWebStencilsProcessor.Create(nil);
    try
      Processor.InputFileName := TemplateFileName('customers.html');
      Processor.AddVar('info', TPageInfo.Create('Customers'), True);

      // (12) the open query IS the data -- False: the processor
      // must not free it, we do that ourselves
      Processor.AddVar('customers', Query, False);

      Response.ContentType := 'text/html; charset=utf-8';
      Response.Content := Processor.Content;
    finally
      Processor.Free;
    end;
  finally
    Query.Free;
  end;
end;
```

Die Nummerierung setzt die der vorherigen Beiträge fort:

1. Jede Instanz des Web-Moduls erzeugt im Konstruktor ihre eigene `TFDConnection` und gibt sie im Destruktor frei. Da WebBroker nie zwei Anfragen gleichzeitig auf demselben Modul ausführt, teilen sich nie zwei Threads eine Verbindung.
2. Der optionale Query-Parameter `city` filtert die Liste. Er gelangt ausschließlich als Parameter (`:city`) zur Datenbank, niemals als Teil des SQL-Strings. Was auch immer jemand in die URL schreibt, es kann die Anweisung nicht verändern. Ich weiß, dass Sie das wissen. Ich sehe aber immer noch oft genug per String-Verkettung zusammengebautes SQL in Produktivcode, um es jedes einzelne Mal hinzuschreiben.
3. Die geöffnete Query wird direkt an WebStencils übergeben. Es gibt keine Objektliste und keinen Mapping-Code: WebStencils iteriert über die Zeilen eines Datasets und liest die Felder über ihren Namen. Der dritte Parameter ist diesmal `False`, denn die Query gehört uns und wird in unserem eigenen `finally`-Block freigegeben. Übergeben Sie den Besitz an den Prozessor, geben Sie die Query zweimal frei.

Schritt 4: Das Template

Im Template wird aus dem Dataset eine Tabelle. Speichern Sie es als `templates/customers.html`:

```
@LayoutPage layout
<h1>@info.Title</h1>
<table>
  <thead>
    <tr><th>#</th><th>Name</th><th>City</th></tr>
  </thead>
  <tbody>
    @ForEach (var customer in customers) {
    <tr>
      <td>@customer.id</td>
      <td>@customer.name</td>
      <td><a href="/hello/customers?city=@customer.city">@customer.city</a></td>
    </tr>
    }
  </tbody>
</table>
<p><a href="/hello/customers">Show all customers</a> | <a href="/hello">Home</a></p>
```

`@ForEach (var customer in customers)` läuft durch die Zeilen der Query, und `@customer.name` liest das Feld `name` der aktuellen Zeile -- dieselbe Syntax, die wir im vorherigen Beitrag für Objekteigenschaften verwendet haben. Die Feldnamen sind die Spaltennamen aus unserem `SELECT`. Das Template nutzt das Layout aus dem vorherigen Beitrag, die Seite erhält also denselben Rahmen und dieselbe Fußzeile, ohne eine einzige zusätzliche Zeile.

Jede Stadt ist ein Link, der die Liste nach dieser Stadt filtert. Die Kunden aus New York sind genau aus diesem Grund dabei: Ein Filter, der immer nur eine Zeile liefert, beweist nicht viel.

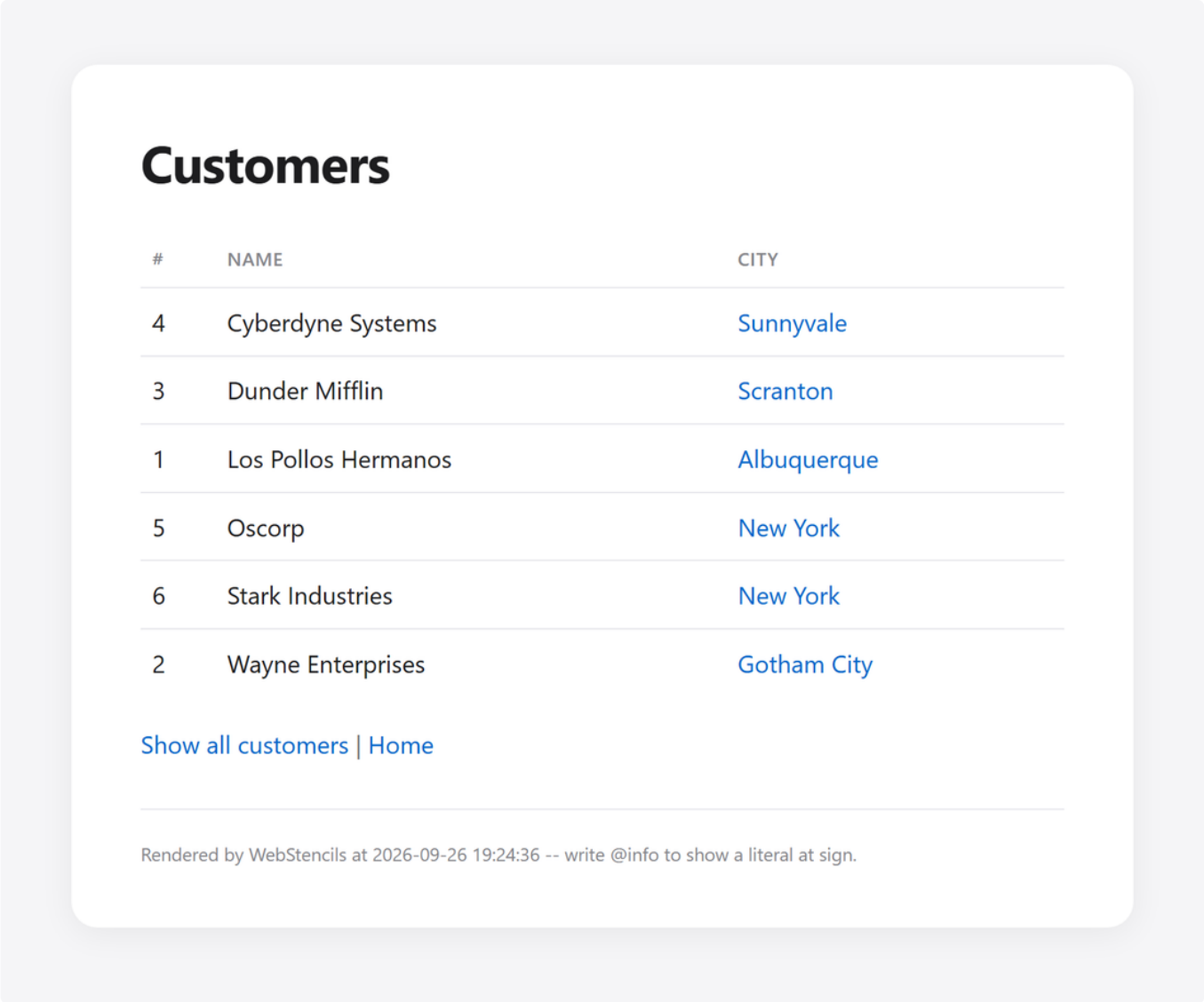
Schritt 5: Starten

Kopieren Sie den Ordner `templates` erneut neben `HelloServer.exe` -- er enthält eine neue Datei -- und starten Sie den Server. Die Konsole zeigt jetzt zuerst den Pfad der Datenbankdatei an.

```
Database: C:\Projects\HelloServer\Win32\Debug\customers.sqlite3
WebBroker server running on port 8080
Try: http://localhost:8080/hello/customers
Press Enter to stop.
```

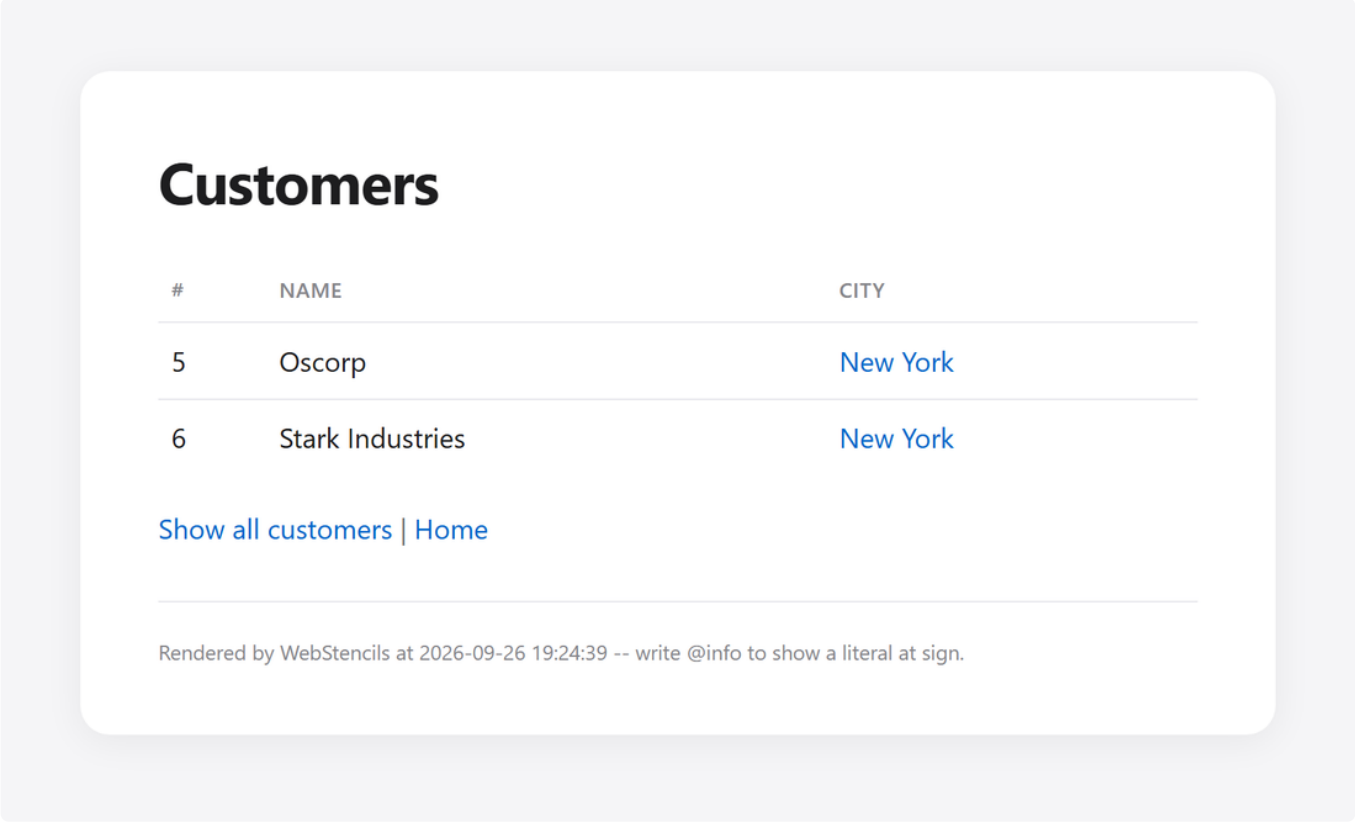
Beim Start legt der Server die Datenbank bei Bedarf an und gibt aus, wo sie liegt.

Öffnen Sie `http://localhost:8080/hello/customers`, und Sie sollten alle sechs Kunden sehen, nach Namen sortiert.



Sechs Zeilen direkt aus einer FireDAC-Query, gerendert mit dem gemeinsamen Layout.

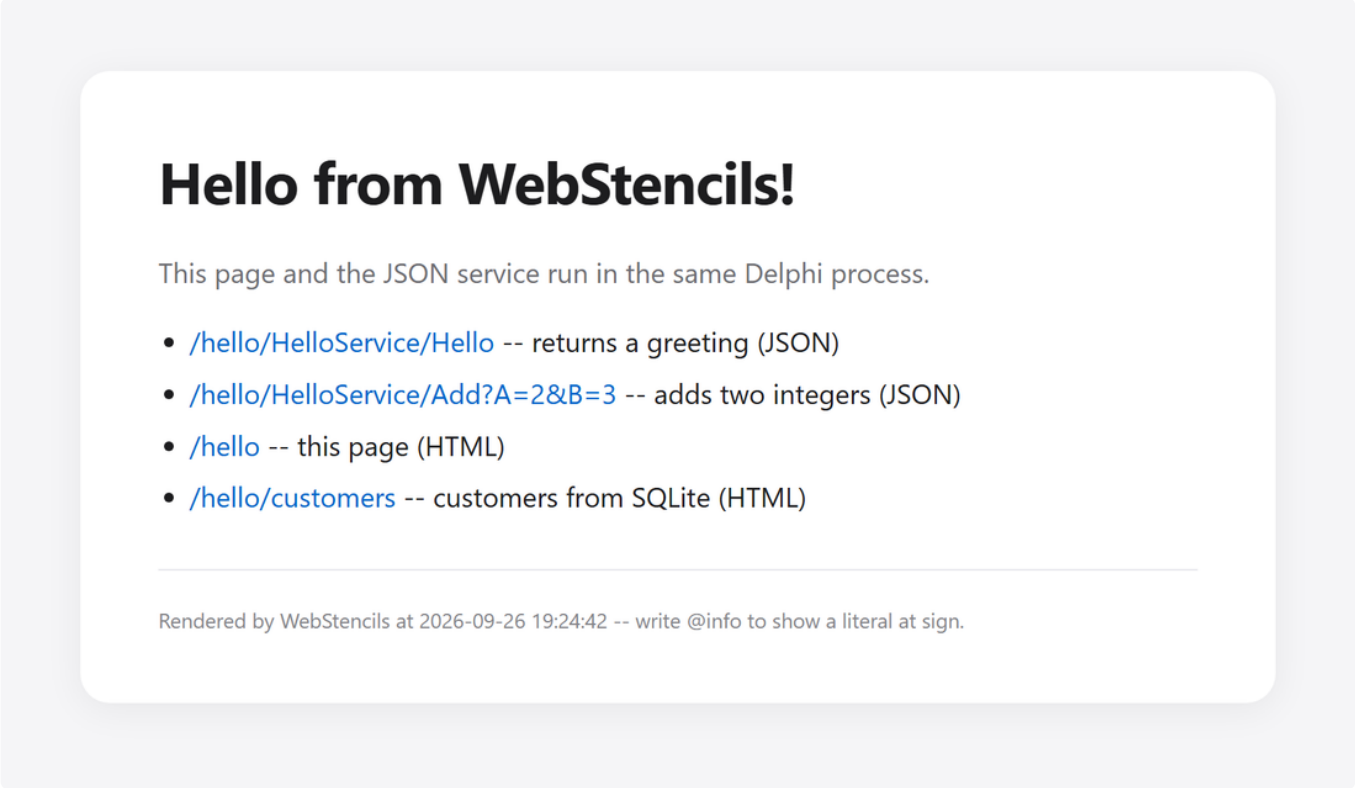
Klicken Sie auf New York, und die Liste schrumpft auf Oscorp und Stark Industries. Schick.



Gefiltert mit <code>?city=New%20York</code>; die Stadt gelangt als Parameter zu SQLite.

Beenden Sie nun den Server und schauen Sie neben die ausführbare Datei: Dort liegt `customers.sqlite3`. Starten Sie den Server erneut, und nichts wird doppelt eingefügt. Löschen Sie die Datei, und der nächste Start legt sie von Grund auf neu an. Genau darum geht es bei dieser Übung: Die Datenbank ist Teil des Codes und nichts, was Sie mitbringen müssen.

Der JSON-Service aus dem ersten Beitrag und die Startseite aus dem zweiten funktionieren natürlich weiterhin. Unser kleiner Server liefert jetzt JSON, HTML aus statischen Daten und datenbankgestütztes HTML aus einer einzigen ausführbaren Datei.



Die Startseite aus dem vorigen Beitrag, jetzt mit einem vierten Eintrag für die Kundenliste.

Wo dieses Beispiel endet

Für ein kleines internes Werkzeug trägt diese Struktur recht weit. Sie sollten aber wissen, wo sie aufhört. `CREATE TABLE IF NOT EXISTS` ist keine Migrationsstrategie: Sobald Sie zum ersten Mal einer bestehenden Datenbank eine Spalte hinzufügen müssen, wollen Sie versionierte Schemaänderungen. Die Seite liest nur Daten; Formulare zum Anlegen oder Bearbeiten von Kunden sind der naheliegende nächste Schritt und brauchen eine ordentliche Validierung. Und SQLite kommt mit vielen Lesern sehr gut zurecht, erlaubt aber immer nur einen Schreiber gleichzeitig, wie die SQLite-Dokumentation zu geeigneten Einsatzgebieten ausführlich erläutert. Für einen schreiblastigen Service mit vielen gleichzeitigen Benutzern ist ein Datenbankserver wie PostgreSQL die bessere Wahl, und mit FireDAC ist der Wechsel im Wesentlichen eine Frage der Verbindungsparameter.

> **Alternativen**
> Dieselbe Seite ist auch mit anderen Stacks ähnlich kurz: Flask mit Jinja-Templates und dem in Python eingebauten sqlite3-Modul oder Express auf Node.js mit einer Template-Engine Ihrer Wahl. In Delphi ist Zeos eine seit Langem etablierte Open-Source-Alternative zu FireDAC für den Datenbankzugriff. Der Reiz der Kombination in diesem Beitrag liegt darin, dass jedes einzelne Teil bereits in Ihrer Delphi-Installation steckt.

Fazit

Wir haben eine WebStencils-Seite aus einer SQLite-Datenbank gespeist, die das Programm selbst anlegt: Datei, Tabelle und Beispieldaten beim ersten Start, und keinerlei Änderungen bei jedem weiteren Start. Drei Details tragen das Beispiel. `LockingMode=Normal` erlaubt jedem Web-Modul, seine eigene Verbindung offen

zu halten, ohne jeden Schreibzugriff auszusperren. Query-Parameter halten Benutzereingaben aus dem SQL heraus. Und WebStencils iteriert direkt über eine FireDAC-Query und liest die Felder über ihren Namen.

> Die beste Beispieldatenbank ist die, die niemand herunterladen muss.

Damit schließen wir die ersten Schritte unserer Serie ab. Wir haben mit einem JSON-Service begonnen, serverseitig gerendertes HTML hinzugefügt und eine Datenbank angebunden -- alles mit WebBroker, WebStencils und FireDAC. Vertraut wird man damit nur, indem man Dinge verändert. Fügen Sie also als Nächstes ein Formular hinzu, das einen Kunden anlegt, und stellen Sie sich vor, welche Möglichkeiten Sie jetzt haben...

Vollständiger Quellcode

Zum Nachschlagen finden Sie hier die vollständigen Dateien, die sich gegenüber dem vorherigen Beitrag geändert haben. Die unveränderten Dateien `HelloPageModel.pas` und `HelloWebModule.dfm` sowie die Templates aus den vorherigen Beiträgen vervollständigen das Projekt.

```
program HelloServer;

{$APPTYPE CONSOLE}

uses
  System.SysUtils,
  Web.WebReq,
  IdHTTPWebBrokerBridge,
  HelloPageModel in 'HelloPageModel.pas',
  CustomerData in 'CustomerData.pas',
  HelloWebModule in 'HelloWebModule.pas' {HelloModule: TWebModule};

const
  PORT = 8080;

var
  Server: TIdHTTPWebBrokerBridge;
begin
  try
    // (0) create the database file, table, and sample data if needed
    CreateDatabase;
    WriteLn('Database: ', DatabaseFileName);

    // (1) tell WebBroker which class answers the requests
    WebRequestHandler.WebModuleClass := THelloModule;

    // (2) the HTTP server: Indy, bridged to WebBroker
    Server := TIdHTTPWebBrokerBridge.Create(nil);
    try
      Server.DefaultPort := PORT;
      Server.Active := True;

      WriteLn('WebBroker server running on port ', PORT);
      WriteLn('Try: http://localhost:8080/hello/customers');
      WriteLn('Press Enter to stop.');
      ReadLn;

      Server.Active := False;
    finally
      Server.Free;
    end;
  except
    on E: Exception do
    begin
      WriteLn(E.ClassName, ': ', E.Message);
      ExitCode := 1;
    end;
  end;
```

```
end.
```

```
unit HelloWebModule;

interface

uses
  System.SysUtils,
  System.Classes,
  System.JSON,
  Web.HTTPApp,
  FireDAC.Comp.Client;

type
  THelloModule = class(TWebModule)
  private
    FConnection: TFDConnection;
    procedure AddRoute(const APathInfo: string; AMethod: TMethodType;
      AHandler: THTTPMethodEvent; ADefault: Boolean = False);
    procedure SendJson(Response: TWebResponse; AStatusCode: Integer;
      AJson: TJSONObject);
    procedure Cors(Sender: TObject; Request: TWebRequest;
      Response: TWebResponse; var Handled: Boolean);
    procedure HelloAction(Sender: TObject; Request: TWebRequest;
      Response: TWebResponse; var Handled: Boolean);
    procedure AddAction(Sender: TObject; Request: TWebRequest;
      Response: TWebResponse; var Handled: Boolean);
    procedure HomeAction(Sender: TObject; Request: TWebRequest;
      Response: TWebResponse; var Handled: Boolean);
    procedure CustomersAction(Sender: TObject; Request: TWebRequest;
      Response: TWebResponse; var Handled: Boolean);
    procedure NotFoundAction(Sender: TObject; Request: TWebRequest;
      Response: TWebResponse; var Handled: Boolean);
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  end;

implementation

uses
  System.IOUtils,
  System.Generics.Collections,
  Web.Stencils,
  HelloPageModel,
  CustomerData;

{$R *.dfm}

function TemplateFileName(const AName: string): string;
begin
  // the templates folder sits next to the executable
  Result := TPath.Combine(
    TPath.Combine(ExtractFilePath(ParamStr(0)), 'templates'), AName);
end;

{ THelloModule }

constructor THelloModule.Create(AOwner: TComponent);
begin
  inherited;

  // (10) every web module instance gets its own connection
  FConnection := CreateConnection;

  // (1) runs before any action -- adds the CORS headers
  BeforeDispatch := Cors;
```

```
  // (2) one action per URL
  AddRoute('/hello/HelloService/Hello', mtGet, HelloAction);
  AddRoute('/hello/HelloService/Add', mtGet, AddAction);
  AddRoute('/hello', mtGet, HomeAction);
  AddRoute('/hello/customers', mtGet, CustomersAction);

  // (3) the default action answers everything else
  AddRoute('', mtAny, NotFoundAction, True);
end;

destructor THelloModule.Destroy;
begin
  FConnection.Free;
  inherited;
end;

procedure THelloModule.AddRoute(const APathInfo: string; AMethod: TMethodType;
  AHandler: THTTPMethodEvent; ADefault: Boolean);
var
  Item: TWebActionItem;
begin
  Item := Actions.Add;
  Item.PathInfo := APathInfo;
  Item.MethodType := AMethod;
  Item.Default := ADefault;
  Item.OnAction := AHandler;
end;

procedure THelloModule.SendJson(Response: TWebResponse; AStatusCode: Integer;
  AJson: TJSONObject);
begin
  try
    Response.StatusCode := AStatusCode;
    Response.ContentType := 'application/json; charset=utf-8';
    Response.Content := AJson.ToJSON;
  finally
    AJson.Free;
  end;
end;

procedure THelloModule.Cors(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
begin
  // (4) CORS: '*' allows EVERY origin, which effectively switches the
  // browser's cross-origin protection off for this server. Fine for
  // development; in production, replace '*' with the origin of your
  // web application, e.g. 'https://app.example.com'.
  Response.SetCustomHeader('Access-Control-Allow-Origin', '*');

  // (5) answer the browser's preflight request right here
  if SameText(Request.Method, 'OPTIONS') then
  begin
    Response.SetCustomHeader('Access-Control-Allow-Methods', 'GET, OPTIONS');
    Response.SetCustomHeader('Access-Control-Allow-Headers', 'Content-Type');
    Response.StatusCode := 204;
    Handled := True;
  end;
end;

procedure THelloModule.HelloAction(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
begin
  SendJson(Response, 200,
    TJSONObject.Create.AddPair('value', 'Hello from WebBroker!'));
end;

procedure THelloModule.AddAction(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
```

```
var
  A, B: Integer;
begin
  // (6) nobody converts the parameters for us -- we do it ourselves
  if TryStrToInt(Request.QueryFields.Values['A'], A) and
    TryStrToInt(Request.QueryFields.Values['B'], B) then
    SendJson(Response, 200,
      TJSONObject.Create.AddPair('value', TJSONNumber.Create(A + B)))
  else
    SendJson(Response, 400,
      TJSONObject.Create.AddPair('error', 'A and B must be integers'));
end;

procedure THelloModule.HomeAction(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
var
  Processor: TWebStencilsProcessor;
  Endpoints: TObjectList<TEndpoint>;
begin
  Processor := TWebStencilsProcessor.Create(nil);
  try
    // (7) the page to render; it names its own layout
    Processor.InputFileName := TemplateFileName('home.html');

    // (8) the data -- True hands ownership to the processor
    Processor.AddVar('info', TPageInfo.Create('Hello from WebStencils!'), True);

    Endpoints := TObjectList<TEndpoint>.Create;
    Endpoints.Add(TEndpoint.Create('/hello/HelloService/Hello',
      'returns a greeting', False));
    Endpoints.Add(TEndpoint.Create('/hello/HelloService/Add?A=2&B=3',
      'adds two integers', False));
    Endpoints.Add(TEndpoint.Create('/hello', 'this page', True));
    Endpoints.Add(TEndpoint.Create('/hello/customers',
      'customers from SQLite', True));
    Processor.AddVar('endpoints', Endpoints, True);

    // (9) Content runs the template engine and returns the HTML
    Response.ContentType := 'text/html; charset=utf-8';
    Response.Content := Processor.Content;
  finally
    Processor.Free;
  end;
end;

procedure THelloModule.CustomersAction(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
var
  Query: TFDQuery;
  Processor: TWebStencilsProcessor;
  City: string;
begin
  City := Request.QueryFields.Values['city'];

  Query := TFDQuery.Create(nil);
  try
    Query.Connection := FConnection;

    // (11) user input only ever travels as a parameter
    if City = '' then
      Query.Open('SELECT id, name, city FROM customers ORDER BY name')
    else
      Query.Open('SELECT id, name, city FROM customers ' +
        'WHERE city = :city ORDER BY name', [City]);

    Processor := TWebStencilsProcessor.Create(nil);
    try
      Processor.InputFileName := TemplateFileName('customers.html');
      Processor.AddVar('info', TPageInfo.Create('Customers'), True);
```

```
      // (12) the open query IS the data -- False: the processor
      // must not free it, we do that ourselves
      Processor.AddVar('customers', Query, False);

      Response.ContentType := 'text/html; charset=utf-8';
      Response.Content := Processor.Content;
    finally
      Processor.Free;
    end;
  finally
    Query.Free;
  end;
end;

procedure THelloModule.NotFoundAction(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
begin
  SendJson(Response, 404,
    TJSONObject.Create.AddPair('error', 'Not found: ' + Request.PathInfo));
end;

end.
```

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 —