NEWS · SEPTEMBER 23, 2026

Ein TMS-XData-Server in Delphi in fünf Schritten: ohne Formular, ohne Wizard

Ein vollständiger TMS-XData-REST-Server für Delphi in drei kleinen Quelldateien und fünf Schritten -- eine Konsolenanwendung ohne Formular, ohne Wizard und ohne Datenbank, inklusive der URL-Reservierung und der CORS-Zeile, die in Tutorials gerne fehlen.

By Dr. Holger Flick

Kunden erzählen mir ständig, dass ein TMS-XData-Server so viele Schritte braucht. Ich höre es in Schulungen, ich lese es in E-Mails, und ich verstehe, woher der Eindruck kommt. Die meisten Einführungen beginnen mit Datei, Neu, einem Wizard und einem Datenmodul mit ein paar nicht-visuellen Komponenten. So kommen Sie schnell zu einem laufenden Server, aber es bleibt verborgen, wie wenig tatsächlich dahintersteckt. Die Behauptung ist schlicht falsch. Solange keine Datenbank im Spiel ist, ist ein XData-Server sehr einfach zu bauen.

Drehen wir den Spieß also um. In diesem Beitrag bauen wir einen vollständigen REST-Server mit TMS XData,

dem kommerziellen REST/JSON-Server-Framework für Delphi von TMS Software, als schlichte Konsolenanwendung. Es gibt kein Formular, kein Datenmodul, keinen Wizard und keine Datenbank. Jede Zeile wird von Hand getippt, und jede Zeile wird erklärt.

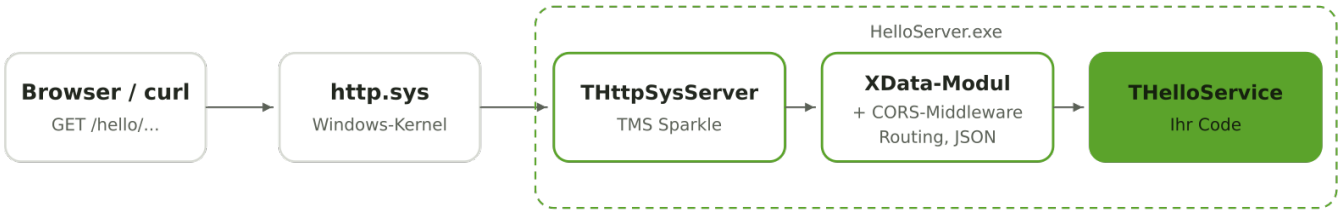
Am Ende haben Sie drei kleine Quelldateien, einen Webservice, der `Hello from TMS XData!` als JSON liefert, und eine ehrliche Zählung der nötigen Schritte -- einschließlich der beiden, die nichts mit Pascal zu tun haben und an denen die meisten hängen bleiben.

Was wir bauen

Bevor wir Code schreiben, lohnt sich ein Blick auf die beteiligten Teile, denn es sind nur vier. XData spricht nicht selbst mit dem Netzwerk. Es setzt auf TMS Sparkle auf, dem HTTP-Framework desselben Herstellers, und

Sparkles `THttpSysServer` setzt wiederum auf http.sys auf, dem HTTP-Stack, der Teil des Windows-Kernels ist.

Die Sparkle-Dokumentation beschreibt diese Architektur im Detail.



Der Weg einer Anfrage: Windows reicht sie an Sparkle weiter, die Middleware ergänzt die CORS-Header, XData ruft Ihre Methode auf

Lesen Sie das Diagramm von links nach rechts und achten Sie darauf, wo Ihr eigener Code lebt: Es ist nur der Kasten ganz rechts. Windows nimmt die Verbindung an, Sparkle empfängt die Anfrage, das XData-Modul findet die passende Methode und wandelt das Ergebnis in JSON um. Selbst schreiben wir lediglich ein Delphi-Interface, eine Klasse, die es implementiert, und ein Dutzend Zeilen, die die Kästen zusammenstecken.

Daraus ergeben sich die fünf Schritte: den Vertrag definieren, ihn implementieren, das Serverprogramm schreiben, die URL in Windows reservieren und den Service aufrufen. Sie brauchen Delphi und ein installiertes TMS XData, sonst nichts.

Schritt 1: Den Service-Vertrag definieren

In XData beginnt alles mit einem Interface, dem sogenannten Service Contract. Es listet die Operationen auf, die der Server anbietet, und die Dokumentation zu Service Operations nennt die Anforderungen: Es stammt

von `IInvokable` ab, besitzt eine GUID, trägt das Attribut `[ServiceContract]` und wird mit `RegisterServiceType` registriert. Legen Sie eine neue Konsolenanwendung an, speichern Sie sie als `HelloServer` und fügen Sie eine Unit namens `HelloService.pas` hinzu:

```
unit HelloService;

interface

uses
  XData.Service.Common;

type
  [ServiceContract]
  IHelloService = interface(IInvokable)
    ['{6F3A1C52-9B7E-4D0A-8E21-5C4B7A9D3E10}']
    // (1) GET instead of the default POST, so a browser can call it
    [HttpGet] function Hello: string;
    // (2) parameters of a GET operation arrive in the query string
    [HttpGet] function Add(A, B: Integer): Integer;
  end;

implementation

initialization
  // (3) make the contract known to the XData model
  RegisterServiceType(TypeInfo(IHelloService));

end.
```

Die Nummern in den Kommentaren entsprechen den folgenden Anmerkungen:

1. Standardmäßig beantwortet XData eine Service-Operation per `POST`. Das Attribut `[HttpGet]` stellt die Methode auf `GET` um, sodass wir sie testen können, indem wir die URL in einen Browser tippen.
2. `Add` zeigt, dass Parameter ohne Zusatzaufwand funktionieren. Bei einer `GET`-Operation liest XData sie standardmäßig aus dem Query-String, der Aufruf lautet also `Add?A=2&B=3`.
3. `RegisterServiceType` fügt das Interface dem XData-Modell hinzu. Ohne diese Zeile kann XData nicht wissen, dass der Vertrag existiert. Erzeugen Sie in der IDE mit `Strg+Umschalt+G` Ihre eigene GUID, statt meine zu kopieren.

Sie fragen sich vielleicht, wo die URL der Operation festgelegt wird. Nirgends, und genau das ist der Punkt. XData leitet sie aus den Namen ab: der Interface-Name ohne das führende `I`, dann der Methodenname. Aus `IHelloService.Hello` wird `/HelloService/Hello`. Wenn Sie andere Pfade bevorzugen, gibt Ihnen das auf derselben Dokumentationsseite beschriebene Attribut `[Route]` die volle Kontrolle. Für einen minimalen Server ist der Standard völlig in Ordnung.

Schritt 2: Den Service implementieren

Der Vertrag braucht eine Klasse, die die eigentliche Arbeit erledigt. Fügen Sie eine zweite Unit namens `HelloServiceImpl.pas` hinzu. Ich halte Vertrag und Implementierung in getrennten Units, weil sich die Interface-Unit später mit einem Delphi-Client teilen lässt, während die Implementierung auf dem Server bleibt. Der XData-Wizard schlägt dieselbe Aufteilung standardmäßig vor, aus demselben Grund.

```
unit HelloServiceImpl;

interface

uses
  XData.Server.Module,
  XData.Service.Common,
  HelloService;

type
  [ServiceImplementation]
  THelloService = class(TInterfacedObject, IHelloService)
  private
    function Hello: string;
    function Add(A, B: Integer): Integer;
  end;

implementation

{ THelloService }

function THelloService.Hello: string;
begin
  Result := 'Hello from TMS XData!';
end;

function THelloService.Add(A, B: Integer): Integer;
begin
  Result := A + B;
end;

initialization
  RegisterServiceType(THelloService);

end.
```

Hier gibt es wirklich keine Magie. Die Klasse implementiert das Interface, trägt das Attribut `[ServiceImplementation]` und wird im `initialization`-Abschnitt registriert, diesmal durch Übergabe der Klasse selbst. Beachten Sie, was fehlt: Es gibt keinen JSON-Code, kein Parsen von Query-Parametern und keine Behandlung von HTTP-Statuscodes. Die Methoden nehmen Delphi-Typen entgegen und geben Delphi-Typen zurück. Um den Rest kümmert sich XData.

Schritt 3: Das Serverprogramm schreiben

Jetzt stecken wir die Teile in der Projektdatei zusammen. Das ist der Teil, den normalerweise ein Wizard oder ein paar Design-Time-Komponenten für Sie erledigen, und wie Sie gleich sehen werden, ist es nicht viel. Ersetzen Sie den Inhalt von `HelloServer.dpr` durch Folgendes:

```
program HelloServer;

{$APPTYPE CONSOLE}

uses
  System.SysUtils,
  Sparkle.HttpSys.Server,
  Sparkle.Middleware.Cors,
  XData.Server.Module,
  HelloService in 'HelloService.pas',
  HelloServiceImpl in 'HelloServiceImpl.pas';

const
  // '+' means: every host name on this machine, port 2001, path /hello
  BASE_URL = 'http://+:2001/hello';
```

```
var
  Server: THttpSysServer;
  Module: TXDataServerModule;
begin
  try
    // (1) the HTTP server, built on the Windows http.sys stack
    Server := THttpSysServer.Create;
    try
      // (2) the XData module -- no database, no connection pool
      Module := TXDataServerModule.Create(BASE_URL);

      // (3) 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'.
      Module.AddMiddleware(TCorsMiddleware.Create('*'));

      // (4) hand the module to the server and start listening
      Server.AddModule(Module);
      Server.Start;

      WriteLn('XData server running at ', BASE_URL);
      WriteLn('Try: http://localhost:2001/hello/HelloService/Hello');
      WriteLn('Press Enter to stop.');
      ReadLn;

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

Auch hier entsprechen die Nummern den Kommentaren im Code:

1. `THttpSysServer` aus der Unit `Sparkle.HttpSys.Server` ist der HTTP-Server. Er ist keine Komponente auf einem Formular, sondern ein einfaches Objekt, das wir selbst erzeugen und freigeben.
2. `TXDataServerModule` ist, in den Worten der XData-Dokumentation, die Klasse, die den XData-Server

   implementiert. Die meisten Beispiele übergeben als zweites Argument einen Datenbank-Connection-Pool, weil XData häufig zusammen mit dem ORM TMS Aurelius eingesetzt wird. Wir brauchen keine Datenbank, und es gibt eine Konstruktor-Überladung, die nur die Basis-URL entgegennimmt.
3. Die CORS-Middleware. Sie bekommt weiter unten einen eigenen Abschnitt, denn wer sie weglässt, erhält den verwirrendsten Fehler der ganzen Übung.
4. `AddModule` übergibt das Modul an den Server, `Start` beginnt mit dem Lauschen. Die offiziellen Beispiele geben am Ende nur den Server frei, nie das Modul, und wir folgen diesem Muster.

Beide Units stehen in der `uses`-Klausel des Projekts, und das ist wichtiger, als es aussieht. Die Services registrieren sich in ihren `initialization`-Abschnitten selbst, und die laufen nur, wenn die Units Teil des Programms sind. Nirgends im Servercode taucht `THelloService` namentlich auf. Die Registrierung ist die Verbindung.

Kompilieren Sie das Projekt. Ist XData installiert, findet der Compiler die Sparkle- und XData-Units über den Bibliothekspfad, es gibt also nichts zu konfigurieren. Starten Sie das Programm aber noch nicht...

Schritt 4: Die URL in Windows reservieren

Diesen Schritt lassen Tutorials gerne aus, und er hat nichts mit Delphi zu tun. Weil http.sys Teil des Betriebssystems ist, möchte Windows wissen, wer auf welcher URL lauschen darf. Die Sparkle-Dokumentation ist da eindeutig: Die URL, auf der Ihr Server lauscht, muss reserviert sein, sonst kann die Anwendung nicht auf

Anfragen antworten.

Öffnen Sie eine Eingabeaufforderung **als Administrator** und führen Sie einmalig aus:

```
netsh http add urlacl url=http://+:2001/hello/ user=%USERDOMAIN%\%USERNAME%
```

Die URL entspricht unserer Konstante `BASE_URL`, mit abschließendem Schrägstrich. Die Reservierung wird in Windows gespeichert und übersteht Neustarts. Sie erledigen das also einmal pro Rechner und URL, nicht bei jedem Start des Servers. Microsoft dokumentiert den Befehl unter add urlacl. Wenn Sie lieber klicken als tippen:

Sparkle liefert ein kleines Werkzeug namens `TMSHttpConfig` mit, das dasselbe über eine Oberfläche erledigt, und mit der Klasse `THttpSysServerConfig` geht es auch aus Delphi-Code, etwa in einem Installer.

> **Erst die URL reservieren, dann am Code zweifeln**
>
> Ohne Reservierung kompiliert das Programm problemlos und scheitert dann sofort in `Server.Start`. Unser `except`-Block gibt Folgendes aus, und der Prozess endet mit Exit-Code 1: ``text EHttpSysSessionException: Could not add the following Url to the server. Be sure that you have reserved the Url in Http.Sys with proper permissions.  Url: http://+:2001/hello Error: Access is denied `` Die letzte Zeile ist die Fehlermeldung von Windows und erscheint deshalb in der Sprache Ihrer Windows-Installation, auf einem deutschen System also als „Zugriff verweigert“. Die Reservierung gehört auf jeden Rechner, auf dem der Server läuft, also auch auf den Produktivserver und den PC Ihrer Kollegin.

Die Versuchung ist groß, den Server stattdessen einfach mit Administratorrechten zu starten, und das funktioniert tatsächlich: Ein Prozess mit erhöhten Rechten darf die URL auch ohne Reservierung registrieren. Bitte machen Sie das nicht zur Gewohnheit. Ein Webservice, der Anfragen aus dem Netzwerk annimmt, ist das letzte Programm, das mit erhöhten Rechten laufen sollte, und die Reservierung ist eine einzeilige Alternative.

Schritt 5: Starten und den Service aufrufen

Starten Sie `HelloServer.exe`, und die Konsole zeigt Ihnen, wo der Server lauscht. Öffnen Sie einen Browser und geben Sie `http://localhost:2001/hello/HelloService/Hello` ein, oder verwenden Sie `curl` in einem zweiten Konsolenfenster:

```
curl http://localhost:2001/hello/HelloService/Hello
```

Die Antwort ist ein kleines JSON-Dokument:

```
{
    "value": "Hello from TMS XData!"
}
```

XData verpackt einen einfachen Rückgabewert in ein Objekt mit der Eigenschaft `value`; das ist das dokumentierte Format für Service-Ergebnisse. Die zweite Operation beweist, dass Parameter genauso funktionieren:

```
curl "http://localhost:2001/hello/HelloService/Add?A=2&B=3"
```

Hier lautet die Antwort `"value": 5`. Wir haben zwei `Integer`-Parameter in einem Delphi-Interface deklariert, und XData hat sie aus dem Query-String geholt, konvertiert, die Methode aufgerufen und das Ergebnis zurückkonvertiert. Schick.

Das ist ein funktionierender REST-Server. Drei Quelldateien, ein Windows-Befehl.

Warum die CORS-Zeile dort steht

Der Server würde auch ohne die Middleware-Zeile laufen, und `curl` würde den Unterschied nie bemerken. Eine Webanwendung schon. Browser setzen die Same-Origin-Policy durch: JavaScript, das von einem Origin geladen wurde, etwa `http://localhost:3000`, darf Antworten eines anderen Origins, etwa `http://localhost:2001`, nicht lesen, es sei denn, dieser Server erlaubt es ausdrücklich. Der Erlaubnismechanismus heißt CORS, kurz für Cross-Origin Resource Sharing, und funktioniert über Antwort-Header.

Beachten Sie, dass schon ein anderer Port ein anderer Origin ist. Sobald Sie also ein Web-Frontend bauen -- mit TMS WEB Core, React oder reinem JavaScript, das spielt keine Rolle -- und es diesen Server aufrufen lassen, blockiert der Browser die Antwort und schreibt einen CORS-Fehler in seine Konsole. Der Service selbst ist in Ordnung. Es ist der fehlende Header, der Sie aufhält. Genau dazu bekomme ich ständig E-Mails von Kunden: Der TMS-WEB-Core-Client verbindet sich nicht mit dem Server, während derselbe Service aus ihrer VCL-Anwendung und per URL-Eingabe im Browser einwandfrei funktioniert. Jedes einzelne Mal lautet die Antwort: die fehlende CORS-Middleware. Diesem Fehler hinterherzujagen ist verwirrend, gerade weil jeder Test außerhalb des Browsers gelingt.

`TCorsMiddleware` aus der Unit `Sparkle.Middleware.Cors` ergänzt die nötigen Header und beantwortet laut dem Artikel von TMS zu diesem Thema auch die Preflight-Anfragen, die ein Browser vor Methoden wie `DELETE` schickt. Die Middleware-Dokumentation führt die Konstruktor-Überladungen auf: Der erste Parameter ist der erlaubte Origin, optionale weitere legen die erlaubten Methoden und das Max-Age fest.

> **Das Sternchen ist eine Entwicklungseinstellung**
>
> `TCorsMiddleware.Create('*')` sendet `Access-Control-Allow-Origin: *` und erlaubt damit **jeder** Website, Ihren Server aus einem Browser heraus aufzurufen. Faktisch schaltet das den Schutz ab. Während der Entwicklung wollen Sie genau das, und für eine wirklich öffentliche, nur lesende API kann es richtig sein. In allen anderen Fällen übergeben Sie den echten Origin Ihrer Webanwendung, zum Beispiel `TCorsMiddleware.Create('https://app.example.com')`. Deshalb steht die Warnung direkt im Quellcode: Kommentare überleben Copy-and-paste, Blogbeiträge nicht.

Noch etwas zur Einordnung: CORS ist eine Regel, an die sich Browser halten. Es authentifiziert niemanden, und `curl` ignoriert es vollständig. Es ersetzt keine ordentliche Authentifizierung, die XData über weitere Middleware wie JWT unterstützt. Das ist ein Thema für einen anderen Beitrag.

Wo dieser minimale Server endet

Ich habe bewusst das absolute Minimum gezeigt, und Sie sollten wissen, was ich weggelassen habe. Es gibt kein HTTPS, keine Authentifizierung, kein Logging und keine Datenbank. Der Server ist eine Konsolenanwendung; in Produktion würden Sie dieselben wenigen Zeilen typischerweise in einem Windows-Dienst unterbringen. All das baut auf der Struktur auf, die Sie gerade gesehen haben, und nichts davon ändert die drei Dateien grundlegend.

Außerdem ist `THttpSysServer` an Windows gebunden, weil http.sys es ist. Sparkle bietet weitere Möglichkeiten, ein Modul zu hosten, darunter einen Indy-basierten Server, der in derselben Server-Dokumentation beschrieben ist, falls Sie woanders laufen müssen.

Sie fragen sich vielleicht, ob ich gegen den Wizard und die Design-Time-Komponenten argumentiere. Das tue ich nicht. Der Wizard erzeugt genau die Art von Service-Units, die wir gerade getippt haben, und die Komponenten sind ein bequemer Weg, einen Server zu konfigurieren. Meiner Meinung nach sollten Sie den Server aber mindestens einmal von Hand schreiben. Wenn etwas nicht funktioniert, wissen Sie dann, in welchen der vier Kästen im Diagramm Sie schauen müssen, und dieses Wissen macht sich schnell bezahlt. Das Gegenargument ist genauso berechtigt: Wenn Ihr Team zehn Server pflegt, schlägt generierter, einheitlich strukturierter Code handgeschriebene Individualität. Nutzen Sie den Wizard für die Routine und den manuellen Weg für das Verständnis.

> **Alternativen**
>
> TMS XData ist ein kommerzielles Produkt. Wenn Sie mit Open Source bei Delphi bleiben möchten, schauen Sie sich mORMot 2, DelphiMVCFramework und Horse an; alle drei sind in der Delphi-Community gut bekannt. Außerhalb von Delphi ist ein Service dieser Größe mit Express auf Node.js oder FastAPI in Python ähnlich klein. Warum XData meine persönliche Wahl ist, habe ich in einem früheren Beitrag dargelegt; was für Sie passt, hängt von Ihrem Team und dem Code ab, den Sie bereits haben.

Fazit

Wir wollten herausfinden, wie wenige Schritte ein Delphi-REST-Server wirklich braucht, und die Zählung ergibt fünf: ein Vertrag, eine Implementierung, ein Serverprogramm, eine URL-Reservierung und eine Anfrage. Nur drei davon haben mit Pascal zu tun.

Wenn Sie sich nur wenige Dinge merken, dann diese. Der Service ist ein gewöhnliches Delphi-Interface, und XData leitet URLs und JSON daraus ab, es gibt also keinen Serialisierungscode zu schreiben. Die Units registrieren sich selbst und müssen deshalb in der `uses`-Klausel des Projekts stehen. Die URL-Reservierung ist eine Windows-Anforderung und das Erste, was Sie prüfen sollten, wenn ein Server nicht erreichbar ist. Und die CORS-Middleware entscheidet, ob ein browserbasierter Client Ihren Server überhaupt nutzen kann -- wobei das Sternchen eine Bequemlichkeit für die Entwicklung ist, die Sie vor der Auslieferung ersetzen sollten.

> Ein Webservice ist ein Interface, eine Klasse und ein Dutzend Zeilen, um beide zu hosten. Alles andere ist Konfiguration.

Der vollständige Quellcode besteht aus den drei Listings oben; kopieren Sie sie in einen Ordner, öffnen Sie `HelloServer.dpr`, und es kann losgehen. Vertraut wird man damit nur, indem man Dinge verändert. Fügen Sie also eine dritte Methode hinzu, geben Sie ein Objekt statt eines Strings zurück und schauen Sie sich an, was zurückkommt. Stellen Sie sich vor, welche Möglichkeiten Sie jetzt haben...

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 —