NEWS · SEPTEMBER 23, 2026

A TMS XData Server in Delphi in Five Steps: No Form, No Wizard

A complete TMS XData REST server for Delphi in three small source files and five steps -- a console application with no form, no wizard, and no database, plus the URL reservation and the CORS line that tutorials tend to skip.

By Dr. Holger Flick

Customers tell me all the time that building a TMS XData server takes so many steps. I hear it in trainings, I read it in emails, and I understand where it comes from. Most introductions start with File, New, a wizard, and a data module with a few non-visual components on it. That gets you a running server quickly, but it also hides how little there actually is to it. The claim is simply not true. As long as there is no database involved, an XData server is very easy to build.

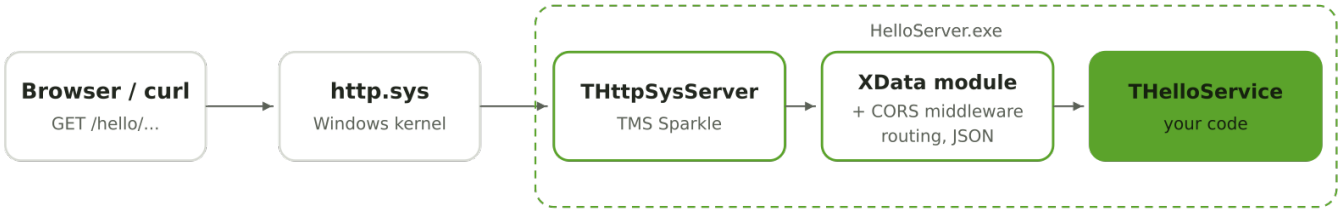
So let's do it the other way around. In this post, we will build a complete REST server with TMS XData, the commercial REST/JSON server framework for Delphi from TMS Software, as a plain console application. There is no form, no data module, no wizard, and no database. Every line is typed by hand, and every line gets explained.

You will end up with three small source files, a web service that answers `Hello from TMS XData!` as JSON, and an honest count of the steps it takes -- including the two that have nothing to do with Pascal and trip people up the most.

What we are building

Before we write any code, it helps to know which parts are involved, because there are only four of them. XData does not talk to the network itself. It sits on top of TMS Sparkle, the HTTP framework from the same vendor, and Sparkle's `THttpSysServer` in turn sits on top of http.sys, the HTTP stack that is part of the Windows kernel.

The Sparkle documentation describes this architecture in detail.



The path of one request: Windows hands it to Sparkle, the middleware adds the CORS headers, XData calls your method

Read the diagram from left to right and note where your own code lives: it is only the box on the far right. Windows accepts the connection, Sparkle receives the request, the XData module finds the matching method and converts the result to JSON. The only thing we write ourselves is a Delphi interface, a class that implements it, and a dozen lines that plug the boxes together.

That gives us the five steps: define the contract, implement it, write the server program, reserve the URL in Windows, and call the service. You need Delphi and an installed copy of TMS XData; nothing else.

Step 1: Define the service contract

Everything in XData starts with an interface, the so-called service contract. It lists the operations that the server offers, and the documentation on service operations spells out what it needs: it descends from `IInvokable`, it has a GUID, it carries the `[ServiceContract]` attribute, and it is registered with `RegisterServiceType`. Create a new console application, save it as `HelloServer`, and add a unit named `HelloService.pas`:

```
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.
```

The numbers in the comments match the following notes:

1. By default, XData answers a service operation on `POST`. The `[HttpGet]` attribute switches the method to `GET`, which means we can test it by typing the URL into a browser.
2. `Add` is there to show that parameters work without any extra effort. For a `GET` operation, XData reads them from the query string by default, so the call will be `Add?A=2&B=3`.
3. `RegisterServiceType` adds the interface to the XData model. Without this line, XData has no way of knowing that the contract exists. Generate your own GUID with `Ctrl+Shift+G` in the IDE instead of copying mine.

You might wonder where the URL of the operation is defined. It is not, and that is the point. XData derives it from the names: the interface name without the leading `I`, then the method name. `IHelloService.Hello` becomes `/HelloService/Hello`. If you prefer different paths, the `[Route]` attribute described on the same documentation page gives you full control. For a minimal server, the default is perfectly fine.

Step 2: Implement the service

The contract needs a class that does the actual work. Add a second unit named `HelloServiceImpl.pas`. I keep the contract and the implementation in separate units because the interface unit can be shared with a Delphi client later, while the implementation stays on the server. The XData wizard suggests the same split by default, for the same reason.

```
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.
```

There is really no magic here. The class implements the interface, carries the `[ServiceImplementation]` attribute, and is registered in the `initialization` section, this time by passing the class itself. Note what is missing: there is no JSON code, no parsing of query parameters, and no HTTP status handling. The methods take Delphi types and return Delphi types. XData takes care of the rest.

Step 3: Write the server program

Now we plug the pieces together in the project file. This is the part a wizard or a set of design-time components would normally do for you, and as you are about to see, it is not much. Replace the content of `HelloServer.dpr` with the following:

```
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.
```

Again, the numbers match the comments in the code:

1. `THttpSysServer` from the unit `Sparkle.HttpSys.Server` is the HTTP server. It is not a component on a form; it is a plain object that we create and free ourselves.
2. `TXDataServerModule` is, in the words of the XData documentation, the class that implements the XData server. Most examples pass a database connection pool as the second argument, because XData is often used together with the TMS Aurelius ORM. We do not need a database, and there is a constructor overload that takes only the base URL.
3. The CORS middleware. It gets its own section below, because leaving it out produces the most confusing error of the entire exercise.
4. `AddModule` hands the module to the server, `Start` begins listening. The official examples free only the server at the end, never the module, and we follow that pattern.

Both units appear in the `uses` clause of the project, and that matters more than it seems. The services register themselves in their `initialization` sections, and those only run if the units are part of the program. Nowhere does the server code mention `THelloService` by name. The registration is the connection.

Compile the project. With XData installed, the compiler finds the Sparkle and XData units through the library path, so there is nothing to configure. Do not run it yet, though...

Step 4: Reserve the URL in Windows

This is the step that tutorials tend to skip, and it has nothing to do with Delphi. Because http.sys is part of the operating system, Windows wants to know who is allowed to listen on which URL. The Sparkle documentation is explicit about it: the URL your server listens on must be reserved, otherwise the application will not be able to respond to requests.

Open a command prompt **as administrator** and run this once:

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

The URL matches our `BASE_URL` constant, with a trailing slash. The reservation is stored in Windows and survives reboots, so you do this one time per machine and URL, not every time you start the server. Microsoft documents the command under add urlacl. If you prefer clicking over typing, Sparkle ships with a small tool called `TMSHttpConfig` that does the same through a user interface, and the `THttpSysServerConfig` class lets you do it from Delphi code, for example in an installer.

> **Reserve the URL before you blame your code**
> Without the reservation, the program compiles fine and then fails immediately in `Server.Start`. Our `except` block prints this and the process exits with 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 `` The last line is the Windows error text, so it appears in the language of your Windows installation. The reservation belongs on every machine the server runs on, which includes the production server and the PC of your colleague.

You might be tempted to simply run the server with administrator rights instead, and it does work: an elevated process may register the URL without a reservation. Please do not make that a habit. A web service that accepts requests from the network is the last program that should run with elevated privileges, and the reservation is a one-line alternative.

Step 5: Run it and call the service

Start `HelloServer.exe` and the console tells you where it is listening. Open a browser and enter `http://localhost:2001/hello/HelloService/Hello`, or use `curl` from a second console window:

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

The response is a small JSON document:

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

XData wraps a simple return value in an object with a `value` property; that is the documented format for service results. The second operation proves that parameters work the same way:

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

This one answers with `"value": 5`. We declared two `Integer` parameters in a Delphi interface, and XData took them out of the query string, converted them, called the method, and converted the result back. Neat.

That is a working REST server. Three source files, one Windows command.

Why the CORS line is there

The server would run without the middleware line, and `curl` would never notice the difference. A web application would. Browsers enforce the same-origin policy: JavaScript that was loaded from one origin, say `http://localhost:3000`, may not read responses from another origin, say `http://localhost:2001`, unless that server explicitly permits it. The permission mechanism is called CORS, short for Cross-Origin Resource Sharing, and it works through response headers.

Note that a different port is already a different origin. So the moment you build a web front end -- with TMS WEB Core, React, plain JavaScript, it does not matter -- and let it call this server, the browser blocks the response and prints a CORS error to its console. The service itself is fine. It is the missing header that stops you. I get emails from customers all the time about exactly this: the TMS WEB Core client will not connect to the server, while the same service works flawlessly from their VCL application and when they enter the URL in the browser. Every single time, the answer is the missing CORS middleware. It is a confusing error to chase, precisely because every test outside the browser succeeds.

`TCorsMiddleware` from the unit `Sparkle.Middleware.Cors` adds the required headers and, according to TMS's own article on the topic, also answers the preflight requests a browser sends ahead of methods like `DELETE`.

The middleware documentation lists the constructor overloads: the first parameter is the allowed origin, optional further ones set the allowed methods and the max age.

> **The asterisk is a development setting**
>
> `TCorsMiddleware.Create('*')` sends `Access-Control-Allow-Origin: *`, which permits **every** website to call your server from a browser. In effect, it turns the protection off. That is what you want while developing, and it can be right for a truly public, read-only API. For everything else, pass the real origin of your web application instead, for example `TCorsMiddleware.Create('https://app.example.com')`. That is why the warning sits right in the source code: comments survive copy and paste, blog posts do not.

One more thing to keep in perspective: CORS is a rule that browsers follow. It does not authenticate anybody, and `curl` ignores it entirely. It is no replacement for proper authentication, which XData supports through further middleware such as JWT. That is a topic for another post.

Where this minimal server ends

I presented the bare minimum on purpose, and you should know what I left out. There is no HTTPS, no authentication, no logging, and no database. The server is a console application; in production you would typically host the same few lines in a Windows service. All of that builds on the structure you have just seen, and none of it changes the three files fundamentally.

Also, `THttpSysServer` is tied to Windows because http.sys is. Sparkle offers other ways to host a module, including an Indy-based server described in the same server documentation, if you need to run elsewhere.

You might wonder whether I am arguing against the wizard and the design-time components. I am not. The wizard generates exactly the kind of service units we just typed, and the components are a comfortable way to configure a server. In my opinion, though, you should write the server by hand at least once. When something does not work, you will know which of the four boxes in the diagram to look at, and that knowledge pays for itself quickly. The counter-argument is just as valid: if your team maintains ten servers, generated and uniformly structured code beats hand-written individuality. Use the wizard for the routine and the manual approach for the understanding.

> **Alternatives**
> TMS XData is a commercial product. If you want to stay in Delphi with open source, have a look at mORMot 2, DelphiMVCFramework, and Horse; all three are well-known in the Delphi community. Outside of Delphi, a service this size is equally small in Express on Node.js or FastAPI in Python. I have laid out why XData is my personal choice in an earlier post; the right pick depends on your team and on the code you already have.

Takeaways

We set out to find how few steps a Delphi REST server really needs, and the count is five: a contract, an implementation, a server program, a URL reservation, and a request. Only three of them involve Pascal.

If you remember just a few things, make it these. The service is an ordinary Delphi interface, and XData derives URLs and JSON from it, so there is no serialization code to write. The units register themselves, which means they must be in the project's `uses` clause. The URL reservation is a Windows requirement and the first thing to check when a server cannot be reached. And the CORS middleware decides whether a browser-based client can use your server at all -- with the asterisk being a convenience for development that you should replace before you ship.

> A web service is an interface, a class, and a dozen lines to host them. Everything else is configuration.

The full source code consists of the three listings above; copy them into a folder, open `HelloServer.dpr`, and you are ready to go. Only by changing things will you get comfortable with it, so add a third method, return an object instead of a string, and look at what comes back. Imagine the possibilities you have now...

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 —