NEWS · JULY 14, 2026

Web Services sind nur Endpoints: REST und JSON in Delphi (5/5)

By Dr. Holger Flick



Teil 4 endete mit einer stillen kleinen Bombe. Wir hatten mit WebBroker einen Webserver gebaut, zugesehen, wie ein Browser `GET /` durch eine TCP-Verbindung schickt, und HTML zurückgesendet — und dann habe ich darauf hingewiesen, dass nichts in HTTP verlangt, dass die Antwort HTML *sein* muss. Eine berechnete Antwort kann genauso gut aus Daten bestehen.

Dieser eine Satz ist das ganze Geheimnis dieses letzten Teils. Denn in dem Moment, in dem Ihr Server **Daten statt Seiten** zurückgibt, hört er auf, „eine Website" zu sein, und wird zu dem, was der Rest der Branche einen **Web Service** nennt. Ein paar Namenskonventionen dazu, und Sie haben eine **REST-API**. Geben Sie einer ihrer URLs eine Aufgabe, und Sie haben einen **Endpoint**.

Diese drei Begriffe schüchtern viele erfahrene Desktop-Entwickler ein — ich habe es in Beratungsterminen selbst erlebt. Und es ist völlig unnötig, denn am Ende von Teil 4 *hatten Sie bereits einen gebaut*. Man hatte Sie nur noch nicht miteinander bekannt gemacht.

Dieses Finale leistet deshalb drei Dinge: Es entmystifiziert das Vokabular Wort für Wort, es zeigt den kompletten Roundtrip in echtem Delphi-Code — eine WebBroker-Action, die JSON ausliefert, und eine Desktop-App, die es mit `THTTPClient` konsumiert — und es zeigt, wie TMS XData alles, was diese Serie Ihnen beigebracht hat,

in ein paar Attribute an einem gewöhnlichen Delphi-Interface verwandelt. Danach beenden wir die Serie dort, wohin sie von Anfang an unterwegs war: an der Tür mit der Aufschrift *Docker*.

Dies ist die letzte Station einer fünfteiligen Reise, hier also noch einmal die vollständige Karte.

> **Diese Serie: Netzwerkgrundlagen für Delphi-Entwickler**
> 1. IP-Adressen, localhost und Ports — wo Programme im Netzwerk wohnen 2. DNS, die hosts-Datei und DHCP — wie Maschinen zu Namen und Adressen kommen 3. TCP-Verbindungen und Sockets — wie zwei Programme ein Gespräch führen 4. Webserver, HTTP und WebBroker — das Request/Response-Protokoll obendrauf 5. **Web Services, REST und JSON — dieser Beitrag** — wenn die Antwort aus Daten besteht, nicht aus Seiten

Das Vokabular, entmystifiziert

Jedes einschüchternde Wort in diesem Themenfeld beschreibt etwas Kleines. Gehen wir sie eines nach dem anderen durch, in einfacher Sprache, und heften jedes an das, was Sie aus den Teilen 1–4 bereits kennen.

- **Web Service** — ein Webserver, dessen Antworten für *Programme* gedacht sind, nicht für Menschen.

  Gleicher Port, gleiche TCP-Verbindung, gleicher HTTP-Request/Response-Zyklus wie in Teil 4. Nur der Content-Type der Antwort ändert sich.
- **Endpoint** — eine URL auf diesem Server, hinter der Verhalten steckt. `http://localhost:8080/status` ist

  ein Endpoint: Rufen Sie ihn auf, läuft Code und liefert eine Antwort. Wenn Sie in Teil 4 eine WebBroker-Action geschrieben haben, haben Sie bereits einen Endpoint geschrieben.
- **API** — *Application Programming Interface*: die Gesamtheit aller Endpoints, die ein Service anbietet, plus die

  Regeln für deren Aufruf. Sie ist der Vertrag des Service, so wie der `interface`-Abschnitt einer Delphi-Unit ein Vertrag ist.
- **REST** — ein *Stil*, eine API zu organisieren: URLs benennen **Dinge**, HTTP-Methoden sagen, **was damit geschehen soll**. Dazu gleich mehr.
- **Payload / Request Body** — die Daten, die Sie mit einem Request oder einer Response mitschicken. Wenn

  Ihre App einen neuen Kunden als JSON per POST überträgt, ist dieses JSON die Payload, transportiert im HTTP-Body, den Sie in Teil 4 kennengelernt haben.

Fällt Ihnen etwas auf? Kein einziges dieser Wörter hat einen neuen *Mechanismus* eingeführt. Ports, Namen,

Verbindungen, HTTP — die Maschinerie ist exakt der Stack, den Sie schon gebaut haben. Hier ist der ganze Wandel in einem Bild.



Gleicher Server, gleiches HTTP — der Browser bekommt eine Seite, Ihr Programm bekommt Daten

Lesen Sie das Diagramm von unten nach oben: Der untere Pfeil ist das *gesamte* Thema dieses Beitrags. Wenn der Anfragende ein Programm ist, antwortet der Server in einem Format, das ein Programm mühelos parst — und das Format, auf das sich die gesamte Branche geeinigt hat, ist JSON.

JSON in sechzig Sekunden

JSON — *JavaScript Object Notation*, definiert auf json.org — ist strukturierter Text, den Menschen und Programme gleichermaßen leicht lesen. Wer eine Delphi-Record-Deklaration lesen kann, liest JSON auf den ersten Blick. Hier ein Delphi-Record und sein JSON-Zwilling nebeneinander:

```
type
  TServerStatus = record
    Status: string;         // 'ok'
    Version: string;        // '1.4.2'
    UptimeSeconds: Integer; // 86400
  end;
```

Dieselbe Information als JSON-Dokument — geschweifte Klammern für das Objekt, `"name": value`-Paare, eckige Klammern (hier nicht gezeigt) für Arrays:

```
{
  "status": "ok",
  "version": "1.4.2",
  "uptimeSeconds": 86400
}
```

Delphi bringt alles mit, was man für den Weg zwischen beiden Welten braucht — kein Fremdcode nötig. `System.JSON` liefert `TJSONObject.ParseJSONValue`, um ein Dokument von Hand zu durchlaufen — Sie sehen es unten in Aktion. Und für den Fall „bilde es einfach auf meine Klasse ab" konvertiert die `TJson`-Klasse aus der Unit `REST.Json` in je einer Zeile pro Richtung: `TJson.ObjectToJsonString(MyObject)` serialisiert ein Objekt in einen JSON-String, und `TJson.JsonToObject<T>(MyString)` baut daraus wieder ein Objekt. Wenn ein JSON-Key nicht zu Ihrem Feldnamen passt, legt das `JSONName`-Attribut aus `REST.Json.Types` die Zuordnung explizit fest.

Das ist wirklich die gesamte JSON-Theorie, die diese Serie braucht: Es ist das Textformat, das Ihre Endpoints sprechen.

REST: Dinge bekommen URLs, Verben erledigen die Arbeit

„REST-API" klingt, als gehörte eine Zertifizierungsprüfung dazu. Die Arbeitsintuition passt in zwei Sätze. **URLs benennen Dinge** — Ressourcen, im REST-Vokabular: Kunden, Bestellungen, Rechnungen. **HTTP-Methoden sagen, was damit geschehen soll** — dieselben GET-, POST-, PUT- und DELETE-Methoden, die Sie in Teil 4 als Request-Verben kennengelernt haben, jetzt mit ihrer Wörterbuchbedeutung: lesen, anlegen, ersetzen, entfernen.

Kreuzen Sie beides, und Sie können fast jede reale API vorhersagen, ohne ihre Dokumentation zu lesen. Das Raster unten ist dieses komplette Denkmodell.

	`/customers` die ganze Sammlung	`/customers/42` ein bestimmter Kunde
GET	alle Kunden auflisten	Kunde 42 abrufen
POST	neuen Kunden anlegen	(hier selten genutzt)
PUT	(hier selten genutzt)	Kunde 42 ersetzen
DELETE	(gefährlich — meist gesperrt)	Kunde 42 entfernen

Die REST-Intuition: Ressourcen-URL mal HTTP-Verb ergibt die Operation

Lesen Sie jede Zelle als *Verb + URL = Operation*: `GET /customers/42` holt einen Kunden als JSON; `POST /customers` mit einer JSON-Payload legt einen an. Das Substantiv wohnt in der URL, das Verb in der Methode, die Daten reisen im Body. Das ist die Intuition, mit der berufstätige Entwickler tatsächlich Tag für Tag arbeiten. Der Vollständigkeit halber: „REST" hat durchaus eine formale Definition — sie stammt aus Roy Fieldings Dissertation aus dem Jahr 2000, die eine Reihe von Architektur-Constraints beschreibt — aber Sie brauchen die formale Abhandlung nicht, um vollkommen brauchbare APIs zu lesen, zu konsumieren oder zu entwerfen.

Beweis per WebBroker: Ihr Server aus Teil 4 ist bereits ein Web Service

Machen wir die These wörtlich, mit der kleinstmöglichen Änderung an dem, was Sie beim letzten Mal gebaut haben. In der WebBroker-Anwendung aus Teil 4 ist eine *Action* ein URL-Pfad, der mit einem Event-Handler verdrahtet ist. Hier ist eine Action für den Pfad `/status` — und die einzige Neuerung gegenüber Teil 4 sind zwei Property-Zuweisungen an `TWebResponse`:

```
procedure TWebModule1.StatusActionAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  // Der ganze Unterschied zwischen einer Website und einem Web Service:
  Response.ContentType := 'application/json';
  Response.Content :=
    '{ "status": "ok", "version": "1.4.2", "uptimeSeconds": 86400 }';
end;
```

Starten Sie das Programm, öffnen Sie `http://localhost:8080/status` im Browser, und Sie sehen rohes JSON statt einer Seite. Herzlichen Glückwunsch — dieser Browser-Tab schaut auf **einen Endpoint**. Dass `ContentType` dem Client „das ist `application/json`" mitteilt, ist das höfliche Etikett; der JSON-String in `Content`

ist die Payload. Keine neue Komponente, kein neues Protokoll, kein Framework. Ein Web Service *ist* ein Webserver, der Daten zurückgibt.

Der Kreis schließt sich: Ihre Desktop-App als Client

Nun zu der Hälfte, die in der Praxis am meisten zählt. Hier eine Meinung, die ich offen ausspreche und vertrete: **Für die meisten Delphi-Desktop-Teams macht das Konsumieren von Web Services 90 % des realen Bedarfs aus** — einen Zahlungsanbieter aufrufen, eine Versand-API, einen Lizenzserver, das eigene Backend — während das *Hosten* eines Service der seltenere Job ist. Das ist meine Erfahrung aus Jahren der Beratung, keine gemessene Statistik, gewichten Sie sie also entsprechend; wenn Ihr Produkt einen Server ausliefert, kippt die Balance natürlich. So oder so: Die Client-Seite ist unser Startpunkt.

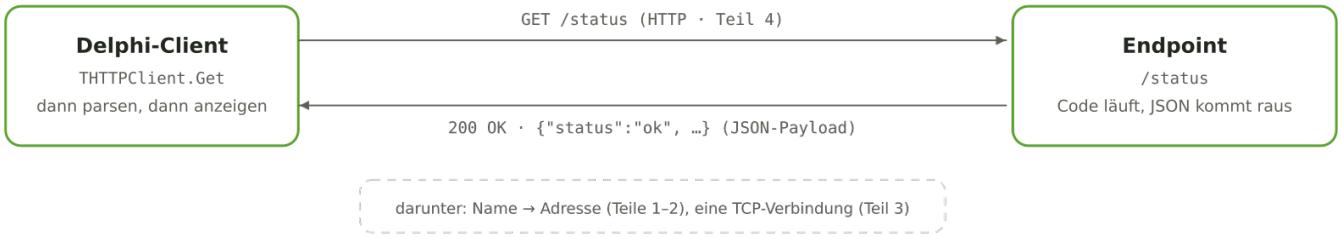
Delphis eingebauter Client ist `THTTPClient` in der Unit `System.Net.HttpClient` — mit einem auf das Formular ziehbaren Komponenten-Wrapper, `TNetHTTPClient` in `System.Net.HttpClientComponent`, falls Sie den RAD-Workflow bevorzugen. Beide sind die modernen, plattformübergreifenden RTL-Clients und funktionieren identisch in Delphi 13 Florence, dem aktuellen Release. Unseren Endpoint aufzurufen und die Antwort zu parsen sieht so aus:

```
uses
  System.Net.HttpClient, System.JSON, System.SysUtils;

procedure TMainForm.ShowServerStatus;
var
  LClient: THTTPClient;
  LResp: IHTTPResponse;
  LRoot: TJSONObject;
begin
  LClient := THTTPClient.Create;
  try
    // Eine Zeile: GET senden, auf die HTTP-Response warten (der Zyklus aus Teil 4).
    LResp := LClient.Get('http://localhost:8080/status');
    if LResp.StatusCode <> 200 then
      raise Exception.CreateFmt('Endpoint returned %d', [LResp.StatusCode]);

    // Die JSON-Payload in einen durchlaufbaren Objektbaum parsen.
    LRoot := TJSONObject.ParseJSONValue(LResp.ContentAsString) as TJSONObject;
    try
      lblVersion.Caption := LRoot.GetValue<string>('version');
      lblUptime.Caption  := LRoot.GetValue<Integer>('uptimeSeconds').ToString + ' s';
    finally
      LRoot.Free;
    end;
  finally
    LClient.Free;
  end;
end;
```

Nehmen Sie sich eine Sekunde, um zu würdigen, was hier quer durch die Serie gerade passiert ist. Ihre VCL-App hat einen Namen aufgelöst (Teil 2) zu einer Adresse und einem Port (Teil 1), eine TCP-Verbindung geöffnet (Teil 3), HTTP gesprochen (Teil 4) und einen Web Service konsumiert (Teil 5) — in rund fünfzehn Zeilen, von denen die meisten `try..finally`-Hygiene sind. Hier ist dieser Roundtrip als ein einziges Bild.



Der Roundtrip: Jede Schicht dieser Serie feuert in einem harmlos wirkenden Get-Aufruf

Die beiden horizontalen Pfeile sind der Request/Response-Zyklus aus Teil 4; die gestrichelte Box darunter sind die Teile 1–3 bei ihrer unsichtbaren Arbeit. Ihre Desktop-App ist jetzt ein Web-Service-Client — und jede Cloud-API auf diesem Planeten wird mit genau diesem Muster konsumiert.

Der erwachsene Server: TMS XData

JSON-Strings von Hand in einer WebBroker-Action zu schreiben ist perfekt fürs *Verstehen*, aber eine echte

API braucht Routing für viele Endpoints, automatische JSON-Serialisierung Ihrer Objekte, Parameter-Binding, Fehlerbehandlung. Genau das kauft Ihnen ein Framework, und das, zu dem ich in der kommerziellen Delphi-Welt greife, ist TMS XData, gebaut auf dem HTTP-Framework TMS Sparkle.

Die Kernidee von XData ist für einen Delphi-Entwickler wunderschön: **Sie deklarieren einen Service-Vertrag als gewöhnliches Delphi-Interface**, und das Framework macht daraus HTTP-Endpoints. Laut der offiziellen Dokumentation zu Service Operations sieht die Vertragsseite so aus:

```
uses
  XData.Service.Common;

type
  [ServiceContract]
  IServerInfoService = interface(IInvokable)
    ['{D8F62F84-6A2E-4C4B-9D0A-3C5B7C11A9E4}']
    [HttpGet] function Status: string;
    function Echo(const Text: string): string;
  end;

initialization
  RegisterServiceType(TypeInfo(IServerInfoService));
```

Eine Zeile pro neuem Konzept: `[ServiceContract]` markiert das Interface als Service; die Ableitung von `IInvokable` (plus die GUID) aktiviert die RTTI, die das Framework braucht; `[HttpGet]` bindet `Status` an einen GET-Request — standardmäßig reagieren XData-Operationen auf POST, dieses Attribut ist also unser REST-Verb-Regler; und `RegisterServiceType` teilt XData mit, dass das Interface existiert. Die Implementierung ist ebenso unspektakulär — eine gewöhnliche Klasse, die das Interface implementiert, mit `[ServiceImplementation]` markiert und auf dieselbe Weise registriert:

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

type
  [ServiceImplementation]
  TServerInfoService = class(TInterfacedObject, IServerInfoService)
  public
    function Status: string;
    function Echo(const Text: string): string;
  end;

function TServerInfoService.Status: string;
begin
  Result := 'ok';
end;

initialization
  RegisterServiceType(TServerInfoService);
```

Und das Hosting ist, gemäß dem Getting-Started-Guide, ein Server-Objekt plus ein Modul — `TXDataServerModule` hat einen Konstruktor-Overload, der nur die Basis-URL nimmt, für reine Service-Endpoints ist also keine Datenbank erforderlich:

```
uses
  Sparkle.HttpSys.Server, XData.Server.Module;

var
  Server: THttpSysServer;
begin
  Server := THttpSysServer.Create;
  Server.AddModule(TXDataServerModule.Create('http://localhost:2001/tms/api'));
  Server.Start;  // GET /tms/api/serverinfoservice/status antwortet nun mit {"value":"ok"}
end;
```

Wenn Sie Komponenten dem Code vorziehen: XData liefert auch Design-Time-Komponenten — einen `TSparkleHttpSysDispatcher` und einen `TXDataServer` aufs Formular ziehen, `BaseUrl` setzen, `Active` auf true stellen. Und auf der Konsumentenseite gibt es eine wunderbare Symmetrie: Mit `TXDataClient` kann ein Delphi-Client den Service über genau dasselbe Interface aufrufen — `Client.Service<IServerInfoService>.Status` — ganz ohne manuelles HTTP oder JSON. Methoden können Ihre eigenen Klassen oder `TList<T>` zurückgeben, und XData serialisiert sie automatisch nach JSON; es kann sogar interaktive OpenAPI/SwaggerUI-Dokumentation für Ihre Endpoints ausliefern — wobei OpenAPI der Standard für die maschinenlesbare Beschreibung einer API ist, ein Thema für einen anderen Tag.

Treten Sie einen Schritt zurück und schauen Sie, was diese Attribute gerade alles geschluckt haben: Ports, Routing, HTTP-Verben, JSON-Serialisierung — die ganze Serie, zusammengefaltet in eine Delphi-Interface-Deklaration. Das ist die Dividende eines guten Frameworks.

> **Alternativen — Open Source und jenseits von Delphi**
>
> Dieselbe Aufgabe erledigen Open-Source-Werkzeuge hervorragend, und die Fairness verlangt, sie zu nennen. In Delphi ist DelphiMVCFramework ein ausgereiftes, aktiv gepflegtes Open-Source-Framework für RESTful, JSON-sprechende Server — wenn eine kommerzielle Lizenz nicht zu Ihrem Projekt passt, ist es eine wirklich starke Wahl, kein Trostpreis. Außerhalb von Delphi sind genau diese Endpoints das, was Teams täglich mit Express und Fastify (Node.js), FastAPI (Python) und ASP.NET Core (C#) bauen.
>
> Die Konzepte dieser Serie übertragen sich unverändert auf alle davon — das ist ja gerade der Punkt.

Wählen Sie nach dem Bedarf Ihres Teams: XData für tiefe RAD-Integration und kommerziellen Support, DelphiMVCFramework für Open Source in Delphi, einen anderen Stack, wenn Ihr Team dort zu Hause ist.

Echte Services brauchen Authentifizierung

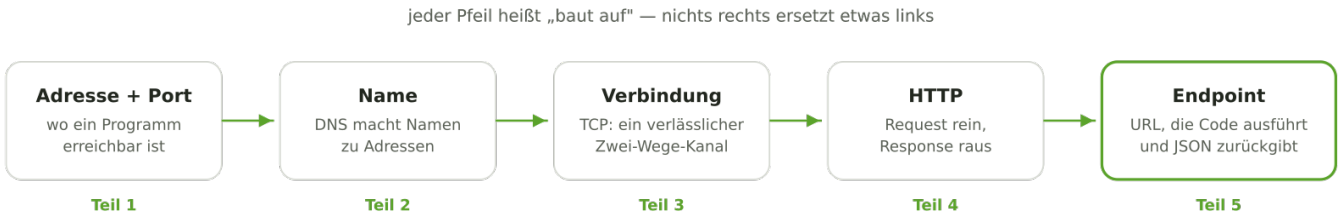
Jedes Beispiel hier steht auf localhost sperrangelweit offen — fürs Lernen in Ordnung, in Produktion tödlich. Reale Endpoints verlangen, dass Aufrufer nachweisen, wer sie sind — typischerweise per API-Key oder Token im `Authorization`-HTTP-Header. Wir stellen das Thema bewusst zurück; setzen Sie nur niemals einen unauthentifizierten Service außerhalb Ihres eigenen Rechners aus.

Was wir sonst noch zurückgestellt haben — mit Absicht

Drei weitere Produktionsthemen blieben außen vor, um das Denkmodell sauber zu halten: HTTPS/TLS (Verschlüsselung der Verbindung — Pflicht für alles Öffentliche), API-Versionierung (Endpoints weiterentwickeln, ohne alte Clients zu brechen) und die formale API-Beschreibung mit OpenAPI. Jedes verdient eine eigene Behandlung; keines ändert etwas an dem, was Sie hier gelernt haben.

Die ganze Serie in einem Bild

Fünf Beiträge, eine Treppe. Jeder Teil hat genau eine Frage beantwortet, und jede Antwort wurde zum Boden, auf dem der nächste Teil stand.

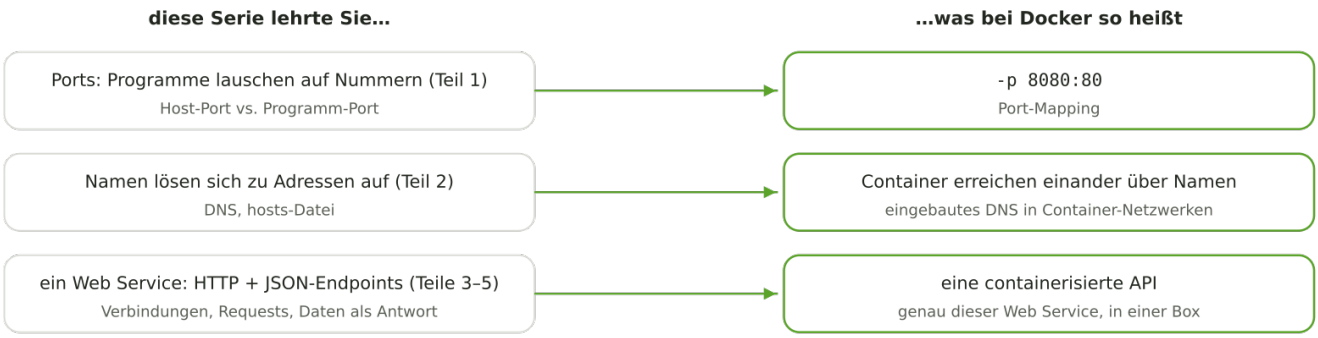


Der Serienbogen: von der nackten Adresse zum Endpoint, der Daten liefert

Lesen Sie es von links nach rechts, und der einst einschüchternde Satz *„unsere REST-API stellt JSON-Endpoints über HTTPS bereit"* zerfällt in fünf kleine Ideen, die Ihnen jetzt gehören. Auch oberhalb dieser Ebene ist nichts im Networking Magie — es sind diese fünf Bausteine, neu kombiniert.

Die Tür mit der Aufschrift Docker

Es gibt einen Grund, warum diese Serie existiert, und ich darf ihn endlich aussprechen: Alles, was Sie gerade gelernt haben, ist **exakt das Vokabular von Containern**. Docker — das Werkzeug, das eine Anwendung mit allem, was sie braucht, in eine isolierte, überall lauffähige Einheit namens Container verpackt — schüchtert Desktop-Entwickler aus genau demselben Grund ein wie vorhin „REST-API": Die Wörter sind fremd. Aber schauen Sie, was die Wörter tatsächlich bedeuten.



Sie sprechen bereits Docker: Jedes Container-Konzept entspricht einem Teil dieser Serie

Jede rechte Box ist eine linke Box in anderen Kleidern. Das kryptische `-p 8080:80` in jedem Docker-Tutorial? Das ist Port-Mapping — „Requests an Host-Port 8080 gehen an Port 80 des Programms im Container".

Reiner Teil 1. Container, die einander als `db` oder `api` finden statt über IP-Adressen? Namensauflösung — Docker-Netzwerke haben eingebautes DNS. Reiner Teil 2. Und „eine containerisierte REST-API deployen"? Das ist der Web Service aus genau diesem Beitrag, verpackt in eine Box mitsamt seiner Laufzeitumgebung, lauschend auf einem Port, HTTP mit JSON beantwortend. Sie könnten diese API heute in Delphi mit XData schreiben, und sie würde von der Box um sie herum nichts mitbekommen.

Ich bringe Ihnen hier bewusst *kein* Docker bei — das verdient eine eigene Serie, und die bekommt es auch:

Eine Docker-Serie für Delphi-Entwickler erscheint als Nächstes auf diesem Blog, auf dem Fundament,

das diese fünf Teile gerade gegossen haben. Wenn sie erscheint, werden Sie keine fremde neue Welt lernen. Sie werden eine alte wiedererkennen.

Das Wichtigste in Kürze

Das einschüchternde Vokabular war der einzige schwere Teil, und der ist jetzt weg. Ein Web Service ist ein Webserver, der Daten zurückgibt. Ein Endpoint ist eine URL mit Code dahinter. REST heißt Substantive in URLs und Verben in Methoden. JSON ist strukturierter Text, den Ihre RTL nativ parst. Ihre Delphi-App kann all das mit `THTTPClient` konsumieren, und Frameworks wie TMS XData — oder das quelloffene DelphiMVCFramework — lassen sie all das mit den Werkzeugen *bereitstellen*, in denen Sie ohnehin denken: Interfaces, Attribute,

Klassen.

> Vor fünf Teilen war „die App spricht mit einer REST-API auf Port 443" ein Satz voller Fremder. Heute ist jedes Wort darin ein Bekannter — und die nächste Serie fügt nur einen einzigen weiteren hinzu: *Container*.

Wenn Ihr Delphi-Produkt eine Cloud-API konsumieren oder ein eigenes Web-Service-Backend aufbauen soll — und Sie jemanden möchten, der genau diesen Weg mit Legacy-VCL-Codebasen schon gegangen ist — sprechen Sie mich an. Und wir sehen uns bei der Docker-Serie.

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 —