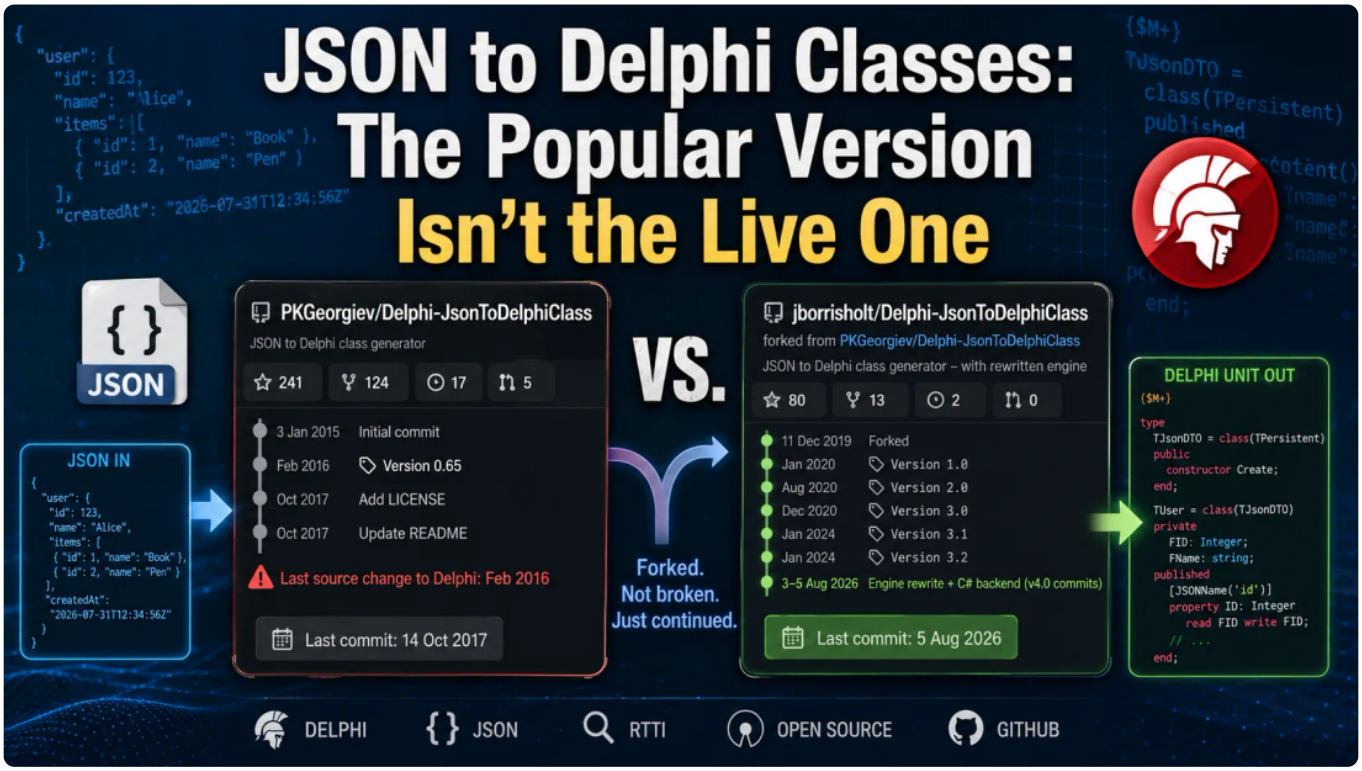
NEWS · AUGUST 7, 2026

JSON zu Delphi-Klassen: Die populäre Fassung ist nicht die lebendige

Ein Leser schrieb mir wegen eines Delphi-Klassengenerators für JSON — und das Interessante daran war, dass die von ihm gepflegte Fassung eine Kopie eines fremden Projekts ist, dass das Original dreimal so viele Sterne und seit 2016 keinen Quelltext mehr bekommen hat, und woran Sie ein lebendiges Repository von einem bloß berühmten unterscheiden.

By Dr. Holger Flick



Vor ein paar Tagen kam eine Nachricht über das Kontaktformular dieser Seite herein. Jens Borrisholt hatte die Beiträge zu JSON und RTTI gelesen und dachte, eines seiner Open-Source-Projekte könnte dazu passen:

ein Werkzeug, das aus einem beliebigen JSON-Dokument die vollständigen Delphi-Klassen erzeugt, die diese Daten aufnehmen. Links JSON einfügen, rechts eine kompilierbare Unit erhalten, Serialisierung inklusive. Er hatte die Generator-Engine gerade von Grund auf neu geschrieben und wollte sie zeigen.

Also habe ich nachgesehen. Und das Erste, was GitHub mir mitteilte, in kleiner grauer Schrift unter dem Repository-Namen, war dies: **forked from PKGeorgiev/Delphi-JsonToDelphiClass**.

Diese Zeile entpuppte sich als das Interessanteste auf der ganzen Seite. Ein **Fork** ist im Sprachgebrauch von

GitHub eine vollständige Kopie des Projekts eines anderen unter Ihrem eigenen Namen — samt kompletter Historie —, die Sie anschließend frei in Ihre eigene Richtung weiterentwickeln dürfen. Das Werkzeug, das Jens mir zeigte, ist also nicht das Original. Es ist seine Fortführung eines Projekts, das jemand anderes begonnen hat, und dieses Original steht nach wie vor öffentlich da.

Und das ist hier von Bedeutung, denn das ursprüngliche Repository hat dreimal so viele Sterne wie die Kopie von Jens, und seine letzte Änderung am Delphi-Quelltext stammt vom Februar 2016. Die Kopie hat weniger Sterne, ein ruhigeres Profil — und eine Generator-Engine, die drei Tage bevor Jens mir schrieb neu gebaut wurde. Wenn Sie heute nach diesem Werkzeug suchen, wären das populäre Ergebnis und das gepflegte Ergebnis nicht dieselbe URL. Diese Lücke ist kein Delphi-Problem, sondern ein Problem kleiner Open-Source-Ökosysteme, das es überall gibt.

In diesem Beitrag geht es um das Werkzeug, um die beiden Menschen dahinter — und um die dreißig Sekunden Lesearbeit, die Ihnen verraten, welches Repository tatsächlich lebt.

Was das Werkzeug wirklich tut

Vor der Archäologie der praktische Teil: Es handelt sich um ein Data-Binding-Werkzeug für JSON, und am schnellsten verstehen Sie es, wenn Sie sich ansehen, was hinten herauskommt.

Sie geben ein JSON-Dokument hinein. Das Werkzeug läuft durch die Struktur, leitet für jeden Wert einen Typ ab, erkennt, welche Formen Objekte und welche Listen sind, und erzeugt eine Delphi-Unit mit einer Klasse je Objektform — also einen Satz DTOs, reine Datenträgerklassen — samt der Verdrahtung, die eine Instanz wieder zu JSON macht und umgekehrt.

Hier ein echtes Beispiel. Diese Unit liegt im Smoke-Test-Build-Ordner des Repositories, ist also die eigene Ausgabe des Generators und nicht etwas, das ich für den Artikel von Hand geschrieben hätte. Sie entspricht einem Dokument dieser Form:

```
{
  "name": "Widget batch",
  "items": [
    { "id": 1 },
    { "id": 2 }
  ]
}
```

Und das erzeugt der Generator daraus:

```
unit RootU;

interface

uses
  Pkg.Json.DTO, System.Generics.Collections, REST.Json.Types;

{$M+}

type
  TItems = class
  private
    FId: Integer;
  published
    property Id: Integer read FId write FId;
  end;

  TRoot = class(TJsonDTO)
  private
    [JSONName('items'), JSONMarshalled(False)]
    FItemsArray: TArray<TItems>;
    [GenericListReflect]
    FItems: TObjectList<TItems>;
    FName: string;
    function GetItems: TObjectList<TItems>;
  protected
    function GetAsJson: string; override;
  published
    property Items: TObjectList<TItems> read GetItems;
    property Name: string read FName write FName;
  public
    destructor Destroy; override;
  end;
```

Fast jede Zeile davon ist etwas, das die früheren Beiträge dieser Serie bereits behandelt haben — genau deshalb hat mich das Projekt neugierig gemacht.

Die Direktive `{$M+}` und der `published`-Abschnitt sorgen dafür, dass der Compiler Laufzeit-Typinformationen für diese Eigenschaften erzeugt — der Mechanismus hinter allem, was RTTI tut. Die Attribute `[JSONName('items')]` und `[JSONMarshalled(False)]` stammen aus `REST.Json.Types` und werden zur Laufzeit vom Serializer gelesen, also exakt das attributgesteuerte Muster aus RTTI Teil 2. Und die Basisklasse erledigt die eigentliche Arbeit über die RTL:

```
function TJsonDTO.GetAsJson: string;
begin
  Result := TJson.ObjectToJsonString(Self, FOptions);
end;
```

Das ist `REST.Json` — einer der drei Wege, auf denen die RTL JSON spricht. Der Generator ersetzt Delphis Serializer also nicht. Er erzeugt Klassen, die so geformt sind, dass Delphis Serializer das Richtige produziert, und füllt die Lücken, wo die Standardeinstellungen der RTL nicht genügen würden.

Die zwei Lücken, die er für Sie schließt

Zwei Details in dieser generierten Unit wirken wie Ballast, bis man weiß, was sie umgehen.

Das erste ist das doppelte Feld für Listen: ein `TArray<TItems>`, das den JSON-Namen trägt, und ein mit `[GenericListReflect]` markiertes `TObjectList<TItems>`, das Ihr Code tatsächlich benutzt. Der Serializer der RTL kommt mit dynamischen Arrays sauber zurecht, würde bei einer `TList<T>` aber bereitwillig auch deren interne Implementierungsfelder serialisieren. Deshalb gibt Ihnen der Generator eine Liste als öffentliche API

und hält ein schlichtes Array als internen Übergang bereit, das beim Serialisieren aktualisiert wird. Die Release Notes des Repositories beschreiben die Korrektur so: Die passende `published`-Eigenschaft wird über RTTI aufgelöst, „avoiding the internal `TList<T>` implementation data that Delphi's default serializer would otherwise emit".

Das zweite sind Datumswerte. `TJsonDTO` setzt seine Optionen im Konstruktor auf `[joDateIsUTC, joDateFormatISO8601]`, und der Generator erkennt ISO-8601-Werte in Ihrer Eingabe und typisiert sie als `TDateTime` statt als `string`. Wenn Sie den Beitrag über den Datumstyp gelesen haben, den JSON nicht hat, wissen Sie: Genau

diese eine Entscheidung — welche der beiden konkurrierenden Konventionen Sie verwenden und ob Sie bei der Zeitzone ehrlich sind — ist der Ort, an dem die meisten JSON-Datumsfehler wohnen.

> **Das ist generierter Code, und er verhält sich auch so**
> Alles oben Gezeigte wird bei jeder Generierung neu erzeugt. Wenn Sie die entstandene Unit von Hand bearbeiten und danach aus einem aktualisierten JSON-Beispiel neu generieren, sind Ihre Änderungen verschwunden. Behandeln Sie das JSON-Dokument als Quelle und die Unit als Build-Ergebnis — oder akzeptieren Sie, dass Sie sie geforkt haben, und hören Sie auf, neu zu generieren.

Zwei Repositories mit demselben Namen

Nun der Teil, der das Thema für mich schreibenswert gemacht hat. Das Werkzeug hat zwei Zuhause auf GitHub, und sie befinden sich in sehr unterschiedlichem Zustand.

Was ein Fork ist — und warum er nichts Schlimmes bedeutet

Wenn Sie Ihre Laufbahn in Delphi verbracht haben und nicht auf GitHub, klingt „Fork" nach einem Zerwürfnis. Meistens ist es keines, also lohnen sich dreißig Sekunden Definition.

Forken ist ein Klick. GitHub nimmt das gesamte Repository — jede Datei, jeden Commit zurück bis zum allerersten — und legt unter Ihrem Konto eine Kopie an, die Ihnen gehört und die Sie frei ändern dürfen. Das Original bleibt unangetastet. Zwei Dinge unterscheiden einen Fork davon, einfach ein ZIP des Quelltextes herunterzuladen: GitHub hält dauerhaft fest, aus welchem Projekt Ihres hervorgegangen ist, und zeigt das unter dem Namen Ihres Repositories an — und es bewahrt die gemeinsame Historie, sodass Ihre Änderungen dem Original weiterhin als Pull Request angeboten werden können, wenn dessen Eigentümer sie haben möchte.

Geforkt wird aus zwei ganz unterschiedlichen Gründen. Der alltägliche ist das Beitragen: Sie forken, beheben einen Fehler und schicken die Korrektur zurück nach oben; danach hat Ihre Kopie ihre Aufgabe erfüllt. Der andere Grund ist der aus dieser Geschichte — das Original bewegt sich nicht mehr, jemand möchte, dass das Werkzeug weiter besser wird, und seine Kopie wird still und leise zu der Fassung, die eigentlich alle benutzen sollten.

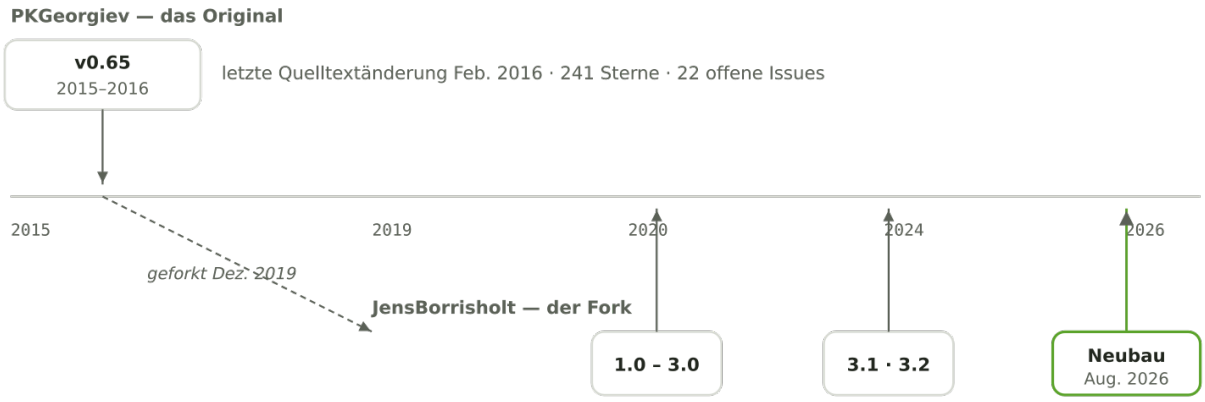
Die nächstliegende Delphi-Analogie: Sie nehmen den Quelltext einer fremden Komponente, legen ihn in Ihre eigene Bibliothek und pflegen ihn das nächste Jahrzehnt selbst, weil der Autor keine E-Mails mehr beantwortet. Das hat jedes Delphi-Haus schon getan. Der Unterschied ist, dass ein Fork es öffentlich tut, mit dokumentierter Abstammung und mit einer Lizenz, die es ausdrücklich erlaubt — und diese Erlaubnis ist keine Formalie. Sie ist der Grund, warum diese Geschichte ein Erfolg ist und kein Diebstahl.

Die beiden Zeitlinien

Das Original hat Petar Georgiev in Sofia, Bulgarien, am 3. Januar 2015 angelegt — eine FireMonkey-Desktop-Anwendung, die JSON in einer Treeview darstellte und passende Delphi-Klassen erzeugte, veröffentlicht unter der MIT-Lizenz. Es wurde damals gut aufgenommen; DelphiABall hat es 2016 vorgestellt. Dann hörte

es auf. Der Master-Branch trägt insgesamt fünf Commits: den Initial-Commit, eine README-Aktualisierung, ein „Version 0.65" im Februar 2016 und zwei Commits im Oktober 2017, die die LICENSE-Datei hinzufügten. Während ich dies schreibe, hat es 241 Sterne, 124 Forks sowie 17 offene Issues und fünf offene Pull Requests, die auf einen Maintainer warten, der weitergezogen ist.

Jens Borrisholt hat es am 11. Dezember 2019 geforkt und liefert seitdem. Version 1.0 kam im Januar 2020, 2.0 im August 2020, 3.0 im Dezember desselben Jahres. Dann eine Lücke von vier Jahren. Dann 3.1 und 3.2 im Januar 2024. Dann wieder eine Lücke — und dann, vom 3. bis 5. August 2026, ein Schwall von Commits, der die Generierungs-Engine vollständig ersetzte und eine zweite Ausgabesprache hinzufügte. Sein Fork hat 80 Sterne.



Dasselbe Werkzeug, zwei Repositories: wo die Sterne sind und wo die Commits sind

Lesen Sie dieses Diagramm in der einen Richtung, und es ist eine Geschichte über Aufgabe. Lesen Sie es in der anderen, und es ist genau das System, wie es gedacht war: Petar veröffentlichte unter MIT, und das ist eine stehende Einladung, die Arbeit weiterzuführen, wenn er sich anderem zuwendet. Jens hat die Einladung angenommen. Hier ist nichts schiefgegangen. Das Einzige, was schiefging: Suchmaschinen und Sternezähler haben die Nachricht nie bekommen.

Woran Sie erkennen, welches Repository lebt

Das lohnt sich zu verallgemeinern, denn das Delphi-Ökosystem ist voll von diesem Muster — und jedes andere kleine Ökosystem auch. Die Signale stehen alle auf der Startseite des Repositories, und keines davon ist die Sternezahl.

Dreißig Sekunden auf einem beliebigen GitHub-Repository

Lesen Sie die graue Zeile unter dem Titel. Steht dort „forked from …", sind Sie auf einer Kopie — sehen Sie sich das Eltern-Repository an und vergleichen Sie. **Achten Sie auf das Datum des letzten Commits im Default-Branch**, nicht auf den „Updated"-Zeitstempel. Ein Push in irgendeinen verwaisten Nebenzweig frischt Letzteren auf und sagt Ihnen nichts. **Öffnen Sie die Releases-Seite.** Getaggte Releases mit Datum sind das klarste Wartungssignal, das ein Projekt abgibt. **Prüfen Sie offene Issues und Pull Requests.** Ein Stapel von beidem, jahrelang ohne Antwort des Maintainers, heißt: Da ist niemand mehr. Null offene Issues bei einem aktiven Projekt heißt meist: Jemand liest sie tatsächlich. **Sehen Sie sich dann die Fork-Liste an**, sortiert nach Aktivität. Wenn das Original ruht, sitzt der Nachfolger fast immer dort drin.

Sterne messen, wie viele Menschen ein Projekt einmal interessant fanden. Sie sind kumulativ und werden nie weniger — niemand geht zurück und entfernt seinen Stern, wenn ein Repository nicht mehr gepflegt wird. Das ist kein Fehler von GitHub, aber es bedeutet, dass die Zahl ganz oben eine Frage über 2016 beantwortet und keine über heute.

Die andere Seite

Ich würde heute jemanden zum Fork schicken, aber das ursprüngliche Repository ist deswegen nicht wertlos, und das Argument ist nicht einseitig. Es ist nach wie vor die kanonische URL, auf die alle verlinken, es enthält die vollständige frühe Historie und die Issue-Diskussionen, und seine MIT-Lizenz ist der Grund, warum alles Nachgelagerte legal ist. Auch Forks werden still — dieser hatte vier Jahre Pause zwischen 3.0 und 3.1 —, und ein Projekt mit einem einzigen Maintainer hat einen Busfaktor von eins, so gut die vergangene Woche auch war. Und wenn Sie Version 3.2 bereits produktiv einsetzen, ist „die Engine wurde letzten Dienstag komplett neu geschrieben" ein Grund, sorgfältig zu testen, und kein Grund, heute Abend zu aktualisieren.

Die beiden Menschen dahinter

Beide Namen verdienen mehr als eine Fußnote, denn keiner davon ist ein Begriff, und beide haben echte Arbeit geleistet.

Petar Georgiev ist Entwickler in Sofia, Bulgarien, mit 50 öffentlichen Repositories und einem Blog unter pgeorgiev.com. Er baute das Original 2015 und löste die interessanten Teile des Problems: beliebiges JSON durchlaufen, Delphi-Typen aus untypisierten Werten ableiten, ISO-8601-Datumswerte automatisch erkennen, reservierte Delphi-Wörter in generierten Bezeichnern behandeln und den Destruktor-Code erzeugen, der verschachtelte Objekte wieder aufräumt. Alles, was das Werkzeug heute kann, wächst aus diesem Entwurf. Danach hat er sich, soweit GitHub das zeigt, anderem zugewandt. Das ist ein völlig normales Ende für ein Nebenprojekt — und beim Abschied MIT zu wählen war die großzügige Entscheidung, die das Werkzeug sein eigenes Interesse überleben ließ.

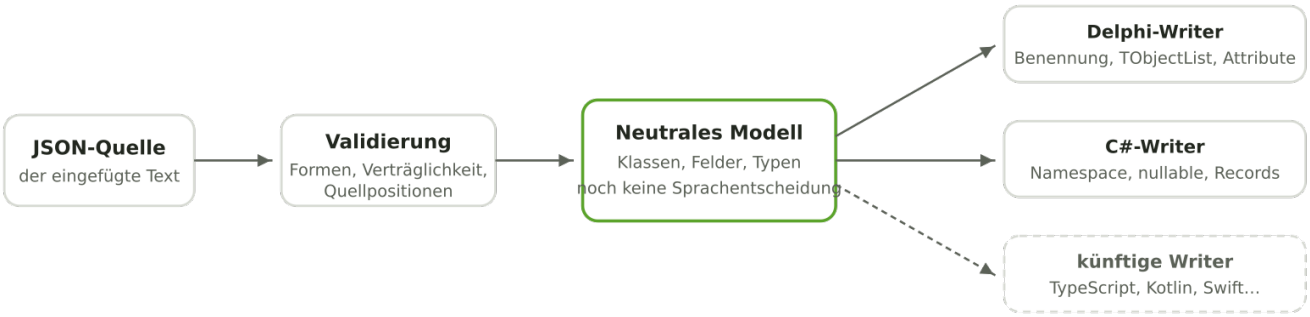
Jens Borrisholt ist Senior Software Developer in Dänemark mit rund dreißig öffentlichen Repositories, und sein GitHub liest sich wie das von jemandem, der baut, was er selbst braucht, und es dann veröffentlicht. Da ist eine Portierung der `Console`-Klasse aus C# nach Delphi — sein am meisten bestirntes Projekt —, objektorientierte Tastatur- und Maus-Hooks, eine Google-Text-to-Speech-Demo, ausgeliefert in Delphi und C#, ein kleines Werkzeug zur Online-/Offline-Erkennung und eine Bibliothek mit JSON-Interceptoren.

Seine Release-Historie kommt in Schüben: ein Schwall 2020, Stille, ein Schwall 2024, Stille, ein deutlich größerer Schwall in diesem Monat. Ich möchte das offen benennen, statt es zu beschönigen, denn es ist die ehrliche Gestalt der meisten Ein-Personen-Open-Source-Projekte. Der Maintainer hat einen Hauptberuf.

Das Projekt bewegt sich, wenn Zeit und Anlass da sind, und ruht sonst. Schubweise ist nicht dasselbe wie aufgegeben — der Unterschied liegt darin, ob die Schübe weiter kommen, und hier tun sie das nachweislich.

Was der Neubau im August 2026 tatsächlich verändert hat

Der Neubau ist der Grund, warum Jens geschrieben hat, und er ist mehr als ein Aufräumen. Der alte Code erzeugte Delphi-Quelltext, während er durch das JSON lief: Traversierung, Validierung, Typableitung, Benennung und Textausgabe geschahen alle in einem zustandsbehafteten Durchlauf. Die neue Engine teilt das in Stufen mit expliziten Ein- und Ausgaben — die Gestalt eines kleinen Compilers.



Ein validiertes Modell in der Mitte — und jede Ausgabesprache wird zu einem Writer

Der Sinn dieses mittleren Kastens ist, dass nichts darin weiß, was Delphi ist. Ein Feld merkt sich seinen JSON-Namen, seinen JSON-Pfad, ob es optional oder nullable war und wo im Quelltext es herkam. Eine Klasse hat eine Identität, die nicht davon abhängt, welchen Bezeichner ihr ein Backend am Ende gibt. Arrays sind rekursiv statt mit einem Dimensionszähler versehen, sodass eine Matrix schlicht „Array von Array von Integer" ist. Und entscheidend: Wie ein Wert in JSON *dargestellt* wird, ist getrennt davon gespeichert, was er *bedeutet* — ein ISO-8601-Zeitstempel bleibt ein JSON-String, während „das ist ein Datum mit Uhrzeit" als Semantik daneben steht.

Erst wenn dieses Modell vollständig ist, läuft ein Sprach-Backend. `TDelphiNaming` entscheidet über Bezeichner und maskiert reservierte Wörter, `TDelphiUnitWriter` wählt zwischen `TList<T>` und `TObjectList<T>` und gibt die Unit aus — und keiner von beiden verändert das Modell.

Zwei Folgen sind im Repository bereits sichtbar. Die Fehlermeldungen wurden deutlich besser: Weil die Validierung gegen den Quelltext läuft und Zeichenbereiche behält, meldet `[[1, 2], ["3", "4"]]` den erwarteten Typ, den tatsächlichen Typ und den fehlschlagenden JSON-Pfad, und die GUI markiert die betreffende Textstelle — statt still einen falschen Delphi-Typ zu wählen. Und am 5. August, einen Tag nachdem das neutrale Modell gelandet war, erschien ein vollständiges C#-Backend, mit eigenen Benennungsregeln, eigenen Einstellungen für Namespaces und nullable Wertetypen und einem eigenen Writer, der eine komplette `.cs`-Datei ausgibt. Das ist Architektur, die sich öffentlich auszahlt.

Meiner Ansicht nach war das die richtige Entscheidung, und der ehrliche Grund für diese Überzeugung ist, dass das C#-Backend einen Tag später kam und nicht ein Quartal später. Eine Abstraktion, die sofort einen zweiten Fall aufnimmt, war eine Abstraktion, die tatsächlich gebraucht wurde. Andererseits ist ein kompletter Engine-Neubau das Riskanteste, was man einem funktionierenden Werkzeug antun kann, und dieser hier ist eine Woche alt, während ich schreibe. Jens hat die alten Benennungs- und Ausgaberegeln durch eigene Kompatibilitätstests abgedeckt, was der richtige Instinkt ist — aber „durch Tests abgedeckt" und „passt zu dem, was Ihr Produktivcode seit vier Jahren einliest" sind zwei verschiedene Aussagen. Version 4.0 existiert derzeit außerdem als Commits und nicht als getaggtes Release. Wenn Sie das in einem Build einsetzen, fixieren Sie also 3.2 und probieren Sie 4.0 daneben aus, bevor Sie umschalten.

Der Rest der Landschaft

Delphi liefert seit Jahrzehnten einen XML Data Binding Wizard mit — richten Sie ihn auf ein Schema oder ein Beispieldokument, und er erzeugt Interfaces und Implementierungsklassen für jeden Knotentyp. Das Gegenstück für JSON kam nie. Die RTL gibt Ihnen die ausgezeichneten Bausteine auf niedriger Ebene — `System.JSON`, `REST.Json` und die RTTI, die beides antreibt, alles behandelt in der JSON-Serie —, aber nichts, was aus einem Beispieldokument eine typisierte Unit macht.

Die Lücke wird auch nicht von außen gefüllt. quicktype, das Werkzeug, zu dem die meisten anderen Ökosysteme greifen, erzeugt Typen aus JSON für 27 Zielsprachen. Delphi und Object Pascal sind nicht darunter. Also hat die Delphi-Community es selbst gebaut, und zwar mehr als einmal.

> **Weitere Wege von JSON zu typisiertem Delphi**
>
> marlonnardi/JsonToDelphi ist ein eigenständiges Projekt derselben Familie, das dieselbe `Pkg.Json.DTO`-Laufzeitlinie teilt und Oberflächen für VCL, FMX und uniGUI ergänzt. Es wird aktiv gepflegt und hat eine treue Anhängerschaft — klären Sie vor dem Einsatz die Lizenzlage, denn das Repository trägt derzeit keine Lizenzdatei. Neon von Paolo Rossi löst das benachbarte Problem hervorragend: Statt Klassen zu erzeugen, serialisiert es die Klassen, die Sie bereits haben, mit weit mehr Kontrolle über Benennung, Schreibweise und eigene Konverter, als die RTL bietet. Es ist die Serialisierungs-Engine hinter der REST-Bibliothek WiRL, und es ist das, wozu ich greifen würde, wenn die Delphi-Seite die Quelle der Wahrheit ist. Wo es die RTL ablöst, habe ich im Beitrag über Serialisierung beschrieben. mORMot 2 geht einen dritten Weg: eine extrem schnelle, RTTI-getriebene JSON-Schicht innerhalb eines viel größeren Werkzeugkastens aus ORM, SOA und REST. Größere Festlegung, entsprechend mehr Möglichkeiten. Und wenn die API, die Sie konsumieren, eine OpenAPI-Beschreibung veröffentlicht, generieren Sie lieber aus *dieser* als aus einer Beispielantwort — landgraf-dev/openapi-delphi-generator und paolo-rossi/OpenAPI-Delphi tun genau das. Ein Schema sagt Ihnen, welche Felder optional sind; eine Beispielantwort kann Ihnen nur sagen, welche Felder an jenem Tag zufällig vorhanden waren.

Dieser letzte Punkt ist die eigentliche Grenze jedes Generators, der von JSON-Beispielen ausgeht, und er gehört klar ausgesprochen statt versteckt. Ableitung aus einem einzigen Dokument ist eine Vermutung: Ein Feld, das in Ihrem Beispiel `null` ist, sieht aus wie ein String; ein Feld, das fehlt, sieht aus, als gäbe es es nicht; und eine Ganzzahl, die bisher immer klein war, sieht aus wie ein `Integer` — bis zu dem Tag, an dem die API eine ID schickt, die ein `Int64` braucht. Der Generator sieht nur, was Sie ihm gezeigt haben. Veröffentlicht der Anbieter ein Schema, gewinnt das Schema. Tut er es nicht — und viele reale APIs tun es nicht —, dann erspart Ihnen ein beispielbasierter Generator einen Nachmittag Tipparbeit, und die Durchsicht des Abgeleiteten ist Ihre Aufgabe, nicht die des Werkzeugs.

Zum Mitnehmen

Das Werkzeug ist wirklich gut und, wenn Sie in Delphi JSON-APIs konsumieren, zwanzig Minuten Ihrer Zeit wert: Fügen Sie eine Antwort ein, lesen Sie die erzeugte Unit, und Sie werden fast alles darin aus der RTL wiedererkennen. Die dauerhaftere Lehre steckt aber in der grauen Zeile unter dem Repository-Titel.

- **Das berühmte und das gepflegte Repository sind häufig nicht dieselbe URL.** Prüfen Sie das Fork-Banner, den letzten Commit im Default-Branch und die Releases-Seite, bevor Sie auf die Sternezahl schauen. Sterne werden nur mehr.
- **Ein Fork ist keine Spaltung.** Petar Georgiev hat die Sache gebaut und unter MIT lizenziert; Jens Borrisholt hat sie vier Jahre später aufgegriffen und liefert seither. Das ist Open Source, die funktioniert, nicht scheitert.

- **Der Neubau hat sich gelohnt, und das C#-Backend ist der Beweis.** Validieren, ein sprachneutrales Modell bauen, dann ausgeben — die zweite Ausgabesprache kam einen Tag nach der Abstraktion.
- **Generieren Sie aus einem Schema, wenn es eines gibt.** Ableitung aus einem einzigen Beispieldokument ist eine begründete Vermutung über Optionalität und Wertebereiche, und die Durchsicht gehört Ihnen.

> Sternezahlen sagen Ihnen, was Menschen einmal interessant fanden. Commit-Daten sagen Ihnen, wer noch da ist.

Zum Schluss, und aufrichtig: Danke, Jens, dass Sie sich gemeldet haben, und für die Jahre unbezahlter Wochenenden hinter diesen Release-Tags. Und danke, Petar, dass Sie es überhaupt gebaut haben und für die Lizenz, die es weiterleben ließ. Delphis RTL lässt Lücken wie diese, und der einzige Grund, warum sie gefüllt werden, ist, dass Menschen in dieser Community sie still und leise füllen und das Ergebnis verschenken. Wenn Sie an etwas in diesem Geist arbeiten, würde ich gern davon hören.

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 —