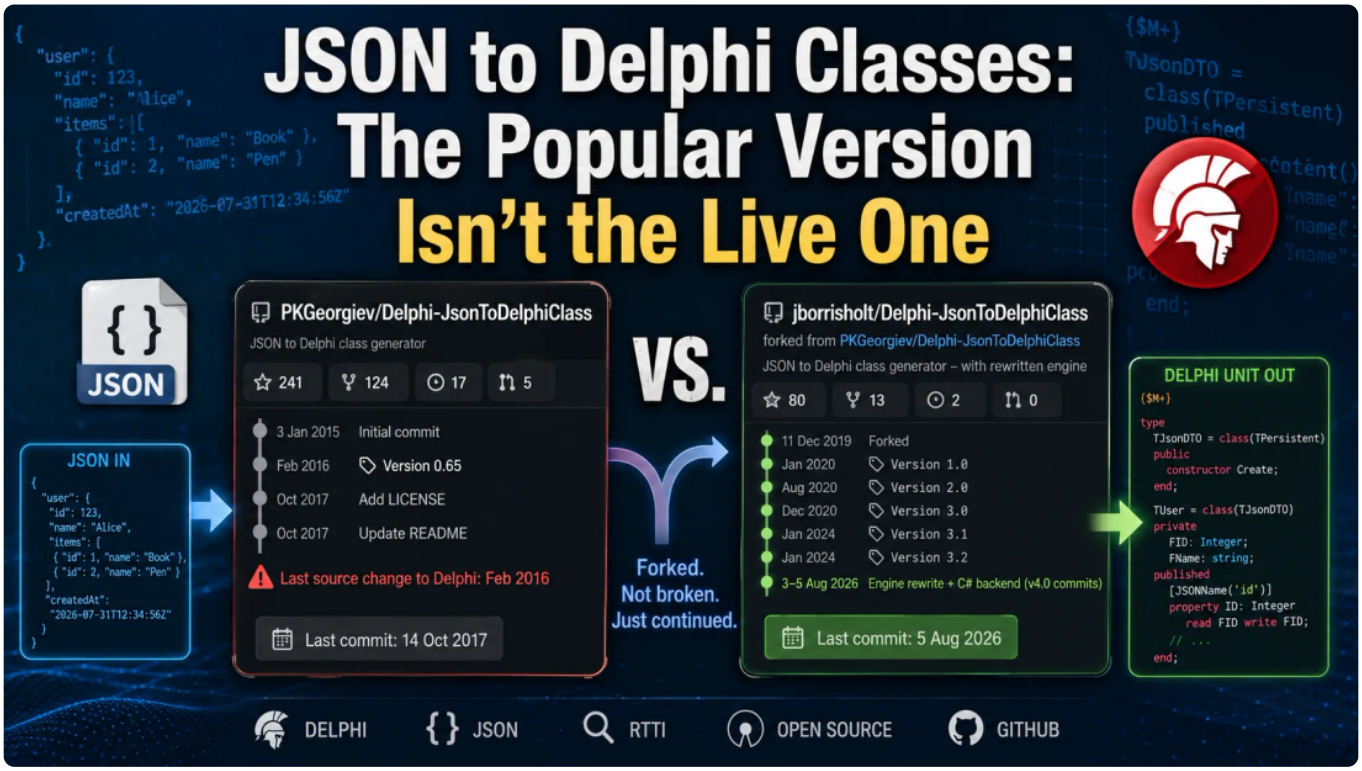
NEWS · AUGUST 7, 2026

JSON to Delphi Classes: The Popular Version Isn't the Live One

A reader wrote in about a Delphi JSON class generator — and the interesting part turned out to be that the version he maintains is a copy of someone else's project, that the original has three times the stars and no code since 2016, and how you tell a living repository from a merely famous one.

By Dr. Holger Flick



A few days ago a message came in through the contact form on this site. Jens Borrisholt had been reading the JSON and RTTI posts and thought one of his open-source projects might be a fit: a tool that takes an arbitrary JSON document and generates the complete Delphi classes to hold it. Paste JSON in one pane, get a compilable unit in the other, serialization included. He had just rewritten the generator engine from scratch and wanted to show it off.

So I went to look. And the first thing GitHub told me, in small grey type under the repository name, was this: **forked from PKGeorgiev/Delphi-JsonToDelphiClass**.

That line turned out to be the most interesting thing on the page. In GitHub's vocabulary, a **fork** is a complete copy of someone else's project taken under your own name — full history included — which you are then free to develop in your own direction. So the tool Jens was showing me is not the original. It is his continuation of a project somebody else started, and the original is still sitting there in public.

That matters here, because the original repository has three times as many stars as Jens's copy, and its last change to the Delphi source was in February 2016. The copy has fewer stars, a quieter profile, and a generator engine that was rebuilt three days before Jens wrote to me. If you searched for this tool today, the popular result and the maintained result would not be the same URL — and that gap is not a Delphi problem, it is a small-community open-source problem that shows up everywhere.

This post is about the tool, about the two people who built it, and about the thirty seconds of reading that tells you which repository is actually alive.

What the tool actually does

Before the archaeology, the practical part: this is a data binding tool for JSON, and the fastest way to understand it is to look at what comes out.

You give it a JSON document. It walks the structure, infers a type for every value, works out which shapes are objects and which are lists, and emits a Delphi unit containing one class per object shape — a set of DTOs,

plain data-carrying classes — plus the wiring needed to turn an instance back into JSON and vice versa.

Here is a real example. This unit sits in the repository's smoke-test build folder, so it is the generator's own output rather than something I hand-wrote for the article. It corresponds to a document shaped like this:

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

And this is what the generator produces from it:

```
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;
```

Almost every line of that is something the earlier posts in this series already covered, which is exactly why the project caught my attention.

The `{$M+}` directive and the `published` section are there so the compiler emits runtime type information for those properties — the mechanism behind everything RTTI does. The `[JSONName('items')]` and `[JSONMarshalled(False)]` attributes come from `REST.Json.Types` and are read at runtime by the serializer, which is precisely the attribute-driven pattern from RTTI Part 2. And the base class does the actual work through the RTL:

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

That is `REST.Json` — one of the three ways the RTL speaks JSON. The generator is not replacing Delphi's serializer. It is generating classes shaped so that Delphi's serializer produces the right thing, and filling the gaps where the RTL's defaults would not.

The two gaps it fills for you

There are two details in that generated unit that look like clutter until you know what they are working around.

The first is the double field for lists: a `TArray<TItems>` that carries the JSON name, and a `TObjectList<TItems>` marked `[GenericListReflect]` that your code actually uses. The RTL's serializer handles dynamic arrays cleanly but, left alone, would happily serialize a `TList<T>`'s internal implementation fields as well. So the generator gives you a list as the public API and keeps a plain array as the internal bridge, refreshing one from

the other at serialization time. The repository's own release notes describe the fix as resolving "the matching published property through RTTI, avoiding the internal `TList<T>` implementation data that Delphi's default serializer would otherwise emit."

The second is dates. `TJsonDTO` sets its options to `[joDateIsUTC, joDateFormatISO8601]` in the constructor, and the generator detects ISO 8601 values in your input and types them as `TDateTime` rather than `string`. If you have read the post about the date type JSON doesn't have, you know that this single decision — which

of the two competing conventions you use, and whether you are honest about the time zone — is where most JSON date bugs live.

> **This is generated code, and it acts like it**
> Everything above is emitted fresh each time you generate. If you edit the produced unit by hand and then regenerate from an updated JSON sample, your edits are gone. Treat the JSON document as the source and the unit as a build output, or accept that you have forked it and stop regenerating.

Two repositories with the same name

Now the part that made this worth writing about. The tool has two homes on GitHub, and they are in very different states.

What forking is, and why it is normal

If you have spent your career in Delphi rather than on GitHub, "fork" sounds like something went wrong. It usually hasn't, so it is worth thirty seconds of definition.

Forking is one click. GitHub takes the entire repository — every file, every commit going back to the first one — and creates a copy under your account that you own and can change freely. The original is untouched. Two things then make a fork different from just downloading a ZIP of the source: GitHub permanently records which project yours came from and displays it under your repository's name, and it keeps the shared history, so your changes can still be offered back to the original as a pull request if the original's owner wants them.

People fork for two quite different reasons. The everyday reason is to contribute: you fork, fix a bug, and send the fix back upstream, after which your copy has done its job. The other reason is the one in this story — the original stops moving, someone else wants the tool to keep improving, and their copy quietly becomes the version that everyone should actually be using.

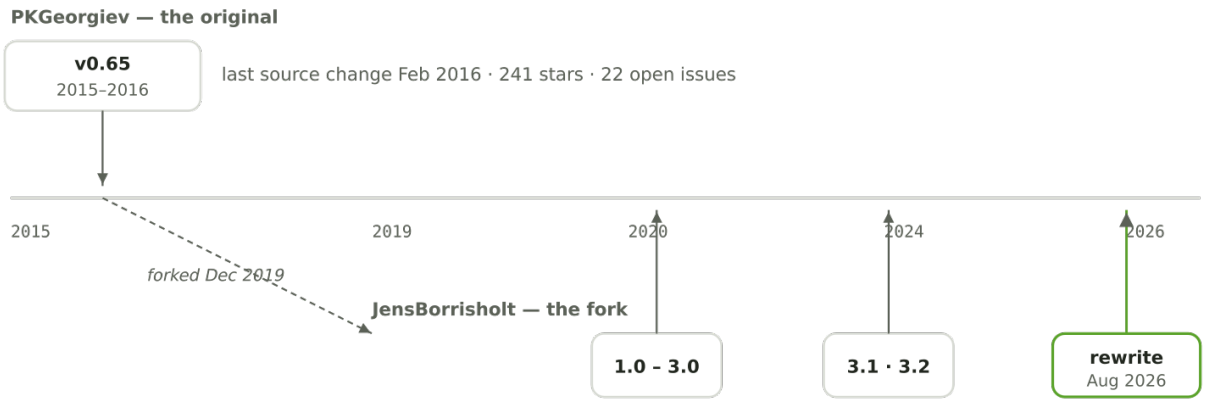
The nearest Delphi analogy is taking a copy of somebody's component source, dropping it into your own library, and maintaining it yourself for the next decade because the author stopped answering email. Every Delphi shop has done it. The difference is that a fork does it in public, with the ancestry recorded and the license making it explicitly allowed — and that permission is not a technicality. It is the reason this whole story is a success rather than a theft.

The two timelines

The original was created by Petar Georgiev in Sofia, Bulgaria, on 3 January 2015 — a FireMonkey desktop app that visualized JSON in a treeview and generated matching Delphi classes, released under the MIT license. It was well received at the time; DelphiABall wrote it up in 2016. Then it stopped. The master branch carries

five commits in total: the initial commit, a README update, a "Version 0.65" release in February 2016, and two commits in October 2017 that added the LICENSE file. As I write this it has 241 stars, 124 forks, and 17 open issues plus five open pull requests, waiting on a maintainer who has moved on.

Jens Borrisholt forked it on 11 December 2019 and has been the one shipping since. Version 1.0 came in January 2020, 2.0 in August 2020, 3.0 that December. Then a four-year gap. Then 3.1 and 3.2 in January 2024. Then another gap — and then, on 3–5 August 2026, a burst of commits that replaced the generation engine entirely and added a second output language. His fork has 80 stars.



The same tool, two repositories: where the stars are and where the commits are

Read that diagram in one direction and it is a story about abandonment. Read it in the other and it is the system working exactly as designed: Petar published under MIT, which is a standing invitation for someone else to pick the work up when he moved on to other things. Jens accepted the invitation. Nothing went wrong here. The only thing that went wrong is that the search engines and the star counts never got the memo.

How to tell which repository is alive

This is worth generalizing, because the Delphi ecosystem is full of this pattern and so is every other small ecosystem. The signals are all on the repository's front page, and none of them is the star count.

> **Thirty seconds on any GitHub repository**
>
> **Read the grey line under the title.** If it says "forked from …", you are on a copy — go look at the parent and compare. **Look at the date on the default branch's last commit**, not the "Updated" timestamp. A push to some abandoned side branch refreshes the latter and tells you nothing. **Open the Releases page.** Tagged releases with dates are the clearest maintenance signal a project gives you. **Check open issues and pull requests.** A stack of both with no maintainer replies for years means nobody is home; zero open issues on an active project usually means someone is actually reading them. **Then look at the forks list**, sorted by recent activity. When the parent is dormant, the successor is almost always sitting in there.

Stars measure how many people once found a project interesting. They are cumulative and they never decay — nobody goes back and un-stars a repository when it stops being maintained. That is not a flaw in GitHub, but it does mean the number at the top of the page answers a question about 2016 and not a question about today.

> **The other side**
>
> I would send someone to the fork today, but the original repository is not worthless and the argument is not one-sided. It is still the canonical URL that everyone links to, it holds the full early history and the issue discussions, and its MIT license is what made everything downstream legal. Forks also go quiet themselves — this one had a four-year gap between 3.0 and 3.1, and a single-maintainer project has a bus factor of one no matter how good last week was. And if you already ship version 3.2 in production, "the engine was rewritten from scratch last Tuesday" is a reason to test carefully before upgrading, not a reason to upgrade tonight.

The two people behind it

Both names deserve more than a footnote, because neither is a household name and both did real work.

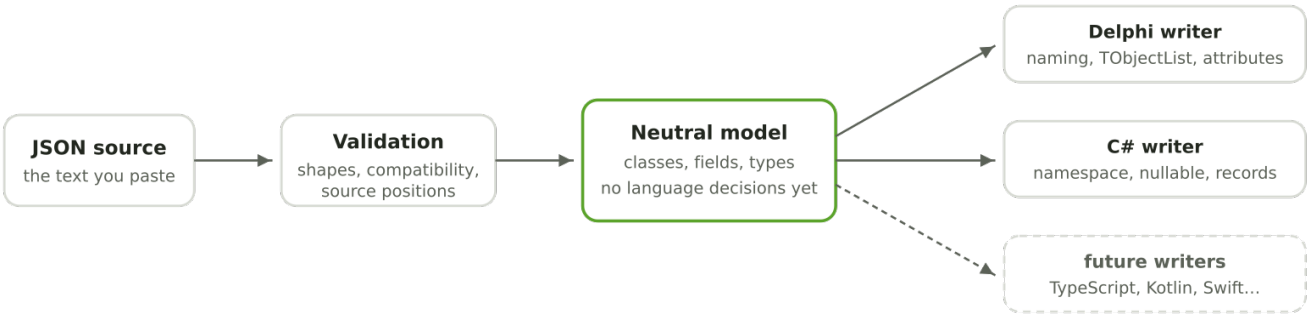
Petar Georgiev is a developer in Sofia, Bulgaria, with 50 public repositories and a blog at pgeorgiev.com. He built the original in 2015 and solved the interesting parts of the problem: walking arbitrary JSON, inferring Delphi types from untyped values, detecting ISO 8601 dates automatically, handling Delphi reserved words in generated identifiers, and generating the destructor code that cleans up nested objects. Everything the tool does today grows out of that design. He then, as far as GitHub shows, moved on. That is an entirely normal end to a side project, and choosing MIT on the way out was the generous decision that let the tool outlive his interest in it.

Jens Borrisholt is a senior software developer in Denmark with around thirty public repositories, and his GitHub reads like someone who builds the thing he needs and then publishes it. There is a port of C#'s `Console` class to Delphi — his most-starred project — object-oriented keyboard and mouse hooks, a Google text-to-speech demo shipped in both Delphi and C#, a small online/offline detection utility, and a JSON interceptors library.

His release history is bursty: a flurry in 2020, silence, a flurry in 2024, silence, a much bigger flurry this month. I want to name that plainly rather than dress it up, because it is the honest shape of most one-person open source. The maintainer has a day job. The project moves when there is time and a reason, and sits still otherwise. Bursty is not the same as abandoned — the difference is whether the bursts keep coming, and here they demonstrably do.

What the August 2026 rewrite actually changed

The rewrite is the reason Jens wrote in, and it is more than a tidy-up. The old code generated Delphi source while it walked the JSON: traversal, validation, type inference, naming and text emission all happened in one stateful pass. The new engine splits those into stages with explicit inputs and outputs — the shape of a small compiler.



One validated model in the middle, and every output language becomes a writer

The point of that middle box is that nothing in it knows what Delphi is. A field remembers its JSON name, its JSON path, whether it was optional or nullable, and where in the source text it came from. A class has an identity that does not depend on whatever identifier a backend eventually gives it. Arrays are recursive rather than carrying a dimension counter, so a matrix is simply "array of array of integer." And crucially, how a value is *represented* in JSON is stored separately from what it *means*: an ISO 8601 timestamp is still recorded as a JSON string, with "this is a date-time" kept alongside it as semantics.

Only after that model is complete does a language backend run. `TDelphiNaming` decides identifiers and escapes reserved words, `TDelphiUnitWriter` chooses `TList<T>` versus `TObjectList<T>` and emits the unit, and neither of them mutates the model.

Two consequences are already visible in the repository. Errors got much better: because validation runs against the source text and keeps character ranges, feeding it `[[1, 2], ["3", "4"]]` reports the expected type, the actual type and the failing JSON path, and the GUI selects the offending text — instead of silently picking a wrong Delphi type. And on 5 August, one day after the neutral model landed, a complete C# backend appeared, with its own naming rules, its own settings for namespaces and nullable value types, and its own writer emitting a full `.cs` file. That is the architecture paying for itself in public.

In my view this was the right call, and the honest reason I believe it is that the C# backend arrived a day later rather than a quarter later. An abstraction that immediately absorbs a second case is an abstraction that was actually needed. That said, a full-engine rewrite is the single riskiest thing you can do to a working tool, and this one is a week old as I write. Jens did cover the old naming and output rules with dedicated compatibility tests, which is the right instinct, but "covered by tests" and "matches what your production code has been parsing for four years" are different claims. Version 4.0 also exists as commits rather than as a tagged release right now, so if you depend on this in a build, pin 3.2 and try 4.0 alongside it before you switch.

The rest of the landscape

Delphi has shipped an XML Data Binding Wizard for decades — point it at a schema or a sample document and it generates interfaces and implementation classes for every node type. The equivalent for JSON never arrived. The RTL gives you the excellent low-level pieces — `System.JSON`, `REST.Json`, and the RTTI needed to drive them, all covered in the JSON series — but nothing that turns a sample document into a typed unit.

The gap is not filled from outside, either. quicktype, the tool most other ecosystems reach for, generates types from JSON for 27 target languages. Delphi and Object Pascal are not among them. So the Delphi community built it, more than once.

> **Other ways to get from JSON to typed Delphi**
>
> marlonnardi/JsonToDelphi is a separate project in the same family, sharing the `Pkg.Json.DTO` runtime lineage and adding VCL, FMX and uniGUI front ends. It is actively pushed and has a friendly following — check the licensing situation before you ship it, as the repository currently carries no license file. Neon by Paolo Rossi solves the adjacent problem beautifully: rather than generating classes, it serializes the classes you already have, with far more control over naming, casing and custom converters than the RTL gives you. It is the serialization engine behind the WiRL REST library, and it is what I would reach for when the Delphi side is the source of truth. I wrote about where it takes over from the RTL in the serialization post. mORMot 2 takes a third route: an extremely fast RTTI-driven JSON layer inside a much larger ORM, SOA and REST toolbox. Bigger commitment, correspondingly more capability. And when the API you are consuming publishes an OpenAPI description, generate from *that* instead of from a sample response — landgraf-dev/openapi-delphi-generator and paolo-rossi/OpenAPI-Delphi both do this. A schema tells you which fields are optional; a sample response can only tell you which fields happened to be present that day.

That last point is the real limit of every JSON-sample-based generator, and it is worth stating clearly rather than burying. Inference from one document is a guess: a field that is null in your sample looks like a string, a field that is absent looks like it does not exist, and an integer that has only ever been small looks like an `Integer` until the day the API sends an ID that needs an `Int64`. The generator can only see what you showed it. If the producer publishes a schema, the schema wins. If it does not — and plenty of real APIs do not — then a sample-based generator saves you an afternoon of typing, and the review of what it inferred is your job, not the tool's.

Takeaways

The tool is genuinely good and, if you consume JSON APIs in Delphi, worth twenty minutes of your time: paste a response, read the generated unit, and you will recognize almost everything in it from the RTL. But the more durable lesson is the one hiding in the grey text under the repository title.

- **The famous repository and the maintained repository are frequently not the same URL.** Check the fork banner, the last commit on the default branch, and the Releases page before you check the star count. Stars only ever go up.
- **A fork is not a schism.** Petar Georgiev built the thing and licensed it MIT; Jens Borrisholt picked it up four years later and has been shipping ever since. That is open source succeeding, not failing.
- **The rewrite was worth doing, and the C# backend is the evidence.** Validate, build a language-neutral model, then emit — the second output language arrived one day after the abstraction did.
- **Generate from a schema when one exists.** Inference from a single sample document is an educated guess about optionality and numeric range, and you own the review.

> Star counts tell you what people found interesting once. Commit dates tell you who is still there.

Finally, and sincerely: thank you, Jens, for getting in touch, and for the years of unpaid weekends behind those release tags. And thank you, Petar, for building it in the first place and for the license that let it keep going. Delphi's RTL leaves gaps like this one, and the only reason they get filled is that people in this community keep quietly filling them and giving the result away. If you are working on something in that spirit, I would like to hear about it.

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 —