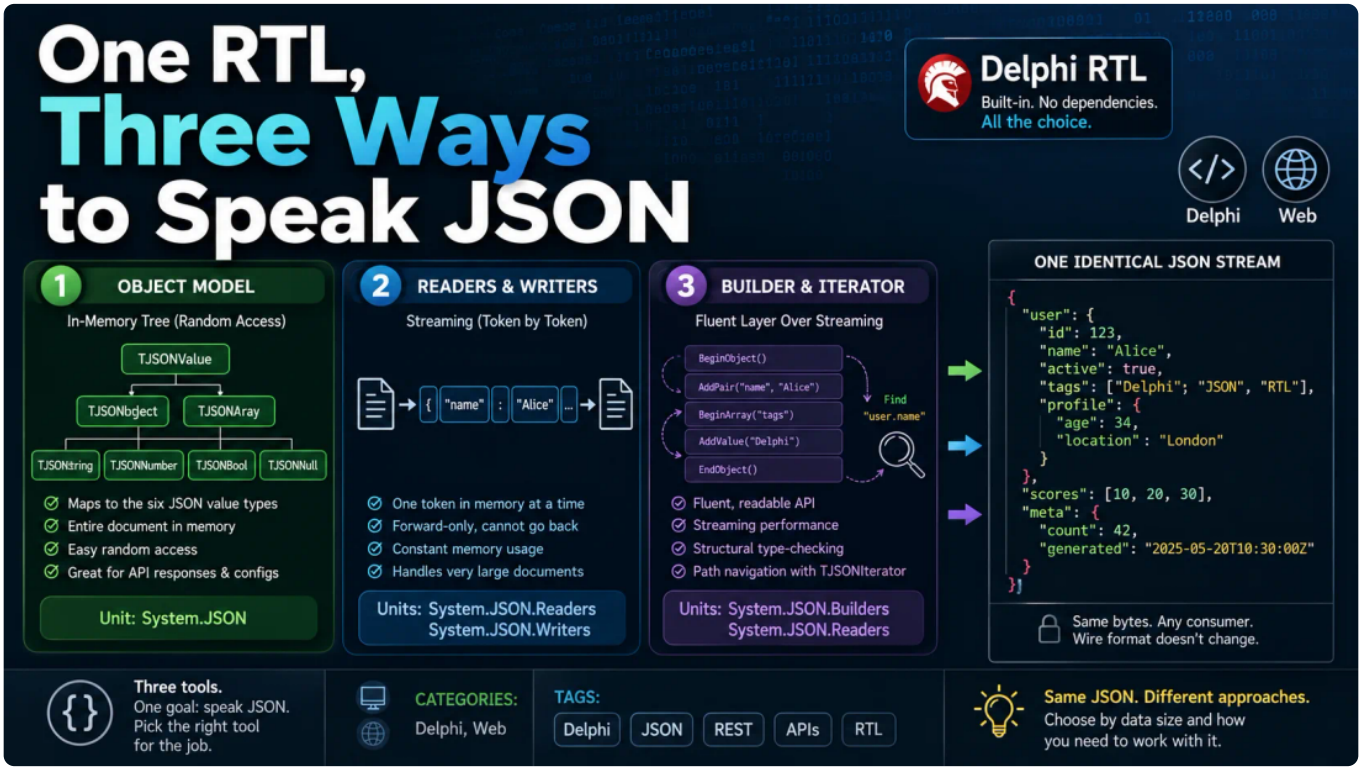
NEWS · JULY 21, 2026

One RTL, Three Ways to Speak JSON

Part three of the JSON series for Delphi developers — the three ways System.JSON lets you read and write JSON, when each one earns its keep, and why REST.Json keeps turning up in code that never asked for it.

By Dr. Holger Flick



In part one we established what JSON actually is — six value types and one recursive rule. In part two we dealt with the type JSON doesn't have, and how to get dates across the wire without shifting anyone's birthday by a day. Both posts were deliberately light on Pascal.

That ends here. This post is about the code, and specifically about a fact that surprises a lot of experienced Delphi developers: the RTL doesn't give you *one* JSON API. It gives you **three**, they live in different units, they have genuinely different strengths, and most people only ever meet the first one.

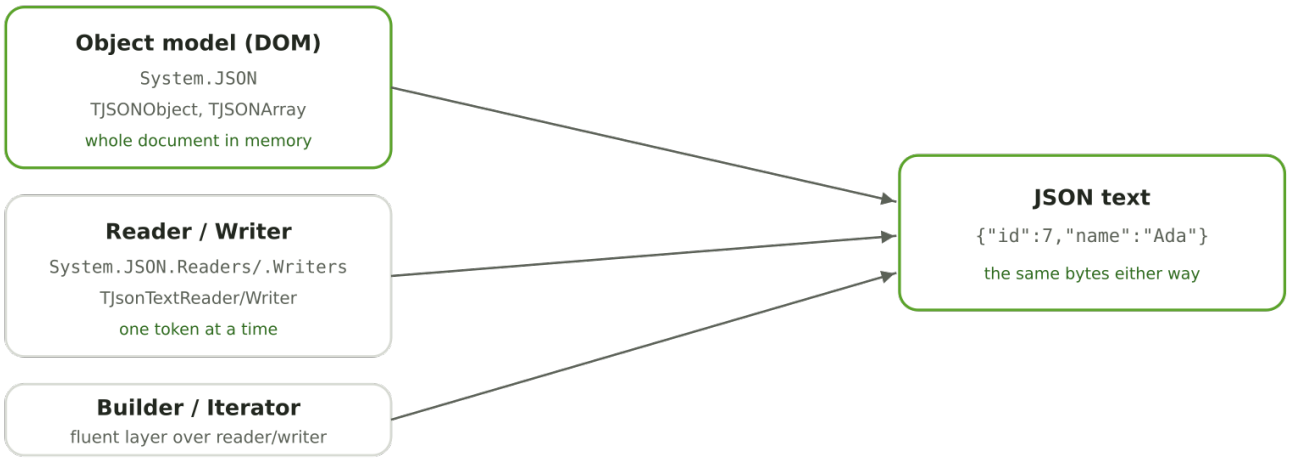
That's usually fine — right up until it isn't. The day someone hands you a 300 MB export and your parse call takes the process down with it, the choice you made without knowing you were making one suddenly costs you an afternoon.

By the end of this post you'll know all three, know which to reach for, and — the part that quietly causes the most confusion in real codebases — know why there's a second, completely separate JSON world called `REST.Json` sitting in your `uses` clause, where it came from, and how to tell the two apart at a glance.

The three frameworks, in one picture

Everything in this post lives in units that ship with Delphi. No downloads, no package manager, nothing to install. The names are worth learning up front, because they're what the documentation indexes on.

Embarcadero's documentation groups them into two named frameworks — the JSON Objects Framework and the Readers and Writers JSON Framework — but the readers-and-writers side really has two distinct faces: the raw token-level reader/writer, and the much friendlier builder/iterator pair layered on top of it. In day-to-day work those feel like three different tools, so that's how I'll treat them.



Three APIs over the same six value types — different shapes, same JSON

Read the diagram left to right: three different programming models on the left, one identical stream of JSON text on the right. Nothing about the output tells a consumer which API produced it — the choice is purely about how *your* code wants to work, and what it can afford in memory. That framing matters, because it means switching frameworks later is a local refactor, not a wire-format change.

Framework 1: the object model

This is the one nearly everyone learns first, and for good reason — it maps directly onto the mental model from part one.

The `System.JSON` unit gives you a class per value type, and they mirror the six types exactly: `TJSONObject`, `TJSONArray`, `TJSONString`, `TJSONNumber`, `TJSONBool`, and `TJSONNull`. They all descend from `TJSONValue`, which is what lets a container hold any of them — the recursive rule from part one, expressed as a class hierarchy.

The trade is stated plainly in the name of the alternative: the object model builds **the entire document as objects in memory**, all at once.

Building JSON

`AddPair` has overloads for the common Delphi types — `string`, `Integer`, `Int64`, `Double`, `Boolean`, and more — so simple values need no wrapper class at all:

```
uses
  System.JSON;

var
  LRoot: TJSONObject;
  LTags: TJSONArray;
begin
  LRoot := TJSONObject.Create;
  try
    LRoot.AddPair('id', 7);                    // Integer overload
    LRoot.AddPair('name', 'Ada Lovelace');     // string overload
    LRoot.AddPair('active', True);             // Boolean overload

    LTags := TJSONArray.Create;
    LTags.Add('admin');
    LTags.Add('author');
    LRoot.AddPair('tags', LTags);              // LRoot now owns LTags

    Writeln(LRoot.ToJSON);
    // {"id":7,"name":"Ada Lovelace","active":true,"tags":["admin","author"]}
  finally
    LRoot.Free;
  end;
end;
```

The ownership comment on the `AddPair('tags', …)` line is the thing to internalise. **Adding a value to a container transfers ownership of it.** Freeing `LRoot` frees the array and every string inside it, so the single `try…finally` around the root is correct and complete. Freeing `LTags` yourself as well would be a double-free.

> **ToJSON versus Format**
> `ToJSON` produces compact output — no spaces, no line breaks — which is what you want on the wire. When you need something a human can read in a log or a test failure, `Format` gives you indented output: `LRoot.Format(2)` indents by two spaces per level. Both are declared on `TJSONAncestor`, so they work on any node, not just the root.

Parsing JSON

Parsing is a class function on `TJSONObject`, and its default behaviour has a sharp edge worth knowing before you meet it in production:

```
class function ParseJSONValue(const Data: string; UseBool: Boolean = False;
  RaiseExc: Boolean = False): TJSONValue; overload; static;
```

Look at `RaiseExc: Boolean = False`. **By default, malformed JSON does not raise an exception — it returns `nil`.** That's a defensible design (parsing network input shouldn't blow up your call stack unasked), but it means the check is on you, and skipping it converts a clear "bad JSON" error into an access violation somewhere further down.

```
var
  LValue: TJSONValue;
  LRoot: TJSONObject;
  LName: string;
  LId: Integer;
begin
  LValue := TJSONObject.ParseJSONValue(LResponseBody);
  if LValue = nil then
    raise Exception.Create('Response was not valid JSON');
  try
    if not (LValue is TJSONObject) then
      raise Exception.Create('Expected a JSON object at the root');
    LRoot := TJSONObject(LValue);

    LName := LRoot.GetValue<string>('name', '(unknown)');
    LId   := LRoot.GetValue<Integer>('id', 0);
  finally
    LValue.Free;    // the parser hands you ownership of the tree
  end;
end;
```

Two details carry real weight here. The **root of a JSON document need not be an object** — `[1,2,3]` and even `42` are valid JSON documents, so `ParseJSONValue` returns the base `TJSONValue` and the `is TJSONObject` test is genuine defensive code, not ceremony. And `ParseJSONValue` **transfers ownership to you**: the returned tree is yours to free, exactly once, at the root.

That generic `GetValue<T>` overload taking a default is the most pleasant thing in the whole unit, and it's underused:

```
function GetValue<T>(const APath: string; ADefaultValue: T): T; overload;
```

It handles the missing-key case and the type conversion in one call, which collapses the usual four-line dance of "find it, check for nil, check the type, read it" into something you can actually read.

Paths, for nested documents

Real API responses nest, and walking them level by level gets verbose fast. `FindValue` accepts a path expression, so you can reach into a structure in one step:

```
// {"user":{"name":"Ada","roles":["admin","author"]}}
LFirstRole := LRoot.GetValue<string>('user.roles[0]', '');
```

Dots step into objects, `[n]` indexes arrays. `FindValue` returns `nil` when any part of the path is missing rather than raising, which makes it a good fit for optional fields in a response you don't fully control.

> **Duplicate keys are legal JSON, and this is where implementations differ**
>
> RFC 8259 says object names "SHOULD be unique" — a recommendation, not a requirement — and goes on to note that implementations vary in how they handle duplicates. It's explicit that behaviour is "unpredictable" for software that doesn't agree on a rule. So `{"a":1,"a":2}` is valid JSON, and what any given parser does with it is a local decision, not a portable guarantee. Never *emit* duplicate keys, and if you consume a feed that does, test your parser's actual behaviour rather than assuming.

Framework 2: readers and writers

The second framework exists because of the sentence I flagged above: the object model holds everything in memory. Embarcadero's own documentation for the readers and writers framework makes the pitch directly — it lets you read and write JSON "directly to a stream, without creating a temporary object," which it credits with "better performance and improved memory consumption."

The classes live in `System.JSON.Readers` and `System.JSON.Writers`. `TJsonReader` and `TJsonWriter` are the abstract bases; the concrete text implementations are `TJsonTextReader` and `TJsonTextWriter`, which work against a `TTextReader`/`TTextWriter`.

The model is **token at a time**. Instead of a tree, you get a cursor moving through a sequence of events — start object, property name, string value, end object — and you react to each one.

The reader never holds the document — only the current token

The key property is in the caption: only one token is live at a time. Memory use is proportional to the *deepest nesting level*, not the document size — which is why this framework can stream a multi-gigabyte file through a program that never allocates more than a few kilobytes for it. The cost is equally visible in the diagram: the arrow points one way. You cannot go back, and you cannot ask "what's in the `tags` field" until you arrive at it.

Writing with `TJsonTextWriter` is a sequence of explicit structural calls:

```
uses
  System.JSON.Writers, System.JSON.Types, System.IOUtils, System.Classes;

var
  LStream: TStreamWriter;
  LWriter: TJsonTextWriter;
begin
  LStream := TStreamWriter.Create('out.json', False, TEncoding.UTF8);
  try
    LWriter := TJsonTextWriter.Create(LStream);
    try
      LWriter.Formatting := TJsonFormatting.Indented;

      LWriter.WriteStartObject;
      LWriter.WritePropertyName('id');
      LWriter.WriteValue(7);
      LWriter.WritePropertyName('name');
      LWriter.WriteValue('Ada Lovelace');
      LWriter.WritePropertyName('tags');
      LWriter.WriteStartArray;
      LWriter.WriteValue('admin');
      LWriter.WriteValue('author');
      LWriter.WriteEndArray;
      LWriter.WriteEndObject;
    finally
      LWriter.Free;
    end;
  finally
    LStream.Free;    // flushes to disk
  end;
end;
```

Every brace and bracket is a method call you make yourself. That's more typing than `AddPair`, and it's easy to get the nesting wrong in a long procedure — but nothing was ever held in memory, and the bytes went to disk as they were produced.

Framework 3: the builder and the iterator

This is the one people forget exists, and it's the one that makes the second framework pleasant to use. `System.JSON.Builders` provides a **fluent layer over the reader and writer** — the same streaming behaviour, in code that reads like the JSON it produces.

`TJSONObjectBuilder` wraps a writer. Every `Add` returns the collection you're building, so calls chain:

```
uses
  System.JSON.Builders, System.JSON.Writers, System.Classes, System.SysUtils;

var
  LStream: TStringWriter;
  LWriter: TJsonTextWriter;
  LBuilder: TJSONObjectBuilder;
begin
  LStream := TStringWriter.Create;
  try
    LWriter := TJsonTextWriter.Create(LStream);
    try
      LBuilder := TJSONObjectBuilder.Create(LWriter);
      try
        LBuilder
          .BeginObject
            .Add('id', 7)
            .Add('name', 'Ada Lovelace')
            .Add('active', True)
            .BeginArray('tags')
              .Add('admin')
              .Add('author')
            .EndArray
            .BeginObject('address')
              .Add('city', 'Turin')
              .Add('zip', '10121')
            .EndObject
          .EndObject;
      finally
        LBuilder.Free;
      end;
      Writeln(LStream.ToString);
    finally
      LWriter.Free;
    end;
  finally
    LStream.Free;
  end;
end;
```

The indentation of the Pascal mirrors the structure of the JSON, which makes a nesting mistake visible on the page rather than at runtime. `BeginArray` returns a `TElements` (things without keys), `BeginObject` returns a `TPairs` (things with keys), and `EndArray`/`EndObject` walk you back up — so the compiler enforces that you don't add a keyed pair inside an array. **You get streaming performance with structural type-checking**, which is a genuinely nice combination and the reason I reach for the builder over the raw writer almost every time.

On the reading side, `TJSONIterator` wraps a `TJsonReader` and adds navigation the raw reader doesn't have:

```
uses
  System.JSON.Builders, System.JSON.Readers, System.Classes;

var
  LReader: TJsonTextReader;
  LIter: TJSONIterator;
  LStringReader: TStringReader;
begin
  LStringReader := TStringReader.Create(LJsonText);
  try
    LReader := TJsonTextReader.Create(LStringReader);
    try
      LIter := TJSONIterator.Create(LReader);
      try
        if LIter.Find('user.name') then
          Writeln(LIter.AsString);
      finally
        LIter.Free;
      end;
    finally
      LReader.Free;
    end;
  finally
    LStringReader.Free;
  end;
end;
```

`Find` takes the same dotted path syntax as the object model's `FindValue` — the RTL source documents it as "a string consisting of pair names and array indexes, separated by dots," with `'entities.urls[0].indices[1]'` as its own example. So you get path-style access *without* materialising the document, which is exactly the right tool for pulling three fields out of a large response.

> **The iterator is shallow by default**
>
> This is the detail that trips people up, and it's documented but easy to miss. `TJSONIterator` does not descend into nested structures on its own: when it reaches a `StartObject` or `StartArray` token, you must call `Recurse` before the next `Next`, or the iterator **skips the entire object or array**. `Return` walks back out to the parent. It's forward-only navigation with an explicit stack you drive yourself — powerful, but you have to mean it. (`Rewind` exists, but only works if the underlying stream can actually be rewound.)

Choosing between them

Here's how I decide, and I'll be honest that the first option covers the large majority of real work.

Use the object model when the document fits comfortably in memory and you need random access.

API responses, config files, webhook payloads — anything measured in kilobytes or a few megabytes. Being able to poke at `LRoot.GetValue<string>('user.name')` in any order is worth a great deal, and the memory cost is irrelevant at that size. This is the default, and choosing it deliberately is fine.

Use the builder when you're generating JSON, especially large or streamed output. It costs you nothing in readability over the object model — arguably it's *more* readable — and it never builds a tree. If you're writing a JSON export of a dataset, this is the right call from the start.

Use the reader or iterator when the input is large, or arrives as a stream. A multi-hundred-megabyte export, a log file, a response you're processing as it downloads. The object model would need the entire thing resident, plus the object overhead on top of the raw text.

> **The size question, honestly**
> I want to be careful not to overstate the memory point, because "use streaming for performance" gets repeated far more often than it gets measured. For a typical REST payload the object model's overhead is real but completely irrelevant — you will not notice it, and reaching for a streaming API to save a few hundred kilobytes is a poor trade against the readability you give up. The streaming frameworks earn their keep at a scale most applications never reach. My advice: default to the object model, use the builder for generating output because it's pleasant regardless, and switch to the reader when you have an actual measured problem or you know for certain the input is unbounded. Premature streaming is as real a mistake as premature optimisation generally.

The other JSON in your uses clause

Now to the thing that causes more confusion than anything else in this area, and it's not really about the three frameworks at all.

If you've worked in a Delphi codebase of any age, you've seen this:

```
uses
  System.JSON, REST.Json;
```

Two units. Both about JSON. Both shipped by Embarcadero. And code that mixes them in ways that look arbitrary until you know the history.

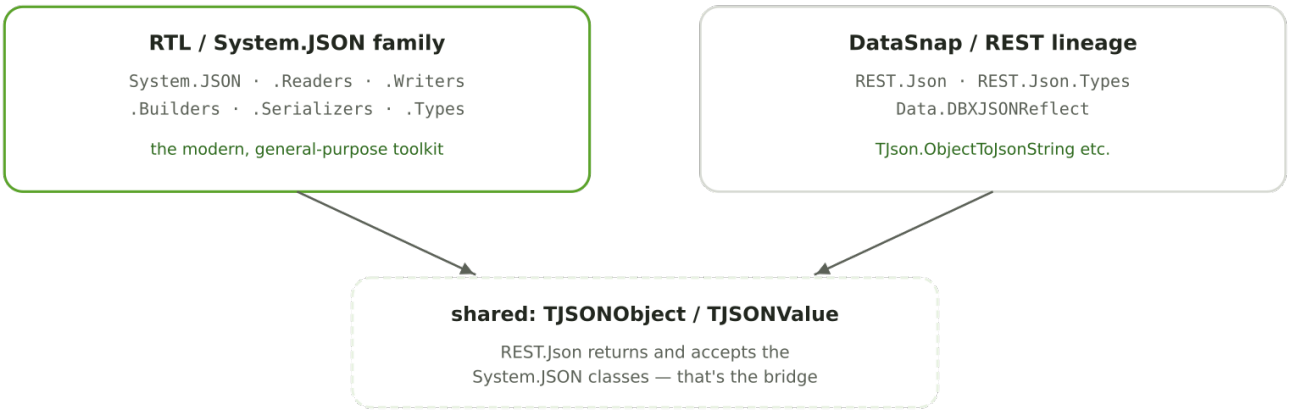
`REST.Json` **comes from a different lineage.** Its ancestry is in DataSnap — Embarcadero's multi-tier framework — and you can still see it in the RTL source today: `REST.Json` is implemented on top of `TJSONMarshal`/`TJSONUnMarshal` from `Data.DBXJSONReflect`, the DBX (DataSnap/dbExpress) reflection marshaller. It was the answer to "how do I turn an object into JSON" years before `System.JSON.Serializers` existed.

What it gives you is one very convenient class, `TJson`, with class methods that do the whole job in a line:

```
uses
  REST.Json;

LJsonText := TJson.ObjectToJsonString(LPerson);
LPerson   := TJson.JsonToObject<TPerson>(LJsonText);
```

That is genuinely useful, and it's why the unit is everywhere. Object serialization is the subject of the next post, so I won't go deeper here — the point for now is simply **knowing which world a type belongs to**.



Two lineages, one shared value model — where the overlap actually is

The dashed box explains why mixed code isn't actually broken. `TJson.ObjectToJsonObject` returns a `System.JSON.TJSONObject`; `TJson.JsonToObject<T>` accepts one. The two worlds meet at the value classes, so a procedure that takes a `TJSONObject` doesn't care which unit produced it. Mixed code compiles and works — it just *looks* incoherent to whoever reads it next.

There's one more wrinkle that explains a lot of the confusion, and it's visible directly in the RTL source. `REST.Json.Types` declares:

```
JSONNameAttribute = System.JSON.Types.JsonNameAttribute;
```

It's an **alias**. The attribute you apply as `[JSONName('user_id')]` from `REST.Json.Types` *is the same type* as `JsonNameAttribute` from `System.JSON.Types`. So the two worlds don't just interoperate at the value level — some of their attributes are literally identical, under two different names. If you've ever wondered why the same attribute seems to work no matter which unit you imported, that's why.

> **A rule of thumb for new code**
>
> My preference, and I'll flag it as a preference rather than a rule: for new code, **pick the `System.JSON.*`*
>
> `family and stay in it**. It's the actively-developed line,` System.JSON.Serializers `(the subject of the next post) is the modern serialization story, and staying in one namespace makes the code legible to the next person. That said — I'd push back firmly on anyone who reads that as "rip` REST.Json `out of working code."` TJson.ObjectToJsonString `is a genuinely good one-liner, it's stable, and there is no prize for a refactor that changes no behaviour. The real cost isn't using either unit; it's using both *without knowing why*. Where this preference is weakest is DataSnap codebases, where` REST.Json` is the idiom the rest of the code already speaks — there, consistency with the surrounding code beats my namespace tidiness every time.

What the rest of the world does

Worth a moment of perspective, because this three-way split isn't a Delphi quirk — it's the standard shape of a mature JSON library, and recognising the pattern makes other stacks instantly readable.

.NET's `System.Text.Json` has exactly this trio: `JsonDocument` (DOM), `Utf8JsonReader`/`Utf8JsonWriter` (streaming), and `JsonSerializer` (objects). Java's Jackson names them outright — the tree model, the streaming API, and data binding. Go's `encoding/json` offers `Unmarshal` into a map and `json.Decoder` for streams. The same three answers, because there are only three sensible answers to "how do I want to touch this document."

In the Delphi world specifically, the built-in frameworks aren't your only option, and two open-source alternatives are worth knowing. JsonDataObjects by Andreas Hausladen (MIT) is a single-unit DOM parser with a reputation for being very fast and very pleasant — its API is arguably nicer than `System.JSON` for everyday work, and being one file makes it trivial to adopt. mORMot 2 (MPL/GPL/LGPL tri-license) includes an extremely fast JSON engine as part of a much larger framework. Both are actively maintained and widely used. The RTL's advantage is simply that it's already there, with no dependency to justify to anyone — which for a lot of teams is the deciding factor, and for others isn't.

Takeaways

Three frameworks sounds like a lot to hold in your head, but the split is principled: they differ in whether the document lives in memory, and whether you can move around it freely.

- **The object model (`System.JSON`) is the sensible default.** `TJSONObject` and friends mirror the six value types directly, and `GetValue<T>` with a default is the most useful method in the unit. Remember that containers own what you add to them, and free only the root.
- `ParseJSONValue` **returns** `nil` **on bad input by default.** `RaiseExc` is `False` unless you say otherwise. Check the result, and check that the root is the type you expected — a valid JSON document can be an array or a bare number.
- **The builder is the nicest way to* write *JSON.** Fluent, streaming, and structurally type-checked, with none of the object model's memory cost. There's little reason not to use it for generated output.
- **Readers and iterators are for scale.** Constant memory, forward-only, path-capable via `TJSONIterator.Find` — and remember `Recurse`, or you'll silently skip whole branches. Reach for them when you have a real size problem, not on principle.
- `REST.Json` **is a separate lineage, not a rival.** It came from DataSnap, it's built on the DBX marshaller, and it meets `System.JSON` at the shared value classes — which is why mixed code works. Prefer one namespace in new code; don't wage war on the other in old code.

> The RTL gives you a tree, a token stream, and a fluent layer over the token stream. Pick by how much data you have and how freely you need to move through it — everything else is a matter of taste.

Next time we take on the question all of this has been circling: **serialization**. What it actually means to turn an object into data and back again, why the round trip is harder than it looks, what Delphi's RTTI-based serializers can do for you out of the box — and where a dedicated library like Paolo Rossi's Neon takes over.

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 —