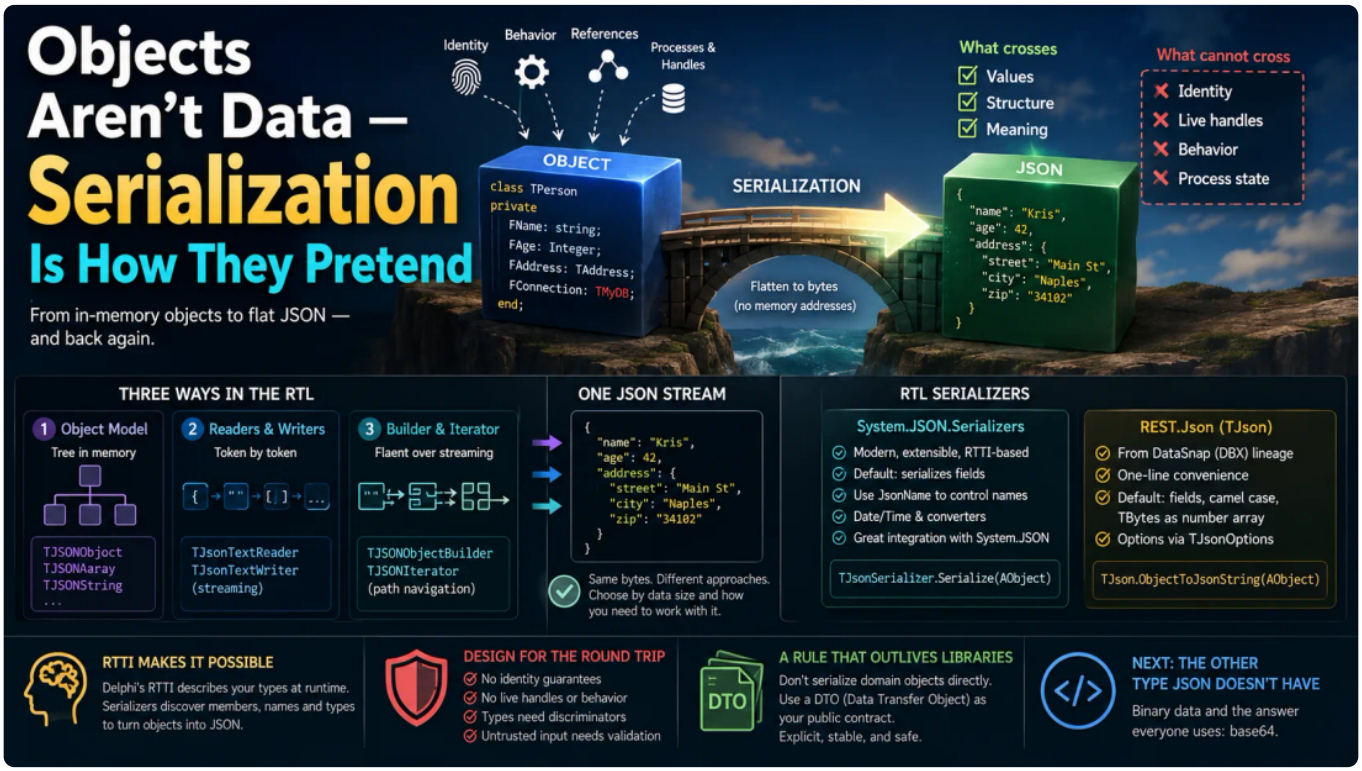
NEWS · JULY 22, 2026

Objects Aren't Data — Serialization Is How They Pretend

Part four of the JSON series for Delphi developers — what serialization and deserialization actually mean, why the round trip loses more than you expect, how to do it with the RTL's RTTI-based serializers, and where Neon takes over.

By Dr. Holger Flick



Every post in this series so far has had you building JSON by hand. In part three we walked through all three ways the RTL lets you do it — the object model, the reader/writer pair, the fluent builder — and every one of them had you naming each field, one at a time, in code.

That works, and for small payloads it's honest and clear. But you have a `TPerson` class with twelve fields, and you're writing twelve `AddPair` calls that essentially restate the class declaration in a different syntax. Then someone adds a thirteenth field, forgets the serializer, and a bug ships.

The obvious wish is: *just turn my object into JSON*. That wish has a name — **serialization** — and Delphi can do it. But before we write a single line of it, I want to spend real time on what the word actually means, because the wish contains a hidden assumption that is not true, and almost every serialization bug I've seen traces back to it.

The assumption is that an object *is* data. It isn't. An object is data plus identity plus behaviour plus references to other objects plus, quite often, a handle to something that only exists while your process is running. Serialization is the act of deciding which of that was ever really data — and then admitting that the rest can't come along.

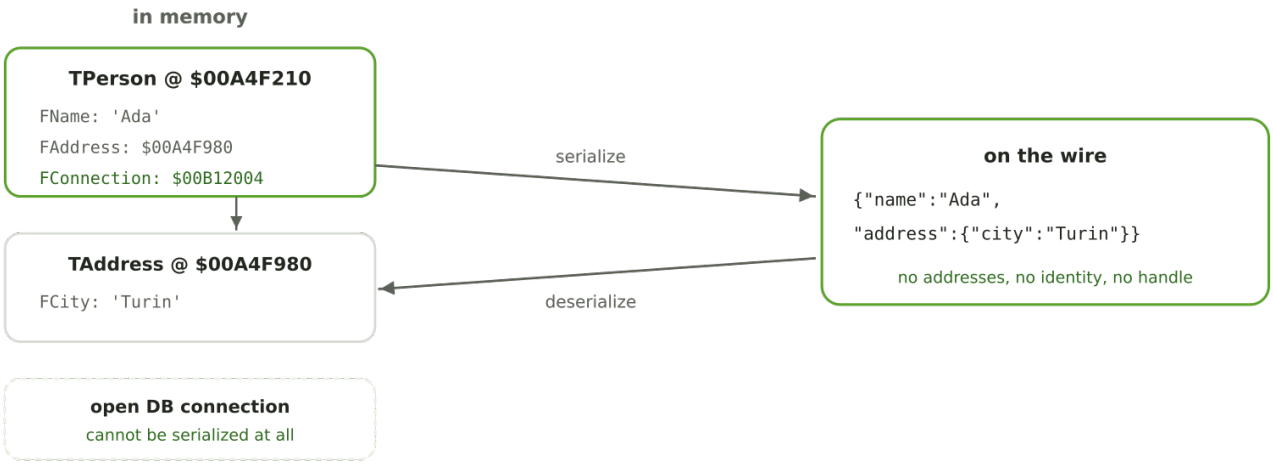
What serialization actually is

Let's define it properly, because the plain-language version is genuinely useful.

Serialization is converting a structure that lives in your program's memory into a flat sequence of bytes that can be stored or transmitted. **Deserialization** is the reverse: taking those bytes and reconstructing a structure in memory from them.

The word "flat" is doing the heavy lifting. In memory, your object is a region of RAM at some address, containing values and — crucially — *pointers* to other regions of RAM. That web of addresses is meaningful only inside your running process. Write those addresses to a file and they become nonsense: the memory they refer to belongs to a process that has since exited, on a machine that may not even be the one reading the file.

So serialization is fundamentally about **breaking the dependency on your process's memory layout**. Whatever comes out the other side has to stand on its own.



Memory has addresses and identity; the wire has neither

Three things happened crossing that gap, and each one is a decision someone had to make. `FAddress` was a pointer; it became a **nested object**, because the thing it pointed to was itself data. `FName` came across untouched, because a string is already data. And `FConnection` — the dashed box — simply **could not go**.

A live database handle has no meaning outside the process that opened it. The serializer's only options are to skip it or to fail, and a good one lets you say which.

That last case is the one worth sitting with. There is no clever encoding that saves it. Some state is genuinely not data, and part of designing a serializable class is knowing which parts those are.

Why the round trip isn't symmetric

It's tempting to think of deserialization as serialization run backwards. It isn't, and the asymmetry causes real bugs.

When you serialize, you have everything: the object, its actual runtime type, every value. When you deserialize, you have **text and a hope**. You must decide what class to instantiate before you've finished reading the data that would tell you. You must handle fields that are absent, fields you've never heard of, and values of the wrong type. Serialization is a total function over valid objects; deserialization is a partial function over arbitrary input.

Two specific losses are worth naming because they surprise people:

Object identity disappears. If your structure has two fields pointing at the *same* `TAddress` instance, memory knows that — one object, two references. Standard JSON has no way to express it. Serialize and you get the address written out twice; deserialize and you get two separate objects that happen to be equal. Mutate one afterwards and the other doesn't change, and a piece of your program's logic quietly stops working. Cyclic references are the same problem in a sharper form: a naive serializer follows the cycle until the stack runs out. (Delphi's `TJsonSerializer` has a `ReferenceResolver` hook for exactly this class of problem, and Neon's docs are candid that circular references need care. Neither is a free win — the honest advice is to design object graphs you intend to serialize as trees.)

Type identity is not carried by default. If a field is declared as `TAnimal` and holds a `TDog`, the JSON records the *values*, not the class name. Deserialize into a `TAnimal` field and you get whatever the deserializer was told to build. Recovering true polymorphism requires a type discriminator written into the payload — a `"$type"` field or similar — which is an explicit design choice, not something you get for free.

> **This is not a JSON limitation, and not a Delphi one**
> Every serialization format faces this. Java's built-in serialization famously preserves object identity and full type information — and paid for it with a long history of security problems, because deserializing untrusted data that names its own classes lets an attacker choose which code runs. Oracle's own secure coding guidelines treat deserialization of untrusted input as a serious risk, and the OWASP community tracks it as insecure deserialization. JSON's inability to name types is, from that angle, a feature: a JSON deserializer only builds classes *you* named in your code. Losing identity and polymorphism is the price of a format that can't be tricked into constructing arbitrary objects.

How Delphi finds out what's in your class

Before the code, one mechanism needs explaining, because everything that follows depends on it and it's the reason this works at all.

For a serializer to turn *any* object into JSON, it has to discover — at runtime, with no knowledge of your class at compile time — what fields and properties that object has, what they're named, and what types they are. The facility that makes this possible is RTTI, Run-Time Type Information: metadata the compiler embeds in your executable describing your types, which `System.Rtti` lets you query.

RTTI is the bridge — the serializer never saw your class at compile time

The middle box is the whole trick, and it has a consequence you should feel in your stomach. **Your field and property names become part of your wire format.** Rename a private field for internal tidiness, and if the

serializer is using fields, you have just changed the JSON your API emits. The compiler will not warn you. This is why the naming and attribute options in the next sections aren't cosmetic — they're the difference between a wire contract you control and one that changes whenever someone refactors.

Serializing with the RTL

Delphi ships **two** object serializers, from different eras, and part three already introduced the split. Both work.

Their defaults do not agree, and that surprise is worth meeting on purpose rather than in production.

Take this class for both examples:

```
type
  TPerson = class
  private
    FName: string;
    FAge: Integer;
  public
    property Name: string read FName write FName;
    property Age: Integer read FAge write FAge;
  end;
```

The modern one: System.JSON.Serializers

`TJsonSerializer` is the current, actively-developed serializer, and it's part of the `System.JSON.*` family from part three:

```
uses
  System.JSON.Serializers;

var
  LSer: TJsonSerializer;
  LPerson: TPerson;
  LJson: string;
begin
  LPerson := TPerson.Create;
  LSer := TJsonSerializer.Create;
  try
    LPerson.Name := 'Ada Lovelace';
    LPerson.Age := 36;

    LJson := LSer.Serialize<TPerson>(LPerson);
    Writeln(LJson);
    // {"FName":"Ada Lovelace","FAge":36}     <-- note the field names
  finally
    LSer.Free;
    LPerson.Free;
  end;
end;
```

Look at that output. `FName`**, not** `Name`**.** The default member serialization is `TJsonMemberSerialization.Fields`,

so you get the private backing fields, `F` prefix and all. That is almost never the JSON you want on a public API, and it's the single most common surprise with this class.

Two attributes fix it. `JsonSerializeAttribute` changes what gets serialized, and `JsonNameAttribute` renames a member on the wire:

```
uses
  System.JSON.Serializers, System.JSON.Types;

type
  [JsonSerialize(TJsonMemberSerialization.&Public)]
  TPerson = class
  private
    FName: string;
    FAge: Integer;
  public
    [JsonName('name')]
    property Name: string read FName write FName;
    [JsonName('age')]
    property Age: Integer read FAge write FAge;
  end;

// now: {"name":"Ada Lovelace","age":36}
```

That's the shape worth adopting as a habit. **Naming every serialized member explicitly with `JsonName` decouples your wire format from your Pascal identifiers** — rename the property freely afterwards and the JSON doesn't move. It's a few lines of attribute noise that buys you a stable contract.

`JsonIgnoreAttribute` excludes a member entirely — the answer to the database-handle problem from the opening diagram:

```
[JsonIgnore]
property Connection: TDBConnection read FConnection write FConnection;
```

Deserialization is the mirror call, and note that you get a fully constructed object back:

```
LPerson := LSer.Deserialize<TPerson>(LJson);   // caller owns the result
```

The serializer also exposes the date handling that part two was all about — `DateFormatHandling` (`Iso`, `Unix`, or `FormatSettings`) and `DateTimeZoneHandling` (`Local` or `Utc`). Setting these once on the serializer is exactly the "decide UTC in configuration, not at every call site" discipline part two argued for.

The older one: REST.Json

`TJson` from `REST.Json` is the DataSnap-lineage serializer, and its appeal is undeniable brevity:

```
uses
  REST.Json;

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

One line each. No serializer instance to create and free. For a quick internal tool this is hard to argue with, and it explains why the unit is everywhere.

Its behaviour is driven by a `TJsonOptions` set, and its defaults are declared in the RTL source as:

```
const CDefaultOptions = [joDateIsUTC, joDateFormatISO8601,
  joBytesFormatArray, joIndentCaseCamel, joSerialFields];
```

That constant repays a close read, because three of those five defaults have real consequences. `joSerial-Fields` means it too serializes **fields** by default — same surprise as above, different unit. `joIndentCaseCamel`

means it *camel-cases* the names it emits, so `FName` becomes `fName` rather than `FName`; the two serializers therefore produce **different JSON for the same class out of the box**. And `joBytesFormatArray` means `TBytes` fields are written as a JSON **array of numbers** — `[137,80,78,71,…]` — rather than base64. For a one-megabyte image that's several megabytes of digits and commas. `joBytesFormatBase64` switches it, and the whole question of binary data in JSON is where this series goes next.

> **Two serializers, two sets of defaults, one class**
> This is the practical hazard. Serialize the same `TPerson` with `TJsonSerializer` and with `TJson` and you get `{"FName":…}` from one and `{"fName":…}` from the other. Neither is wrong; they were designed years apart with different conventions. If a codebase uses both — and mixed codebases are common, for the historical reasons in part three — then which unit a given call sits in silently determines the wire format. **Pick one serializer per project and be explicit about names.** That single decision removes an entire category of confusion.

Where a dedicated library earns its place

The RTL serializers are capable and free, and for a lot of internal work they're genuinely enough. But there's a gap, and it shows up the moment your JSON has to match a contract you didn't write.

Public APIs have naming conventions — `snake_case` is extremely common — and the RTL gives you a per-member `JsonName` attribute and little else. Match a fifty-field `snake_case` API and you're writing fifty attributes by hand, each one an opportunity for a typo that produces a silently-missing field.

Delphi Neon by Paolo Rossi (Apache-2.0) is built around that problem. It's an RTTI-based serializer like the RTL's, but the configuration is the point: naming conventions, visibility, and type handling are set **once, as policy**, rather than annotated per member.

```
uses
  Neon.Core.Persistence.JSON, Neon.Core.Types;

var
  LConfig: INeonConfiguration;
  LJson: TJSONValue;
begin
  LConfig := TNeonConfiguration.Default
    .SetMemberCase(TNeonCase.SnakeCase)
    .SetMembers(TNeonMembers.Properties)
    .SetIgnoreFieldPrefix(True);

  LJson := TNeon.ObjectToJSON(LPerson, LConfig);
  try
    Writeln(TNeon.Print(LJson, True));
    // {"name": "Ada Lovelace", "birth_date": "1815-12-10"}
  finally
    LJson.Free;
  end;
end;
```

That `SetMemberCase(TNeonCase.SnakeCase)` line is the whole argument. **One statement makes every member in every class match the convention**, and it keeps matching as the classes grow. Neon supports `SnakeCase`, `camelCase`, `PascalCase`, `UPPERCASE` and `lowercase`, plus `SetMembers` to choose fields or properties

and `SetVisibility` to control which visibility levels participate — the things the RTL makes you decide per class, or not at all.

Deserialization takes an existing instance, which fits Delphi's ownership model cleanly:

```
LJson := TJSONObject.ParseJSONValue(LText);
try
  TNeon.JSONToObject(LPerson, LJson, LConfig);   // populates LPerson
finally
  LJson.Free;
end;
```

Two more things earn it a place in real projects. Its **type coverage is broader** — records, dynamic arrays, generic `TObjectList<T>`, dictionaries with string keys, and streamable classes, where the RTL's support for generic collections is thinner. And it keeps the `TDate`/`TDateTime` distinction that part two showed the RTL flattening, with a `UseUTCDate` option that settles the timezone question once in configuration.

Where I'd use each — and where Neon is the wrong call

In my view, if you're building or consuming a real REST API with a naming convention you don't control, Neon pays for itself almost immediately, and I'd reach for it by default in that situation. But I want to scope that properly, because "use the library" is not automatically the right answer. Neon is a dependency: it's another repository to track, another thing to update when a new Delphi version lands, and another thing a future maintainer has to learn. If you're serializing three small classes for an internal tool, `TJson.ObjectToJsonString` is one line and adds nothing to your build — using it is the *better* engineering decision there, not the lazy one. The RTL serializers are also the right call when you're shipping a library others consume, where every dependency you take becomes one they inherit. And it's worth saying plainly that the RTL's `TJsonSerializer` is a competent, extensible piece of work with a proper converter architecture — it's not a toy that Neon replaces, it's a lower-level tool with a different emphasis. The question isn't which is better; it's whether your JSON has to match someone else's conventions.

The wider landscape

Neon isn't the only choice, and the alternatives are worth knowing. mORMot 2 includes serialization as part of a very fast, very large framework — the right call if you're already using it, a lot to adopt if you aren't. JsonDataObjects deliberately stops at the DOM and doesn't do object mapping, which is a legitimate design choice rather than a gap. And outside Delphi, this exact pattern repeats: Jackson is Java's answer with the same attribute-and-policy design, .NET's `System.Text.Json` has naming policies that map almost one-to-one onto Neon's `SetMemberCase`, and Python's Pydantic does it with type hints. Everyone converged on the same shape because the problem is the same everywhere.

A rule that outlives every library

Whatever you use, one piece of design advice matters more than the tooling, and it's the practical conclusion of everything above.

Don't serialize your domain objects directly. Make a separate class whose only job is to be the shape of the JSON — a DTO, a Data Transfer Object — and map to it.

The reasoning follows straight from the RTTI diagram. When you serialize your domain class, its field names *become* your public wire contract, enforced by nothing. A refactor that any reasonable developer would consider

safe — renaming a private field, splitting a class, changing a property's visibility — silently changes what your API emits, and you find out from a consumer. A DTO makes the contract explicit and reviewable: it changes when someone means to change it, and your domain model stays free to evolve.

It also solves the awkward cases cleanly. The database handle that can't be serialized simply isn't on the DTO. The password hash that must never leave the process isn't on the DTO — which is a far stronger guarantee than remembering to write `[JsonIgnore]` on it, because a `[JsonIgnore]` someone forgets to add is a data breach, while a field that doesn't exist on the DTO cannot leak.

The cost is real: it's more classes and some mapping code, and for a small internal tool that ceremony genuinely isn't worth it. But the moment JSON crosses a boundary you don't control on both sides, the separation stops being ceremony and starts being the thing that lets you change your code at all.

Takeaways

Serialization is easy to invoke and easy to get subtly wrong, and the difference is almost always in understanding what the round trip can't carry.

- **Objects are not data.** Serialization keeps values and structure, and drops identity, live handles, and — unless you add a discriminator — runtime type. Design classes you intend to serialize as trees of plain data.
- **Deserialization is not the inverse.** It's a partial function over arbitrary input: fields missing, fields unknown, types wrong. Treat incoming JSON as untrusted and check what you get.
- **RTTI makes your identifiers into your wire format.** That's the hidden coupling behind every convenient one-liner. Name members explicitly with `JsonName` (or a Neon naming policy) so refactoring can't move your API.
- **The RTL has two serializers with different defaults.** `TJsonSerializer` writes `FName`; `TJson` writes `fName`; both serialize *fields* unless told otherwise, and `TJson` writes `TBytes` as a number array. Pick one per project.
- **Neon is worth it when the JSON isn't yours to define.** Convention-based naming set once beats per-member attributes at scale. For three classes in an internal tool, the RTL one-liner is the better call.
- **Prefer a DTO to serializing your domain model.** It turns an implicit contract enforced by nothing into an explicit one enforced by a class — and it makes leaking a secret field structurally impossible rather than merely unlikely.

> A serializer answers "what did this object look like?" It cannot answer "what was it?" Everything that goes wrong lives in the gap between those two questions.

Next time we go after the other data type JSON doesn't have — and the one that confuses Delphi developers most. Dates at least *look* like something JSON can carry. **Binary data** doesn't look like anything at all, and the answer everyone reaches for is a word they've typed a hundred times without ever being told what it does: base64.

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 —