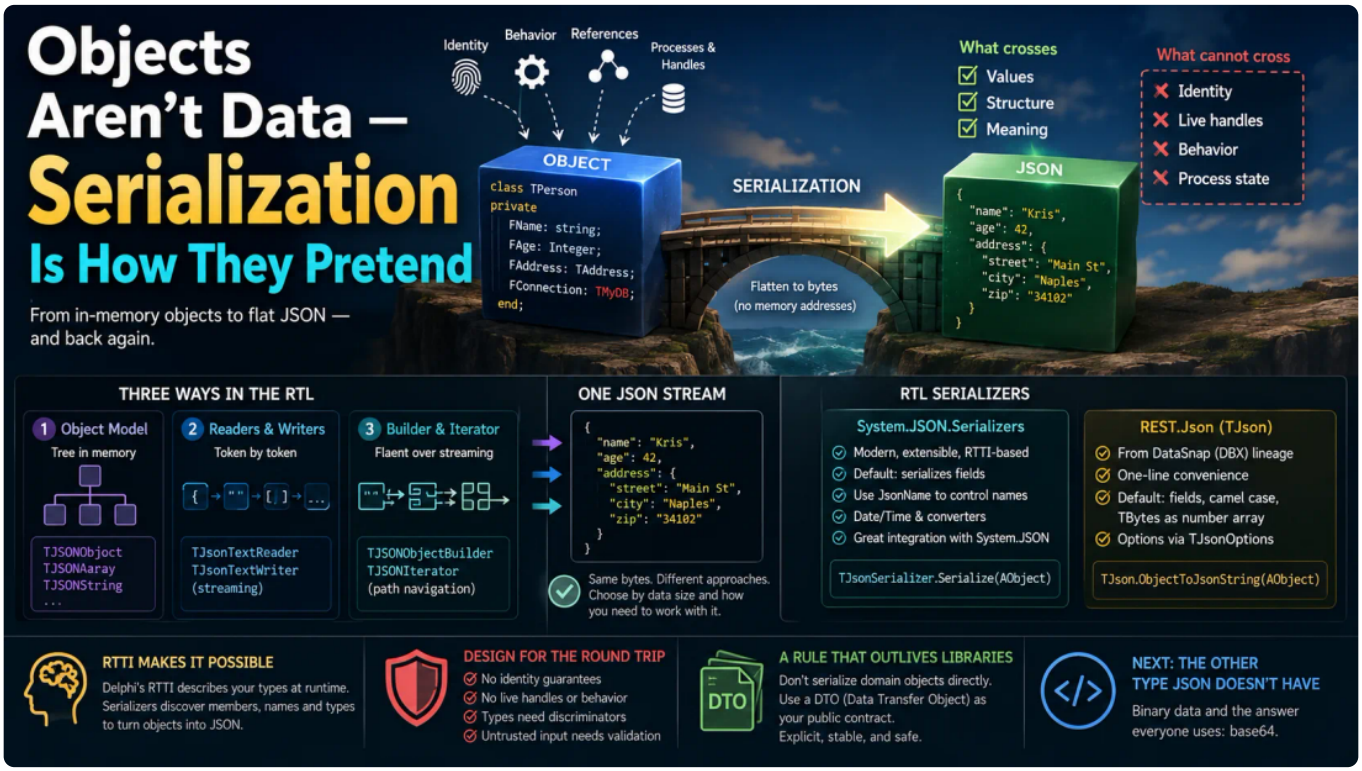
NEWS · JULY 22, 2026

Objekte sind keine Daten — Serialisierung wandelt Objekte in Daten

Teil vier der JSON-Serie für Delphi-Entwickler — was Serialisierung und Deserialisierung wirklich bedeuten, warum der Rundweg mehr verliert als erwartet, wie es mit den RTTI-Serializern der RTL geht und wo Neon übernimmt.

By Dr. Holger Flick



In jedem Beitrag dieser Serie haben Sie JSON bisher von Hand gebaut. In Teil drei sind wir alle drei Wege durchgegangen, die die RTL dafür anbietet — das Objektmodell, das Reader/Writer-Paar, den fluenten Builder —, und bei jedem einzelnen haben Sie jedes Feld einzeln im Code benannt.

Das funktioniert, und bei kleinen Payloads ist es ehrlich und klar. Aber Sie haben eine `TPerson`-Klasse mit zwölf Feldern und schreiben zwölf `AddPair`-Aufrufe, die im Grunde die Klassendeklaration in anderer Syntax wiederholen. Dann fügt jemand ein dreizehntes Feld hinzu, vergisst den Serializer, und ein Bug geht in Produktion.

Der naheliegende Wunsch lautet: *Mach einfach JSON aus meinem Objekt.* Dieser Wunsch hat einen Namen — **Serialisierung** —, und Delphi kann das. Aber bevor wir eine einzige Zeile davon schreiben, möchte ich echte Zeit darauf verwenden, was das Wort eigentlich bedeutet. Denn in diesem Wunsch steckt eine unausgesprochene Annahme, die nicht stimmt — und fast jeder Serialisierungs-Bug, den ich gesehen habe, lässt sich auf sie zurückführen.

Die Annahme ist, ein Objekt *sei* Daten. Ist es nicht. Ein Objekt ist Daten plus Identität plus Verhalten plus Referenzen auf andere Objekte plus, sehr oft, ein Handle auf etwas, das nur existiert, solange Ihr Prozess läuft. Serialisierung ist der Akt, zu entscheiden, welcher Teil davon je wirklich Daten war — und dann einzugestehen, dass der Rest nicht mitkommen kann.

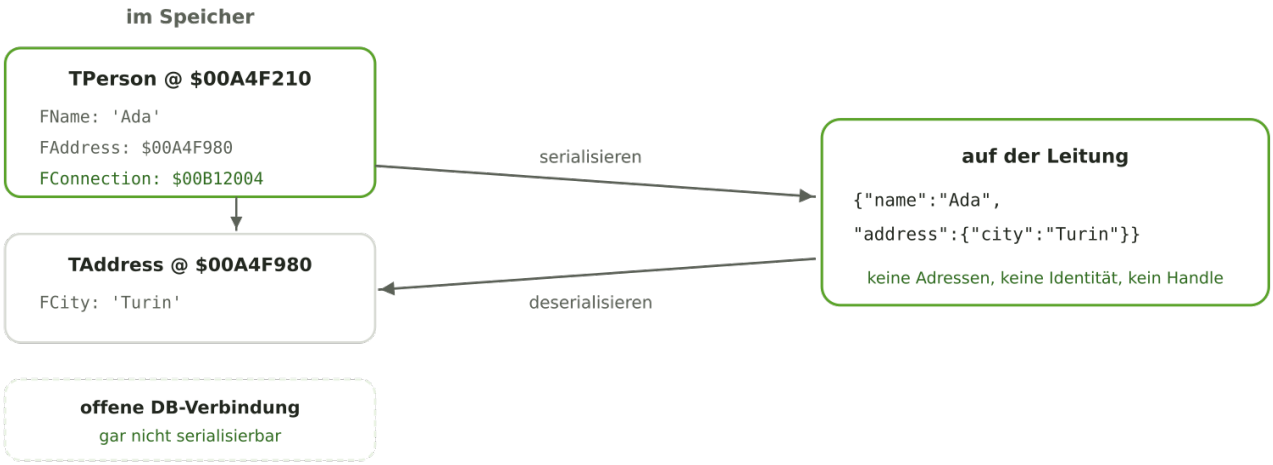
Was Serialisierung tatsächlich ist

Definieren wir es ordentlich, denn die einfache Formulierung ist wirklich nützlich.

Serialisierung ist die Umwandlung einer Struktur, die im Speicher Ihres Programms lebt, in eine flache Folge von Bytes, die gespeichert oder übertragen werden kann. **Deserialisierung** ist die Umkehrung: aus diesen Bytes wieder eine Struktur im Speicher aufbauen.

Das Wort „flach" trägt hier die Hauptlast. Im Speicher ist Ihr Objekt ein Bereich RAM an irgendeiner Adresse, der Werte enthält — und, entscheidend, *Zeiger* auf andere RAM-Bereiche. Dieses Geflecht von Adressen ist nur innerhalb Ihres laufenden Prozesses sinnvoll. Schreiben Sie diese Adressen in eine Datei, werden sie zu Unsinn: Der Speicher, auf den sie verweisen, gehört einem Prozess, der längst beendet ist, auf einer Maschine, die vielleicht nicht einmal die ist, die die Datei liest.

Serialisierung geht also im Kern darum, **die Abhängigkeit vom Speicherlayout Ihres Prozesses zu brechen**. Was auf der anderen Seite herauskommt, muss für sich allein stehen.



Der Speicher hat Adressen und Identität; die Leitung hat beides nicht

Drei Dinge sind beim Überqueren dieser Kluft passiert, und jedes war eine Entscheidung, die jemand treffen musste. `FAddress` war ein Zeiger; daraus wurde ein **verschachteltes Objekt**, weil das, worauf er zeigte, selbst Daten war. `FName` kam unverändert hinüber, weil ein String bereits Daten ist. Und `FConnection` — der gestrichelte Kasten — **konnte schlicht nicht mit**. Ein lebendiges Datenbank-Handle hat außerhalb des Prozesses, der es geöffnet hat, keine Bedeutung. Dem Serializer bleibt nur, es zu überspringen oder zu scheitern, und ein guter lässt Sie sagen, was davon.

Bei diesem letzten Fall lohnt es sich zu verweilen. Es gibt keine geschickte Kodierung, die ihn rettet. Manche Zustände sind schlicht keine Daten, und ein Teil des Entwurfs serialisierbarer Klassen besteht darin, zu wissen, welche Teile das sind.

Warum der Rundweg nicht symmetrisch ist

Man denkt leicht, Deserialisierung sei Serialisierung rückwärts. Ist sie nicht, und die Asymmetrie erzeugt echte Bugs.

Beim Serialisieren haben Sie alles: das Objekt, seinen tatsächlichen Laufzeittyp, jeden Wert. Beim Deserialisieren haben Sie **Text und eine Hoffnung**. Sie müssen entscheiden, welche Klasse Sie instanziieren, bevor Sie die Daten fertig gelesen haben, die Ihnen das sagen würden. Sie müssen mit fehlenden Feldern umgehen, mit Feldern, von denen Sie nie gehört haben, und mit Werten falschen Typs. Serialisierung ist eine totale Funktion über gültige Objekte; Deserialisierung ist eine partielle Funktion über beliebige Eingaben.

Zwei konkrete Verluste sind es wert, benannt zu werden, weil sie überraschen:

Objektidentität verschwindet. Wenn Ihre Struktur zwei Felder hat, die auf *dieselbe* `TAddress`-Instanz zeigen, weiß der Speicher das — ein Objekt, zwei Referenzen. Standard-JSON kann das nicht ausdrücken. Serialisieren Sie, wird die Adresse zweimal ausgeschrieben; deserialisieren Sie, bekommen Sie zwei getrennte Objekte, die zufällig gleich sind. Ändern Sie danach eines, ändert sich das andere nicht — und ein Stück Ihrer Programmlogik funktioniert stillschweigend nicht mehr. Zyklische Referenzen sind dasselbe Problem in schärferer Form: Ein naiver Serializer folgt dem Zyklus, bis der Stack ausgeht. (Delphis `TJsonSerializer` hat für genau diese Problemklasse einen `ReferenceResolver`-Hook, und Neons Doku ist offen darüber, dass zirkuläre Referenzen Sorgfalt verlangen. Beides ist kein Freifahrtschein — der ehrliche Rat lautet: Entwerfen Sie Objektgraphen, die Sie serialisieren wollen, als Bäume.)

Typidentität wird standardmäßig nicht mitgeführt. Ist ein Feld als `TAnimal` deklariert und hält ein `TDog`, hält das JSON die *Werte* fest, nicht den Klassennamen. Deserialisieren Sie in ein `TAnimal`-Feld, bekommen Sie das, was dem Deserializer aufgetragen wurde. Echte Polymorphie zurückzugewinnen erfordert einen Typdiskriminator im Payload — ein `"$type"`-Feld oder Ähnliches —, und das ist eine bewusste Entwurfsentscheidung, nichts, was Sie geschenkt bekommen.

> **Das ist keine JSON-Einschränkung und keine Delphi-Einschränkung**
>
> Jedes Serialisierungsformat steht davor. Javas eingebaute Serialisierung erhält bekanntlich Objektidentität und volle Typinformation — und bezahlte dafür mit einer langen Geschichte von Sicherheitsproblemen, denn das Deserialisieren nicht vertrauenswürdiger Daten, die ihre eigenen Klassen benennen, lässt einen Angreifer wählen, welcher Code läuft. Oracles eigene Secure-Coding-Richtlinien behandeln die Deserialisierung nicht vertrauenswürdiger Eingaben als ernstes Risiko, und die OWASP-Community führt sie als Insecure Deserialization. Dass JSON keine Typen benennen kann, ist aus diesem Blickwinkel ein Vorteil: Ein JSON-Deserializer baut nur Klassen, die *Sie* in Ihrem Code benannt haben. Identität und Polymorphie zu verlieren ist der Preis für ein Format, das sich nicht dazu verleiten lässt, beliebige Objekte zu konstruieren.

Wie Delphi herausfindet, was in Ihrer Klasse steckt

Vor dem Code muss ein Mechanismus erklärt werden, denn alles Folgende hängt davon ab, und er ist der Grund, warum das überhaupt funktioniert.

Damit ein Serializer *irgendein* Objekt in JSON verwandeln kann, muss er zur Laufzeit herausfinden — ohne Kenntnis Ihrer Klasse zur Übersetzungszeit —, welche Felder und Eigenschaften dieses Objekt hat, wie sie heißen und welche Typen sie haben. Die Einrichtung, die das möglich macht, ist RTTI, Run-Time Type Information: Metadaten, die der Compiler in Ihre ausführbare Datei einbettet und die `System.Rtti` abfragbar macht.

RTTI ist die Brücke — der Serializer hat Ihre Klasse nie zur Übersetzungszeit gesehen

Der mittlere Kasten ist der ganze Trick, und er hat eine Konsequenz, die Sie im Bauch spüren sollten. **Ihre Feld- und Eigenschaftsnamen werden Teil Ihres Wire-Formats.** Benennen Sie ein privates Feld aus internem Ordnungssinn um, und wenn der Serializer Felder verwendet, haben Sie soeben das JSON geändert, das Ihre API ausliefert. Der Compiler wird Sie nicht warnen. Deshalb sind die Namens- und Attributoptionen der nächsten Abschnitte nicht kosmetisch — sie sind der Unterschied zwischen einem Wire-Vertrag, den Sie kontrollieren, und einem, der sich ändert, sobald jemand refactored.

Serialisieren mit der RTL

Delphi liefert **zwei** Objekt-Serializer aus verschiedenen Epochen, und Teil drei hat die Trennung schon eingeführt. Beide funktionieren. Ihre Voreinstellungen sind sich nicht einig, und dieser Überraschung begegnet man besser mit Absicht als in Produktion.

Nehmen wir für beide Beispiele diese Klasse:

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

Der moderne: System.JSON.Serializers

`TJsonSerializer` ist der aktuelle, aktiv gepflegte Serializer und Teil der `System.JSON.*`-Familie aus Teil drei:

```
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}     <-- beachten Sie die Feldnamen
  finally
    LSer.Free;
    LPerson.Free;
  end;
end;
```

Schauen Sie sich diese Ausgabe an. `FName`**, nicht** `Name`**.** Die voreingestellte Member-Serialisierung ist `TJson-MemberSerialization.Fields`, Sie bekommen also die privaten Backing-Felder, `F`-Präfix inklusive. Das ist fast nie das JSON, das Sie auf einer öffentlichen API wollen, und es ist die häufigste Überraschung bei dieser Klasse.

Zwei Attribute beheben das. `JsonSerializeAttribute` ändert, *was* serialisiert wird, und `JsonNameAttribute` benennt ein Member auf der Leitung um:

```
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;

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

Diese Form lohnt sich als Gewohnheit. **Jedes serialisierte Member explizit mit** `JsonName` **zu benennen, entkoppelt Ihr Wire-Format von Ihren Pascal-Bezeichnern** — benennen Sie die Eigenschaft danach frei um, und das JSON bewegt sich nicht. Ein paar Zeilen Attributrauschen kaufen Ihnen einen stabilen Vertrag.

`JsonIgnoreAttribute` schließt ein Member ganz aus — die Antwort auf das Datenbank-Handle-Problem aus dem Eingangsdiagramm:

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

Die Deserialisierung ist der Spiegelaufruf, und beachten Sie, dass Sie ein fertig konstruiertes Objekt zurückbekommen:

```
LPerson := LSer.Deserialize<TPerson>(LJson);   // der Aufrufer besitzt das Ergebnis
```

Der Serializer legt auch die Datumsbehandlung offen, um die es in Teil zwei ging — `DateFormatHandling` (`Iso`, `Unix` oder `FormatSettings`) und `DateTimeZoneHandling` (`Local` oder `Utc`). Diese einmal am Serializer zu setzen ist genau die Disziplin „UTC in der Konfiguration entscheiden, nicht an jeder Aufrufstelle", für die Teil zwei argumentiert hat.

Der ältere: REST.Json

`TJson` aus `REST.Json` ist der Serializer der DataSnap-Linie, und seine Kürze ist unbestreitbar reizvoll:

```
uses
  REST.Json;

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

Je eine Zeile. Keine Serializer-Instanz zum Erzeugen und Freigeben. Für ein schnelles internes Werkzeug ist dagegen schwer zu argumentieren, und es erklärt, warum die Unit überall ist.

Ihr Verhalten steuert ein `TJsonOptions`-Set, und die Voreinstellungen sind im RTL-Quelltext so deklariert:

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

Diese Konstante lohnt genaues Lesen, denn drei der fünf Voreinstellungen haben echte Konsequenzen. `joSerialFields` bedeutet, dass auch dieser Serializer standardmäßig **Felder** serialisiert — dieselbe Überraschung wie oben, andere Unit. `joIndentCaseCamel` bedeutet, dass er die ausgegebenen Namen *in camelCase* setzt, aus `FName` wird also `fName` statt `FName`; die beiden Serializer erzeugen damit ab Werk **unterschiedliches JSON für dieselbe Klasse**. Und `joBytesFormatArray` bedeutet, dass `TBytes`-Felder als JSON-**Array von Zahlen** geschrieben werden — `[137,80,78,71,…]` — statt als Base64. Für ein Ein-Megabyte-Bild sind das mehrere Megabyte Ziffern und Kommas. `joBytesFormatBase64` schaltet das um, und die ganze Frage binärer Daten in JSON ist genau da, wo diese Serie als Nächstes hingeht.

> **Zwei Serializer, zwei Sätze Voreinstellungen, eine Klasse**
> Das ist die praktische Gefahr. Serialisieren Sie dieselbe `TPerson` mit `TJsonSerializer` und mit `TJson`, bekommen Sie `{"FName":…}` von dem einen und `{"fName":…}` vom anderen. Keiner ist falsch; sie wurden Jahre auseinander mit unterschiedlichen Konventionen entworfen. Nutzt eine Codebasis beide — und gemischte Codebasen sind aus den historischen Gründen aus Teil drei verbreitet —, dann bestimmt allein die Unit, in der ein Aufruf steht, stillschweigend das Wire-Format. **Wählen Sie einen Serializer pro Projekt und seien Sie bei Namen explizit.** Diese eine Entscheidung räumt eine ganze Kategorie von Verwirrung ab.

Wo eine spezialisierte Bibliothek ihren Platz verdient

Die RTL-Serializer sind leistungsfähig und kostenlos, und für viel interne Arbeit reichen sie wirklich. Aber es gibt eine Lücke, und sie zeigt sich in dem Moment, in dem Ihr JSON zu einem Vertrag passen muss, den Sie nicht geschrieben haben.

Öffentliche APIs haben Namenskonventionen — `snake_case` ist extrem verbreitet —, und die RTL gibt Ihnen ein `JsonName`-Attribut pro Member und sonst wenig. Passen Sie zu einer API mit fünfzig `snake_case`-Feldern, und Sie schreiben fünfzig Attribute von Hand, jedes eine Gelegenheit für einen Tippfehler, der ein stillschweigend fehlendes Feld erzeugt.

Delphi Neon von Paolo Rossi (Apache-2.0) ist um genau dieses Problem herum gebaut. Es ist ein RTTI-basierter Serializer wie der der RTL, aber die Konfiguration ist der Punkt: Namenskonventionen, Sichtbarkeit und Typbehandlung werden **einmal als Richtlinie** gesetzt, statt pro Member annotiert.

```
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;
```

Die Zeile `SetMemberCase(TNeonCase.SnakeCase)` ist das ganze Argument. **Eine Anweisung bringt jedes Member in jeder Klasse auf die Konvention**, und sie bleibt passend, während die Klassen wachsen. Neon unterstützt `SnakeCase`, `camelCase`, `PascalCase`, `UPPERCASE` und `lowercase`, dazu `SetMembers` zur Wahl zwischen Feldern und Eigenschaften und `SetVisibility` zur Steuerung, welche Sichtbarkeitsstufen teilnehmen — die Dinge, die die RTL Sie pro Klasse entscheiden lässt, oder gar nicht.

Die Deserialisierung nimmt eine bestehende Instanz entgegen, was sauber zu Delphis Besitzmodell passt:

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

Zwei weitere Dinge sichern ihm einen Platz in realen Projekten. Seine **Typabdeckung ist breiter** — Records, dynamische Arrays, generische `TObjectList<T>`, Dictionaries mit String-Schlüsseln und streambare Klassen, wo die Unterstützung der RTL für generische Collections dünner ist. Und es bewahrt die Unterscheidung zwischen `TDate` und `TDateTime`, die die RTL laut Teil zwei einebnet, mit einer `UseUTCDate`-Option, die die Zeitzonenfrage einmal in der Konfiguration klärt.

> **Wo ich was nehmen würde — und wo Neon die falsche Wahl ist**

> Aus meiner Sicht: Wenn Sie eine echte REST-API bauen oder konsumieren, deren Namenskonvention Sie nicht kontrollieren, zahlt sich Neon fast sofort aus, und in dieser Lage würde ich standardmäßig danach greifen. Aber ich möchte das sauber eingrenzen, denn „nimm die Bibliothek" ist nicht automatisch die richtige Antwort. Neon ist eine Abhängigkeit: ein weiteres Repository, das man verfolgt, eine weitere Sache, die man bei einer neuen Delphi-Version aktualisiert, und eine weitere Sache, die ein künftiger Betreuer lernen muss. Wenn Sie drei kleine Klassen für ein internes Werkzeug serialisieren, ist `TJson.ObjectToJsonString` eine Zeile und fügt Ihrem Build nichts hinzu — es zu verwenden ist dort die *bessere* technische Entscheidung, nicht die faule. Die RTL-Serializer sind auch die richtige Wahl, wenn
>
> Sie eine Bibliothek ausliefern, die andere konsumieren, denn jede Abhängigkeit, die Sie eingehen, erben sie. Und man sollte klar sagen: Die `TJsonSerializer` der RTL ist ein kompetentes, erweiterbares Stück Arbeit mit einer ordentlichen Converter-Architektur — kein Spielzeug, das Neon ersetzt, sondern ein tiefer angesiedeltes Werkzeug mit anderem Schwerpunkt. Die Frage ist nicht, was besser ist, sondern ob Ihr JSON zu den Konventionen anderer passen muss.

> **Die weitere Landschaft**
> Neon ist nicht die einzige Wahl, und die Alternativen sind es wert, gekannt zu werden. mORMot 2 enthält
>
> Serialisierung als Teil eines sehr schnellen, sehr großen Frameworks — die richtige Wahl, wenn Sie es ohnehin einsetzen, eine Menge zu übernehmen, wenn nicht. JsonDataObjects hört bewusst beim DOM
>
> auf und macht kein Objekt-Mapping, was eine legitime Entwurfsentscheidung ist und keine Lücke. Und außerhalb von Delphi wiederholt sich genau dieses Muster: Jackson ist Javas Antwort mit demselben
>
> Attribut-und-Richtlinien-Entwurf, .NETs `System.Text.Json` hat Naming Policies, die fast eins zu eins auf Neons `SetMemberCase` abbilden, und Pythons Pydantic macht es mit Type Hints. Alle sind bei derselben
>
> Form gelandet, weil das Problem überall dasselbe ist.

Eine Regel, die jede Bibliothek überlebt

Was auch immer Sie verwenden — ein Stück Entwurfsrat wiegt schwerer als das Werkzeug, und es ist die praktische Schlussfolgerung aus allem oben.

Serialisieren Sie Ihre Domänenobjekte nicht direkt. Bauen Sie eine eigene Klasse, deren einzige Aufgabe

es ist, die Form des JSON zu sein — ein DTO, ein Data Transfer Object — und mappen Sie darauf.

Die Begründung folgt direkt aus dem RTTI-Diagramm. Wenn Sie Ihre Domänenklasse serialisieren, *werden* ihre

Feldnamen zu Ihrem öffentlichen Wire-Vertrag, durchgesetzt von nichts. Ein Refactoring, das jeder vernünftige Entwickler für sicher halten würde — ein privates Feld umbenennen, eine Klasse aufteilen, die Sichtbarkeit einer Eigenschaft ändern —, ändert stillschweigend, was Ihre API ausliefert, und Sie erfahren es von einem Konsumenten. Ein DTO macht den Vertrag explizit und überprüfbar: Er ändert sich, wenn jemand ihn ändern will, und Ihr Domänenmodell bleibt frei, sich zu entwickeln.

Es löst auch die unangenehmen Fälle sauber. Das Datenbank-Handle, das nicht serialisierbar ist, steht schlicht nicht im DTO. Der Passwort-Hash, der den Prozess nie verlassen darf, steht nicht im DTO — was eine weit stärkere Zusage ist, als daran zu denken, `[JsonIgnore]` daran zu schreiben. Denn ein vergessenes `[JsonIgnore]` ist ein Datenleck, während ein Feld, das im DTO nicht existiert, nicht lecken *kann*.

Der Preis ist real: mehr Klassen und etwas Mapping-Code, und für ein kleines internes Werkzeug ist diese Zeremonie ihr Geld wirklich nicht wert. Aber sobald JSON eine Grenze überquert, die Sie nicht auf beiden Seiten kontrollieren, hört die Trennung auf, Zeremonie zu sein, und wird zu dem, was Ihnen erlaubt, Ihren Code überhaupt noch zu ändern.

Fazit

Serialisierung ist leicht aufzurufen und leicht subtil falsch zu machen, und der Unterschied liegt fast immer im Verständnis dessen, was der Rundweg nicht tragen kann.

- **Objekte sind keine Daten.** Serialisierung bewahrt Werte und Struktur und lässt Identität, lebende Handles und — ohne Diskriminator — den Laufzeittyp fallen. Entwerfen Sie Klassen, die Sie serialisieren wollen, als Bäume schlichter Daten.
- **Deserialisierung ist nicht die Umkehrung.** Sie ist eine partielle Funktion über beliebige Eingaben: fehlende Felder, unbekannte Felder, falsche Typen. Behandeln Sie eingehendes JSON als nicht vertrauenswürdig und prüfen Sie, was Sie bekommen.
- **RTTI macht Ihre Bezeichner zu Ihrem Wire-Format.** Das ist die verborgene Kopplung hinter jedem bequemen Einzeiler. Benennen Sie Member explizit mit `JsonName` (oder einer Neon-Namensrichtlinie), damit Refactoring Ihre API nicht verschieben kann.
- **Die RTL hat zwei Serializer mit unterschiedlichen Voreinstellungen.** `TJsonSerializer` schreibt `FName`; `TJson` schreibt `fName`; beide serialisieren *Felder*, solange man nichts anderes sagt, und `TJson` schreibt `TBytes` als Zahlen-Array. Wählen Sie einen pro Projekt.
- **Neon lohnt sich, wenn das JSON nicht Ihnen gehört.** Einmal gesetzte konventionsbasierte Benennung schlägt Attribute pro Member in der Menge. Für drei Klassen in einem internen Werkzeug ist der RTL-Einzeiler die bessere Wahl.
- **Bevorzugen Sie ein DTO gegenüber der Serialisierung Ihres Domänenmodells.** Es verwandelt einen impliziten, von nichts durchgesetzten Vertrag in einen expliziten, von einer Klasse durchgesetzten — und macht das Leaken eines geheimen Felds strukturell unmöglich statt bloß unwahrscheinlich.

> Ein Serializer beantwortet „wie sah dieses Objekt aus?". Er kann nicht beantworten „was war es?". Alles, was schiefgeht, wohnt in der Lücke zwischen diesen beiden Fragen.

Nächstes Mal nehmen wir uns den anderen Datentyp vor, den JSON nicht hat — und den, der Delphi-Entwickler am meisten verwirrt. Daten *sehen* wenigstens nach etwas aus, das JSON tragen könnte. **Binärdaten** sehen nach gar nichts aus, und die Antwort, zu der alle greifen, ist ein Wort, das man hundertmal getippt hat, ohne je gesagt zu bekommen, was es tut: 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 —