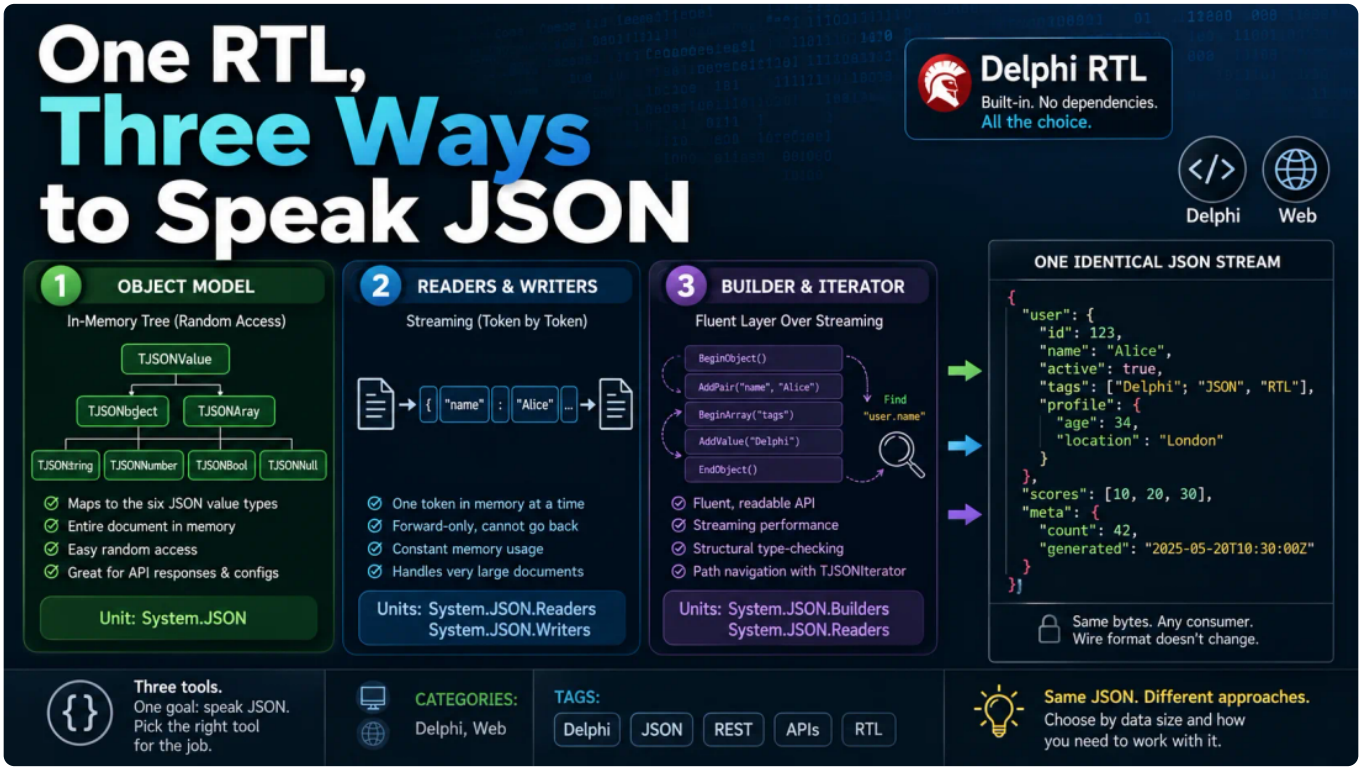
NEWS · JULY 21, 2026

Eine RTL, drei Wege, JSON zu sprechen

Teil drei der JSON-Serie für Delphi-Entwickler — die drei Wege, auf denen System.JSON JSON liest und schreibt, wann sich welcher lohnt, und warum REST.Json immer wieder in Code auftaucht, der nie danach gefragt hat.

By Dr. Holger Flick



In Teil eins haben wir festgehalten, was JSON eigentlich ist — sechs Wertetypen und eine rekursive Regel. In Teil zwei ging es um den Typ, den JSON nicht hat, und darum, wie man Datumsangaben über die Leitung bekommt, ohne jemandem den Geburtstag um einen Tag zu verschieben. Beide Beiträge waren bewusst arm an Pascal.

Damit ist jetzt Schluss. In diesem Beitrag geht es um den Code, und speziell um eine Tatsache, die viele erfahrene Delphi-Entwickler überrascht: Die RTL gibt Ihnen nicht *eine* JSON-API. Sie gibt Ihnen **drei**, sie leben in verschiedenen Units, sie haben wirklich unterschiedliche Stärken — und die meisten lernen nur die erste kennen.

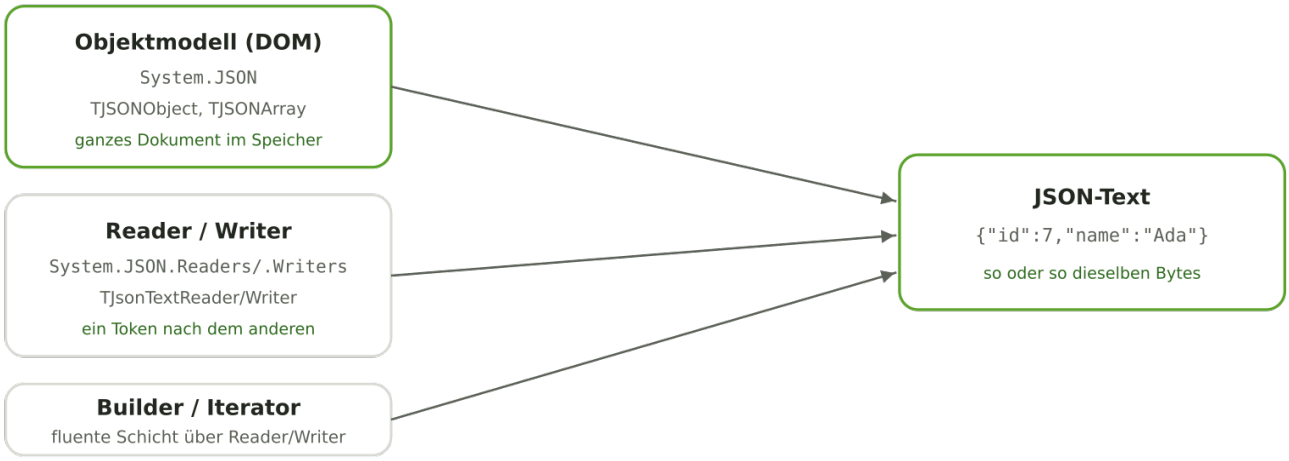
Das ist normalerweise völlig in Ordnung — bis zu dem Tag, an dem es das nicht mehr ist. Wenn Ihnen jemand einen 300-MB-Export überreicht und Ihr Parse-Aufruf den Prozess mit in den Abgrund reißt, kostet die Entscheidung, von der Sie nicht wussten, dass Sie sie getroffen haben, plötzlich einen Nachmittag.

Am Ende dieses Beitrags kennen Sie alle drei, wissen, wann Sie welche greifen — und Sie verstehen den Punkt, der in realen Codebasen die meiste Verwirrung stiftet: warum eine zweite, völlig getrennte JSON-Welt namens `REST.Json` in Ihrer `uses`-Klausel sitzt, woher sie kommt und wie Sie beide auf einen Blick auseinanderhalten.

Die drei Frameworks in einem Bild

Alles in diesem Beitrag lebt in Units, die mit Delphi ausgeliefert werden. Keine Downloads, kein Paketmanager, nichts zu installieren. Die Namen lohnen sich vorab, denn danach ist die Dokumentation indiziert.

Embarcaderos Dokumentation gruppiert sie in zwei benannte Frameworks — das JSON Objects Framework und das Readers and Writers JSON Framework —, aber die Reader-und-Writer-Seite hat in Wahrheit zwei verschiedene Gesichter: den rohen Reader/Writer auf Token-Ebene und das deutlich freundlichere Builder/Iterator-Paar, das darauf aufsetzt. Im Alltag fühlen sich das wie drei verschiedene Werkzeuge an, und so behandle ich sie hier.



Drei APIs über denselben sechs Wertetypen — verschiedene Formen, dasselbe JSON

Lesen Sie das Diagramm von links nach rechts: drei verschiedene Programmiermodelle links, ein identischer Strom von JSON-Text rechts. Nichts an der Ausgabe verrät einem Konsumenten, welche API sie erzeugt hat — die Wahl betrifft ausschließlich, wie *Ihr* Code arbeiten möchte und was er sich an Speicher leisten kann. Diese Einordnung ist wichtig, denn sie bedeutet: Ein späterer Wechsel des Frameworks ist ein lokales Refactoring, keine Änderung am Wire-Format.

Framework 1: das Objektmodell

Das ist das, was fast alle zuerst lernen, und das aus gutem Grund — es bildet das mentale Modell aus Teil eins direkt ab.

Die Unit `System.JSON` gibt Ihnen eine Klasse pro Wertetyp, und sie spiegeln die sechs Typen exakt: `TJSONObject`, `TJSONArray`, `TJSONString`, `TJSONNumber`, `TJSONBool` und `TJSONNull`. Alle stammen von `TJSONValue` ab, und genau das erlaubt es einem Container, jeden von ihnen aufzunehmen — die rekursive Regel aus Teil eins, ausgedrückt als Klassenhierarchie.

Der Preis steht im Namen der Alternative: Das Objektmodell baut **das gesamte Dokument als Objekte im Speicher** auf, auf einen Schlag.

JSON bauen

`AddPair` hat Überladungen für die gängigen Delphi-Typen — `string`, `Integer`, `Int64`, `Double`, `Boolean` und mehr —, einfache Werte brauchen also gar keine Wrapper-Klasse:

```
uses
  System.JSON;

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

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

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

Der Kommentar zur Zeile `AddPair('tags', …)` ist das, was Sie verinnerlichen sollten. **Einen Wert zu einem Container hinzuzufügen, überträgt dessen Besitz.** `LRoot` freizugeben gibt das Array und jeden String darin mit frei — das einzelne `try…finally` um die Wurzel ist also korrekt und vollständig. `LTags` zusätzlich selbst freizugeben wäre ein Double-Free.

> **ToJSON versus Format**
> `ToJSON` erzeugt kompakte Ausgabe — keine Leerzeichen, keine Zeilenumbrüche —, und das ist, was Sie auf der Leitung wollen. Wenn Sie etwas brauchen, das ein Mensch in einem Log oder einem fehlgeschlagenen Test lesen kann, liefert `Format` eingerückte Ausgabe: `LRoot.Format(2)` rückt pro Ebene um zwei Leerzeichen ein. Beide sind auf `TJSONAncestor` deklariert, funktionieren also auf jedem Knoten, nicht nur auf der Wurzel.

JSON parsen

Das Parsen ist eine Klassenfunktion auf `TJSONObject`, und ihr Standardverhalten hat eine scharfe Kante, die man besser kennt, bevor man ihr in der Produktion begegnet:

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

Beachten Sie `RaiseExc: Boolean = False`. **Standardmäßig löst fehlerhaftes JSON keine Exception aus — es liefert** `nil` **zurück.** Das ist eine vertretbare Entscheidung (das Parsen von Netzwerkeingaben sollte Ihren Call-Stack nicht ungefragt sprengen), aber die Prüfung liegt damit bei Ihnen, und sie zu überspringen verwandelt einen klaren „ungültiges JSON"-Fehler in eine Zugriffsverletzung irgendwo weiter unten.

```
var
  LValue: TJSONValue;
  LRoot: TJSONObject;
  LName: string;
  LId: Integer;
begin
  LValue := TJSONObject.ParseJSONValue(LResponseBody);
  if LValue = nil then
    raise Exception.Create('Antwort war kein gültiges JSON');
  try
    if not (LValue is TJSONObject) then
      raise Exception.Create('JSON-Objekt an der Wurzel erwartet');
    LRoot := TJSONObject(LValue);

    LName := LRoot.GetValue<string>('name', '(unbekannt)');
    LId   := LRoot.GetValue<Integer>('id', 0);
  finally
    LValue.Free;    // der Parser übergibt Ihnen den Besitz am Baum
  end;
end;
```

Zwei Details tragen hier echtes Gewicht. Die **Wurzel eines JSON-Dokuments muss kein Objekt sein** — `[1,2,3]` und sogar `42` sind gültige JSON-Dokumente. Deshalb liefert `ParseJSONValue` das Basis-`TJSONValue`, und die Prüfung `is TJSONObject` ist echter defensiver Code, keine Zeremonie. Und `ParseJSONValue` **überträgt den Besitz an Sie**: Der zurückgegebene Baum gehört Ihnen und ist genau einmal an der Wurzel freizugeben.

Die generische `GetValue<T>`-Überladung mit Standardwert ist das Angenehmste in der ganzen Unit — und wird zu selten genutzt:

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

Sie behandelt den Fall des fehlenden Schlüssels und die Typkonvertierung in einem Aufruf, was den üblichen Vierzeiler aus „finden, auf nil prüfen, Typ prüfen, lesen" zu etwas zusammenfaltet, das man tatsächlich lesen kann.

Pfade für verschachtelte Dokumente

Echte API-Antworten verschachteln, und sie Ebene für Ebene abzulaufen wird schnell geschwätzig. `FindValue` akzeptiert einen Pfadausdruck, sodass Sie in einem Schritt in eine Struktur greifen:

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

Punkte steigen in Objekte hinab, `[n]` indiziert Arrays. `FindValue` liefert `nil`, wenn irgendein Teil des Pfades fehlt, statt eine Exception auszulösen — das passt gut zu optionalen Feldern in einer Antwort, die Sie nicht vollständig kontrollieren.

> **Doppelte Schlüssel sind gültiges JSON — und genau hier unterscheiden sich Implementierungen**
> RFC 8259 sagt, Objektnamen „SHOULD be unique" — eine Empfehlung, keine Pflicht — und hält weiter fest, dass Implementierungen unterschiedlich mit Duplikaten umgehen. Der RFC ist explizit, dass das Verhalten für Software, die sich nicht auf eine Regel einigt, „unpredictable" ist. `{"a":1,"a":2}` ist also gültiges JSON, und was ein gegebener Parser daraus macht, ist eine lokale Entscheidung, keine portable Zusage. Erzeugen Sie niemals doppelte Schlüssel, und wenn Sie einen Feed konsumieren, der es tut, testen Sie das tatsächliche Verhalten Ihres Parsers, statt es anzunehmen.

Framework 2: Reader und Writer

Das zweite Framework existiert wegen des Satzes, den ich oben markiert habe: Das Objektmodell hält alles im Speicher. Embarcaderos eigene Dokumentation zum Readers-and-Writers-Framework macht die Ansage direkt — es erlaubt, JSON „directly to a stream, without creating a temporary object" zu lesen und zu schreiben, was sie mit „better performance and improved memory consumption" begründet.

Die Klassen leben in `System.JSON.Readers` und `System.JSON.Writers`. `TJsonReader` und `TJsonWriter` sind die abstrakten Basisklassen; die konkreten Text-Implementierungen sind `TJsonTextReader` und `TJsonTextWriter`, die gegen einen `TTextReader`/`TTextWriter` arbeiten.

Das Modell ist **Token für Token**. Statt eines Baums bekommen Sie einen Cursor, der sich durch eine Folge von Ereignissen bewegt — Objektbeginn, Eigenschaftsname, String-Wert, Objektende —, und Sie reagieren auf jedes einzelne.



Der Reader hält nie das Dokument — nur das aktuelle Token

Die entscheidende Eigenschaft steht in der Bildunterschrift: Es ist immer nur ein Token lebendig. Der Speicherverbrauch ist proportional zur *tiefsten Verschachtelungsebene*, nicht zur Dokumentgröße — und deshalb kann dieses Framework eine mehrere Gigabyte große Datei durch ein Programm streamen, das dafür nie mehr als ein paar Kilobyte belegt. Der Preis ist im Diagramm genauso sichtbar: Der Pfeil zeigt in eine Richtung. Sie können nicht zurück, und Sie können nicht fragen, „was steht im Feld `tags`", bevor Sie dort angekommen sind.

Das Schreiben mit `TJsonTextWriter` ist eine Folge expliziter Strukturaufrufe:

```
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;    // schreibt auf die Platte
  end;
end;
```

Jede geschweifte und eckige Klammer ist ein Methodenaufruf, den Sie selbst machen. Das ist mehr Tipparbeit als `AddPair`, und in einer langen Prozedur verschachtelt man sich leicht falsch — aber nichts wurde je im Speicher gehalten, und die Bytes gingen auf die Platte, während sie entstanden.

Framework 3: Builder und Iterator

Das ist das, was die meisten vergessen — und das, welches das zweite Framework angenehm macht. `System.JSON.Builders` liefert eine **fluente Schicht über Reader und Writer**: dasselbe Streaming-Verhalten,

in Code, der sich liest wie das JSON, das er erzeugt.

`TJSONObjectBuilder` umhüllt einen Writer. Jedes `Add` liefert die Collection zurück, an der Sie bauen, sodass sich Aufrufe verketten:

```
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;
```

Die Einrückung des Pascal spiegelt die Struktur des JSON, sodass ein Verschachtelungsfehler auf dem Papier sichtbar wird statt zur Laufzeit. `BeginArray` liefert ein `TElements` (Dinge ohne Schlüssel), `BeginObject` ein `TPairs` (Dinge mit Schlüssel), und `EndArray`/`EndObject` führen Sie wieder nach oben — der Compiler stellt also sicher, dass Sie kein Schlüssel-Wert-Paar in ein Array einfügen. **Sie bekommen Streaming-Performance mit struktureller Typprüfung**, was eine wirklich schöne Kombination ist und der Grund, warum ich fast immer zum Builder statt zum rohen Writer greife.

Auf der Leseseite umhüllt `TJSONIterator` einen `TJsonReader` und ergänzt Navigation, die der rohe Reader nicht hat:

```
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` nimmt dieselbe Punkt-Pfadsyntax wie `FindValue` im Objektmodell — der RTL-Quelltext dokumentiert sie als „a string consisting of pair names and array indexes, separated by dots", mit `'entities.urls[0].indices[1]'` als eigenem Beispiel. Sie bekommen also pfadbasierten Zugriff, *ohne* das Dokument zu materialisieren — genau das richtige Werkzeug, um drei Felder aus einer großen Antwort zu ziehen.

> **Der Iterator ist standardmäßig flach**
>
> Das ist das Detail, über das man stolpert, und es ist dokumentiert, aber leicht zu übersehen. `TJSONIterator` steigt nicht von selbst in verschachtelte Strukturen ab: Erreicht er ein `StartObject`- oder `StartArray`-Token, müssen Sie `Recurse` aufrufen, bevor Sie das nächste `Next` machen — sonst **überspringt der Iterator das gesamte Objekt oder Array**. `Return` führt zurück zum Elternknoten. Es ist Vorwärtsnavigation mit einem expliziten Stack, den Sie selbst steuern — mächtig, aber Sie müssen es ernst meinen. (`Rewind` existiert, funktioniert aber nur, wenn der zugrundeliegende Stream sich tatsächlich zurücksetzen lässt.)

Die Wahl zwischen ihnen

So entscheide ich — und ich sage ehrlich dazu, dass die erste Option den Großteil der realen Arbeit abdeckt.

Nehmen Sie das Objektmodell, wenn das Dokument bequem in den Speicher passt und Sie wahlfreien Zugriff brauchen. API-Antworten, Konfigurationsdateien, Webhook-Payloads — alles im Bereich von Kilobyte bis wenige Megabyte. In beliebiger Reihenfolge an `LRoot.GetValue<string>('user.name')` herumstochern zu können, ist viel wert, und der Speicherpreis ist in dieser Größenordnung belanglos. Das ist der Standard, und ihn bewusst zu wählen ist völlig in Ordnung.

Nehmen Sie den Builder, wenn Sie JSON erzeugen — besonders große oder gestreamte Ausgabe. Er kostet Sie gegenüber dem Objektmodell nichts an Lesbarkeit — er ist wohl sogar *lesbarer* — und baut nie einen Baum auf. Wenn Sie einen JSON-Export eines Datenbestands schreiben, ist das von Anfang an die richtige Wahl.

Nehmen Sie Reader oder Iterator, wenn die Eingabe groß ist oder als Stream ankommt. Ein Export von mehreren hundert Megabyte, eine Logdatei, eine Antwort, die Sie schon während des Downloads verarbeiten. Das Objektmodell bräuchte das Ganze resident, plus Objekt-Overhead obendrauf.

> **Die Größenfrage, ehrlich beantwortet**
> Ich möchte den Speicherpunkt nicht überzeichnen, denn „aus Performancegründen streamen" wird weit öfter nachgeplappert als gemessen. Bei einem typischen REST-Payload ist der Overhead des Objektmodells real, aber völlig belanglos — Sie werden ihn nicht bemerken, und zu einer Streaming-API zu greifen, um ein paar hundert Kilobyte zu sparen, ist ein schlechter Tausch gegen die Lesbarkeit, die Sie aufgeben. Die Streaming-Frameworks verdienen ihr Geld in Größenordnungen, die die meisten Anwendungen nie erreichen. Mein Rat: standardmäßig das Objektmodell, den Builder für erzeugte Ausgabe (weil er ohnehin angenehm ist), und wechseln Sie zum Reader, wenn Sie ein tatsächlich gemessenes Problem haben oder sicher wissen, dass die Eingabe unbegrenzt ist. Verfrühtes Streaming ist ein genauso realer Fehler wie verfrühte Optimierung im Allgemeinen.

Das andere JSON in Ihrer uses-Klausel

Nun zu dem, was in diesem Bereich mehr Verwirrung stiftet als alles andere — und es geht eigentlich gar nicht um die drei Frameworks.

Wenn Sie in einer Delphi-Codebasis mit etwas Alter gearbeitet haben, kennen Sie das:

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

Zwei Units. Beide zu JSON. Beide von Embarcadero. Und Code, der sie auf eine Weise mischt, die willkürlich wirkt, bis man die Geschichte kennt.

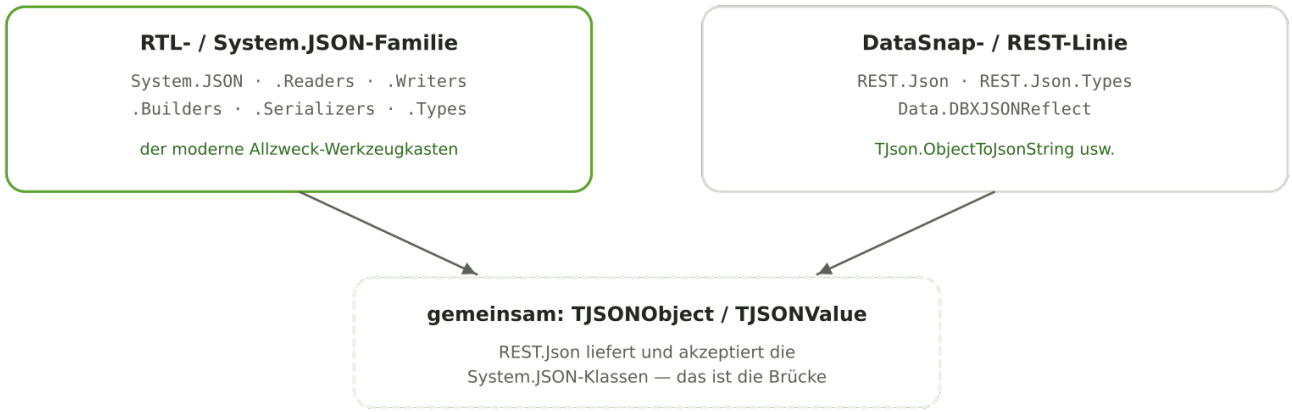
`REST.Json` **stammt aus einer anderen Linie.** Seine Herkunft liegt in DataSnap — Embarcaderos Mehrschichten-Framework —, und man sieht es dem RTL-Quelltext bis heute an: `REST.Json` ist auf `TJSONMarshal`/`TJSONUnMarshal` aus `Data.DBXJSONReflect` aufgebaut, dem Reflection-Marshaller von DBX (DataSnap/dbExpress). Es war die Antwort auf „wie mache ich aus einem Objekt JSON", Jahre bevor `System.JSON.Serializers` existierte.

Was es Ihnen gibt, ist eine sehr bequeme Klasse, `TJson`, mit Klassenmethoden, die die ganze Arbeit in einer Zeile erledigen:

```
uses
  REST.Json;

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

Das ist wirklich nützlich, und deshalb ist die Unit überall. Objektserialisierung ist das Thema des nächsten Beitrags, ich gehe hier also nicht tiefer — der Punkt ist vorerst allein, **zu wissen, zu welcher Welt ein Typ gehört**.



Zwei Linien, ein gemeinsames Wertemodell — hier liegt die Überschneidung

Der gestrichelte Kasten erklärt, warum gemischter Code nicht wirklich kaputt ist. `TJson.ObjectToJsonObject` liefert ein `System.JSON.TJSONObject`; `TJson.JsonToObject<T>` nimmt eines entgegen. Die beiden Welten treffen sich bei den Wertklassen — eine Prozedur, die ein `TJSONObject` entgegennimmt, interessiert sich also nicht dafür, welche Unit es erzeugt hat. Gemischter Code kompiliert und funktioniert; er *wirkt* nur zusammenhanglos auf den, der ihn als Nächstes liest.

Es gibt noch eine Merkwürdigkeit, die viel von der Verwirrung erklärt, und sie steht direkt im RTL-Quelltext. `REST.Json.Types` deklariert:

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

Das ist ein **Alias**. Das Attribut, das Sie als `[JSONName('user_id')]` aus `REST.Json.Types` verwenden, *ist derselbe Typ* wie `JsonNameAttribute` aus `System.JSON.Types`. Die beiden Welten arbeiten also nicht nur auf Wertebene zusammen — manche ihrer Attribute sind buchstäblich identisch, unter zwei verschiedenen Namen. Falls Sie sich je gefragt haben, warum dasselbe Attribut funktioniert, egal welche Unit Sie eingebunden haben: Das ist der Grund.

> **Eine Faustregel für neuen Code**
>
> Meine Präferenz, und ich markiere sie als Präferenz statt als Regel: Wählen Sie für neuen Code die *`System.JSON.-Familie und bleiben Sie darin**. Das ist die aktiv weiterentwickelte Linie, System.JSON.Serializers (Thema des nächsten Beitrags) ist die moderne Serialisierungs-geschichte, und in einem Namensraum zu bleiben macht den Code für die nächste Person lesbar. Allerdings – ich würde entschieden widersprechen, wenn das jemand als „reiß REST.Json aus funktionierendem Code" liest. TJson.ObjectToJsonString ist ein wirklich guter Einzeiler, es ist stabil, und es gibt keinen Preis für ein Refactoring, das kein Verhalten ändert. Die eigentlichen Kosten entstehen nicht durch die Nutzung einer der beiden Units, sondern durch die Nutzung *beider, ohne zu wissen warum*. Am schwächsten ist diese Präferenz in DataSnap-Codebasen, wo REST.Json` das Idiom ist, das der restliche Code ohnehin spricht — dort schlägt Konsistenz mit dem umgebenden Code meine Namensraum-Ordnungsliebe jedes Mal.

Was der Rest der Welt macht

Ein Moment Perspektive lohnt sich, denn diese Dreiteilung ist keine Delphi-Eigenart — sie ist die Standardform einer ausgereiften JSON-Bibliothek, und das Muster zu erkennen macht andere Stacks sofort lesbar.

.NETs `System.Text.Json` hat exakt dieses Trio: `JsonDocument` (DOM), `Utf8JsonReader`/`Utf8JsonWriter` (Streaming) und `JsonSerializer` (Objekte). Javas Jackson benennt sie unumwunden — das Tree Model, die

Streaming-API und Data Binding. Gos `encoding/json` bietet `Unmarshal` in eine Map und `json.Decoder` für Streams. Dieselben drei Antworten, weil es auf „wie möchte ich dieses Dokument anfassen" nur drei sinnvolle Antworten gibt.

Speziell in der Delphi-Welt sind die eingebauten Frameworks nicht Ihre einzige Option, und zwei Open-Source-Alternativen sind es wert, gekannt zu werden. JsonDataObjects von Andreas Hausladen (MIT)

ist ein Ein-Unit-DOM-Parser mit dem Ruf, sehr schnell und sehr angenehm zu sein — die API ist für den Alltag wohl schöner als `System.JSON`, und dass es eine einzige Datei ist, macht die Übernahme trivial. mORMot 2 (MPL/GPL/LGPL-Tri-Lizenz) enthält eine extrem schnelle JSON-Engine als Teil eines viel größeren

Frameworks. Beide werden aktiv gepflegt und breit eingesetzt. Der Vorteil der RTL ist schlicht, dass sie schon da ist, ohne Abhängigkeit, die man jemandem erklären muss — was für viele Teams den Ausschlag gibt und für andere eben nicht.

Fazit

Drei Frameworks klingt nach viel, aber die Aufteilung hat System: Sie unterscheiden sich darin, ob das Dokument im Speicher liegt und ob Sie sich frei darin bewegen können.

- **Das Objektmodell (`System.JSON`) ist der sinnvolle Standard.** `TJSONObject` und Verwandte bilden die sechs Wertetypen direkt ab, und `GetValue<T>` mit Standardwert ist die nützlichste Methode der Unit. Denken Sie daran: Container besitzen, was Sie hinzufügen — geben Sie nur die Wurzel frei.
- `ParseJSONValue` **liefert bei fehlerhafter Eingabe standardmäßig** `nil`**.** `RaiseExc` ist `False`, solange Sie nichts anderes sagen. Prüfen Sie das Ergebnis, und prüfen Sie, ob die Wurzel der erwartete Typ ist — ein gültiges JSON-Dokument darf ein Array oder eine nackte Zahl sein.
- **Der Builder ist der angenehmste Weg, JSON zu* schreiben.* Fluent, streamend und strukturell typgeprüft, ohne den Speicherpreis des Objektmodells. Es gibt wenig Grund, ihn für erzeugte Ausgabe nicht zu nutzen.
- **Reader und Iteratoren sind für Größe da.** Konstanter Speicher, nur vorwärts, pfadfähig über `TJSONIterator.Find` — und denken Sie an `Recurse`, sonst überspringen Sie stillschweigend ganze Zweige. Greifen Sie dazu, wenn Sie ein echtes Größenproblem haben, nicht aus Prinzip.
- `REST.Json` **ist eine eigene Linie, kein Rivale.** Es kam von DataSnap, baut auf dem DBX-Marshaller auf und trifft `System.JSON` bei den gemeinsamen Wertklassen — deshalb funktioniert gemischter Code. Bevorzugen Sie einen Namensraum in neuem Code; führen Sie keinen Krieg gegen den anderen in altem Code.

> Die RTL gibt Ihnen einen Baum, einen Token-Strom und eine fluente Schicht über dem Token-Strom. Wählen Sie nach Datenmenge und danach, wie frei Sie sich bewegen müssen — alles Weitere ist Geschmackssache.

Nächstes Mal nehmen wir uns die Frage vor, um die das alles kreiste: **Serialisierung**. Was es tatsächlich

bedeutet, ein Objekt in Daten zu verwandeln und zurück, warum der Rundweg schwerer ist, als er aussieht, was Delphis RTTI-basierte Serializer ab Werk für Sie tun — und wo eine spezialisierte Bibliothek wie Paolo Rossis Neon übernimmt.

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 —