NEWS · JULY 18, 2026

Wir sind längst von JSON umgeben — hier ist, was es wirklich ist

Der erste Beitrag einer Serie über JSON mit Delphi — was JSON eigentlich ist, warum es XML verdrängt hat, wo Sie ihm täglich begegnen und aus welchen sechs Bausteinen sein gesamtes Datenmodell besteht.

By Dr. Holger Flick



Öffnen Sie jetzt Ihren Editor und denken Sie an das Letzte, das Sie gebaut haben und das mit der Außenwelt gesprochen hat. Ein REST-Aufruf an einen Zahlungsanbieter. Eine Konfigurationsdatei, die Ihre Anwendung beim Start liest. Ein Webhook von GitHub. Die Antwort eines Wetterdienstes, einer Karten-API, eines LLM. Fast jede davon hat auf der Leitung dieselbe Sprache gesprochen — ein kleines Textformat mit geschweiften und eckigen Klammern, das Sie vermutlich schon tausendmal getippt haben, ohne je innezuhalten und zu fragen, was es eigentlich ist.

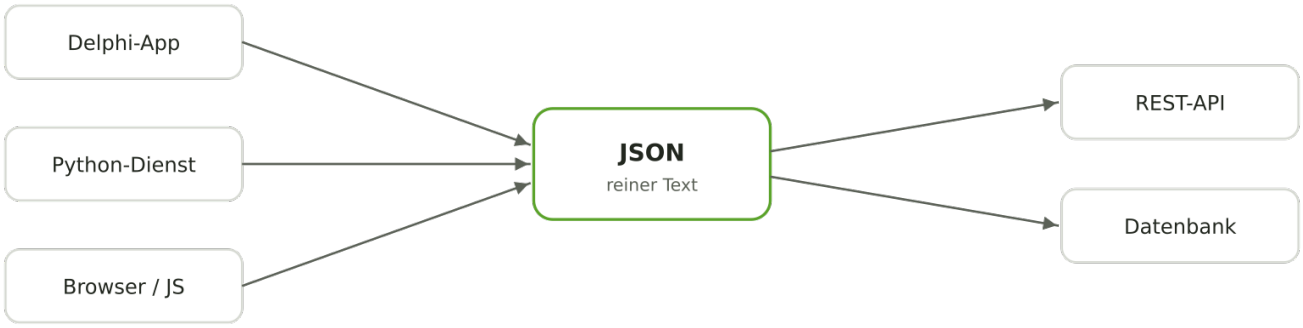
Dieses Format ist JSON, und dieser Beitrag ist der Auftakt einer Serie über die Arbeit damit in Delphi. Bevor wir eine einzige Zeile Pascal schreiben, möchte ich das Unglamouröse tun und das Fundament richtig legen: was JSON *ist*, warum sich die ganze Branche darauf geeinigt hat, warum es immer wieder „das, was XML abgelöst hat" genannt wird — und, der Teil, der wirklich zählt, wenn Sie sich hinsetzen, um JSON zu parsen oder zu erzeugen: die überraschend kleine Menge an Regeln, aus denen sein gesamtes Datenmodell besteht.

Am Ende werden Sie jedes JSON-Dokument, egal wie tief verschachtelt, betrachten und die Handvoll Teile erkennen können, aus denen es zusammengesetzt ist. Dieses gedankliche Modell ist genau das, was jede Bibliothek dieser Serie — angefangen bei den Klassen, die Delphi bereits mitbringt — bearbeiten soll.

Wo Ihnen JSON längst begegnet

Beginnen wir mit der Behauptung im Titel, denn sie ist leicht zu belegen. JSON ist kein Nischenformat, für das man sich bewusst entscheidet — es ist der Standardträger für strukturierte Daten quer durch die moderne Software. Ein kurzer Rundgang durch die Orte, an denen es auftaucht:

- **Web-APIs.** Die überwältigende Mehrheit der öffentlichen REST-APIs liefert JSON zurück. Wenn Ihre Delphi-Anwendung Stripe, GitHub, OpenWeather oder einen LLM-Endpunkt aufruft, ist der zurückkommende Body JSON-Text.
- **Konfigurationsdateien.** Die `package.json` in jedem Node.js-Projekt, `tsconfig.json`, die `settings.json` von VS Code, `appsettings.json` in .NET — die Konfiguration ist weitgehend zu JSON gewandert.
- **Datenbanken.** PostgreSQL hat vollwertige `json`- und `jsonb`-Spaltentypen; MongoDB speichert Dokumente, die für alle praktischen Zwecke JSON sind.
- **Messaging und Logs.** Webhooks, Message Queues und strukturierte Log-Zeilen sind routinemäßig JSON-Objekte, eines pro Ereignis.



Ein Textformat zwischen sehr unterschiedlichen Systemen

Der Sinn des Diagramms sind nicht die Kästchen — es ist der eine Knoten in der Mitte. Systeme, die in unterschiedlichen Sprachen geschrieben sind, auf unterschiedlichen Maschinen laufen und von Teams gebaut wurden, die nie miteinander gesprochen haben, einigen sich alle auf ein einziges reines Textformat, um sich Daten hin und her zu reichen. Diese Einigung ist der ganze Existenzgrund von JSON.

Was JSON eigentlich ist

JSON steht für **JavaScript Object Notation**. Es ist ein Textformat zur Darstellung strukturierter Daten — eine Art, Objekte, Listen, Zahlen und Text als Zeichenkette aufzuschreiben, die jedes Programm erzeugen und jedes Programm wieder einlesen kann.

Zwei Dinge an dieser Definition sind wichtig. Erstens ist es **Text**: ein JSON-Dokument besteht nur aus Zeichen, kodiert (nach dem aktuellen Standard) als UTF-8. Sie können es in Notepad öffnen, per E-Mail verschicken, in einen Chat einfügen. Zweitens ist es **sprachunabhängig**, dem Namen zum Trotz. Das „JavaScript" in JSON ist Geschichte, keine Voraussetzung: die Syntax wurde von der Art übernommen, wie man in JavaScript Objekt- und Array-Literale schreibt, aber das Format selbst gehört zu keiner Sprache. Delphi, Python, Rust und COBOL lesen und schreiben exakt dasselbe JSON.

Die Geschichte ist einen Absatz wert, denn sie erklärt die Gestalt von allem, was folgt. JSON wurde um 2001 von Douglas Crockford eingeführt — er sagt bewusst, er habe es *entdeckt* statt erfunden, denn die Notation steckte bereits in JavaScript, das seine Syntax wiederum aus ECMAScript, 1999 standardisiert, bezog.

Die erste JSON-Nachricht wurde im April 2001 versendet. Crockford beschrieb es auf einer kleinen Website, damit andere es nutzen konnten, und das war beinahe schon die gesamte Spezifikation. Später wurde es formalisiert — als RFC 4627 im Jahr 2006, dann als Standard ECMA-404 im Jahr 2013 und heute als RFC 8259 (Internet-Standard STD 90, 2017), den die beiden Normungsgremien bewusst identisch hielten.

> **Warum der Name ein wenig in die Irre führt**
> Weil JSON „JavaScript" im Namen trägt, nehmen manche an, es sei eine reine JavaScript-Angelegenheit oder gültiges JSON sei gültiges JavaScript und umgekehrt. Beides stimmt nicht. JSON ist ein striktes, in sich geschlossenes Datenformat; JavaScript ist eine Programmiersprache, die zufällig etwas Literal-Syntax mit ihm teilt. In Delphi fassen Sie nie JavaScript an, um mit JSON zu arbeiten — Sie arbeiten mit dem Text und den Klassen, die wir in dieser Serie behandeln.

Warum es so beliebt wurde

Viele Datenformate sind gekommen und gegangen. JSON ist geblieben, und es lohnt sich, genau zu sein, *warum* — denn dieselben Gründe machen es gelegentlich auch zur falschen Wahl.

Die ehrliche Kurzantwort lautet: JSON ist **gerade genug**. Es ist menschenlesbar, sodass Sie eine Antwort mit bloßem Auge überfliegen und ohne Werkzeuge verstehen können. Es bildet fast perfekt die Datenstrukturen ab, in denen Programmierer ohnehin denken — Objekte (Records) und Arrays (Listen) — sodass es sich in nahezu jeder Sprache natürlich anfühlt, ein JSON-Dokument in Programmdaten und zurück zu verwandeln. Es ist kompakt genug für die Leitung, ohne kryptisch zu sein. Und die Grammatik ist klein genug, dass eine kompetente Entwicklerin oder ein kompetenter Entwickler sie vollständig im Kopf behalten kann und ein Parser dafür an einem Nachmittag geschrieben ist.

Diese letzte Eigenschaft wird unterschätzt. Ein Format, das sich trivial parsen lässt, bekommt in *jeder* Sprache schnell einen hochwertigen Parser, und sobald jede Sprache ein Format billig und korrekt lesen kann, erledigt der Netzwerkeffekt den Rest. JSON wurde nicht deshalb zum Standard, weil es das leistungsfähigste Format ist, sondern weil es die meiste Reibung beseitigte.

> **Wo JSON das falsche Werkzeug ist**
> Nichts davon macht JSON universell überlegen, und es lohnt sich, bei den Grenzen ehrlich zu sein. JSON hat keinen eingebauten Datumstyp (das Thema des nächsten Beitrags), keine Kommentare und kein eigenes Schema — für die Validierung der Struktur greifen Sie zu JSON Schema. Als Text ist es größer und langsamer zu parsen als Binärformate wie Protocol Buffers oder MessagePack, weshalb Systeme mit hohem Durchsatz intern oft diese bevorzugen. Und für Dokumente mit viel Auszeichnung und gemischtem Inhalt — man denke an ein Buchkapitel mit Inline-Formatierung — passt das Modell von XML nach wie vor besser. Die Dominanz von JSON ist real, aber es ist eine Dominanz der *Eignung für den häufigen Fall*, keine universelle Überlegenheit.

Die XML-Frage: warum JSON immer wieder „das Nächste nach XML" genannt wird

Wer lange genug in dieser Branche ist, hat JSON schon als das Format gehört, das „XML abgelöst hat". Diese Darstellung ist zur Hälfte richtig, und die falsche Hälfte lohnt es sich zu entwirren.

XML — Extensible Markup Language — war das dominierende Austauschformat der späten 1990er und der 2000er Jahre. Es ist mächtig: Namensräume, Schemata (XSD), Transformation (XSLT), eine vollständige Abfragesprache (XPath/XQuery) und die Fähigkeit, Daten und ausgezeichneten Text in einem Dokument zu mischen. Für viel dieser Mächtigkeit zahlt man allerdings mit Umständlichkeit und Zeremonie. Dieselben Daten, die XML in öffnende und schließende Tags hüllt, drückt JSON mit einem Bruchteil der Zeichen aus.

XML

```
<user>
   <name>Ada</name>
   <active>true</active>
</user>
```

JSON

```
{
   "name": "Ada",
   "active": true
}
```

Derselbe Datensatz, zwei Formate — der Unterschied ist meist Zeremonie

Stellt man die beiden nebeneinander, ist der Reiz sofort greifbar: JSON sagt dasselbe mit weniger. Beachten Sie aber, was das Diagramm *nicht* zeigt — die Namensräume von XML, seine Schemavalidierung, seine Fähigkeit, Attribute und gemischten Inhalt zu tragen. Das ist der ehrliche Teil des Vergleichs. JSON hat nicht gewonnen, indem es mächtiger war als XML; es hat gewonnen, indem es die Teile weggelassen hat, die die meiste alltägliche API- und Konfigurationsarbeit nie genutzt hat, und den Teil behielt, der sauber auf die Art abbildet, wie Programmierer Daten modellieren. Für dokumentlastige Arbeit und in vielen Unternehmens- und Finanzsystemen ist XML nach wie vor das richtige Werkzeug und weiterhin sehr im Einsatz. „Abgelöst" ist übertrieben; „hat den häufigen Fall übernommen" ist genau richtig.

Wie JSON aufgebaut ist: die ganze Grammatik in einem Abschnitt

Hier kommt der Lohn. Alles bisher war Kontext; dies ist der Teil, den Sie täglich nutzen werden. Das gesamte JSON-Datenmodell ist aus einer kleinen Menge von **Werten** gebaut, und ein JSON-Dokument ist nichts weiter als ein einzelner Wert — meist ein Objekt — mit weiteren Werten darin verschachtelt.

Die zwei Container

JSON gibt Ihnen zwei Wege, Dinge zu gruppieren, und sie sind das strukturelle Rückgrat jedes Dokuments.

Ein **Objekt** ist eine ungeordnete Sammlung von **Name/Wert-Paaren**, umschlossen von geschweiften Klammern `{ }`. In den Worten der Spezifikation: *„Ein Objekt ist eine ungeordnete Menge von Name/Wert-Paaren."* Jedes Paar besteht aus einem String-**Namen** (auch Schlüssel oder Eigenschaft genannt), einem Doppelpunkt und einem **Wert**. Denken Sie an einen Record oder ein Wörterbuch (Dictionary).



Ein Objekt: Klammern um Name/Wert-Paare, durch Komma getrennt

Das Diagramm zerlegt ein kleines Objekt in seine Teile: zwei Paare, jedes ein in Anführungszeichen gesetzter Name, ein Doppelpunkt und ein Wert, durch ein Komma getrennt, alles innerhalb der Klammern. Namen sind in JSON **immer** doppelt in Anführungszeichen gesetzte Strings — eine Regel, über die man stolpert, wenn man von JavaScript kommt, wo die Anführungszeichen oft optional sind.

Ein **Array** ist eine geordnete Liste von Werten, umschlossen von eckigen Klammern `[ ]`. Wieder die Spezifikation: *„Ein Array ist eine geordnete Sammlung von Werten."* Anders als die Paare eines Objekts haben die Elemente eines Arrays keine Namen — sie haben Positionen, und die Reihenfolge bleibt erhalten.

Die Werte, die Sie hineinlegen können

Jeder Platz — jeder Wert in einem Objektpaar, jedes Element eines Arrays — ist genau einer dieser Typen. Dies ist die vollständige Liste; mehr gibt es nicht:

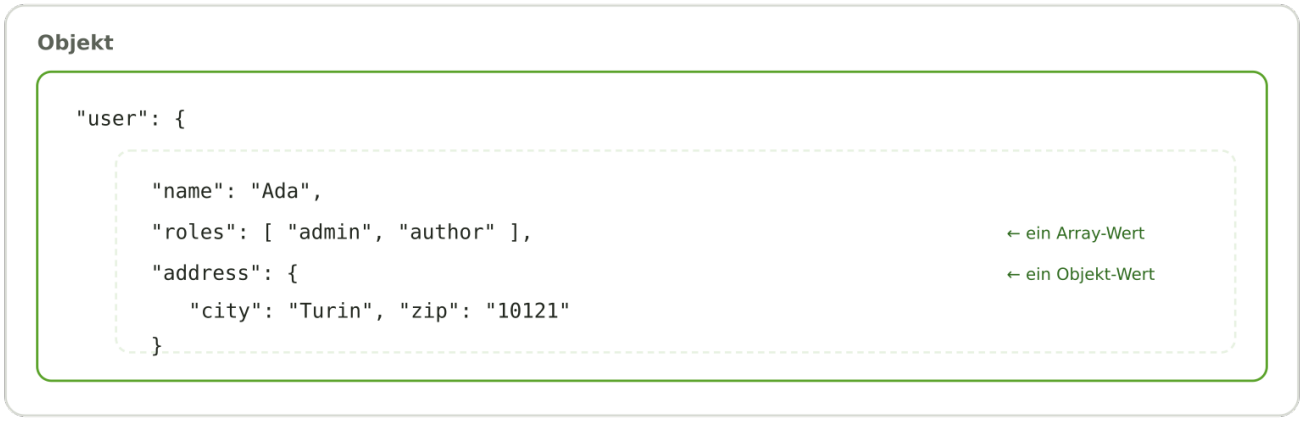
Typ	Was es ist	Beispiel
String	Text, stets in doppelten Anführungszeichen	`"Ada Lovelace"`
Number	Eine Ganz- oder Gleitkommazahl, ohne Anführungszeichen	`42`, `-3.14`, `6.022e23`
Boolean	Das Literal `true` oder `false`	`true`
null	Das ausdrückliche „kein Wert"	`null`
Objekt	Ein verschachteltes `{ }` aus Name/Wert-Paaren	`{ "id": 7 }`
Array	Ein verschachteltes `[ ]` aus Werten	`[1, 2, 3]`

Ein paar ehrliche Details, die in der Praxis zählen. JSON-Strings sind Unicode und verwenden Backslash-Escapes (`\n`, `\"`, `\uXXXX`). JSON-**Zahlen** sind ein einziger Typ — das Format zieht keine Grenze zwischen

Ganzzahl und Gleitkomma, was genau der Grund ist, warum Bibliotheken (auch die von Delphi) entscheiden müssen, wie sie Ihnen eine Zahl zurückgeben, und warum sehr große Ganzzahlen Sorgfalt erfordern. Und entscheidend: es gibt **keinen Datumstyp** — ein Punkt, auf den wir gleich zurückkommen.

Die eine Regel, die es mächtig macht: Rekursion

Hier ist der ganze Trick, und er ist der Grund, warum eine so winzige Grammatik nahezu alles beschreiben kann. Sehen Sie noch einmal auf die Tabelle: zwei der sechs Werttypen — **Objekt** und **Array** — bestehen selbst aus Werten. Ein Wert innerhalb eines Arrays kann also ein weiteres Array sein. Ein Wert innerhalb eines Objekts kann ein weiteres Objekt sein. Das wiederum kann weitere Arrays und Objekte enthalten. Bis ganz nach unten.



Ein Wert kann Werte enthalten, die Werte enthalten — die Verschachtelung hat keine feste Tiefe

Lesen Sie das Diagramm von außen nach innen: das oberste Objekt enthält einen `user`, dessen Wert ein weiteres Objekt ist; *darin* ist `roles` ein Array aus Strings und `address` wiederum ein Objekt. Um das möglich zu machen, wurde nichts Neues eingeführt — es sind dieselben sechs Werttypen, nur so angeordnet, dass die Werte eines Containers selbst Container sind. Diese eine rekursive Regel ist es, die JSON eine flache Konfigurationsdatei und eine tief verschachtelte API-Antwort mit denselben sechs Teilen darstellen lässt. Hat man sie einmal erkannt, kann man jedes JSON-Dokument lesen, egal wie einschüchternd es aussieht, indem man auf jeder Ebene fragt: *ist dieser Wert einer der sechs?* Er ist es immer.

Wie es in der Serie weitergeht

Das ist das Fundament: JSON ist Text, es ist sprachneutral, es hat den häufigen Fall übernommen, der einst XML gehörte, und sein gesamtes Datenmodell besteht aus sechs Werttypen plus einer Regel, die sie sich verschachteln lässt. Damit in der Hand wird der Rest der Serie praktisch und Delphi-spezifisch.

Der nächste Beitrag nimmt sich genau das vor, was die obige Grammatik betont auslässt: **Datum und Uhrzeit**.

Ihnen ist vielleicht aufgefallen, dass es in jener Tabelle keinen Datumstyp gibt — und dieses Fehlen verursacht echte Reibung. JSON hat keine native Möglichkeit zu sagen „das ist ein Zeitstempel", also wird ein Datum per Konvention entweder als speziell formatierter **String** (meist ISO 8601, wie `"2026-07-18T09:30:00Z"`) oder als **Zahl** (eine Anzahl von Sekunden oder Millisekunden seit einem Epochenzeitpunkt) durchgeschmuggelt.

Welches von beiden Sie bekommen — und wie Sie es in Delphi sauber lesen und schreiben — ist ein eigener Beitrag.

Von dort an wird es handfest. Delphi bringt in seiner Laufzeitbibliothek ein vollständiges JSON-Werkzeug mit — die Unit `System.JSON`, mit einer kleinen Familie von Klassen (`TJSONObject`, `TJSONArray`, `TJSONString`, `TJSONNumber`, `TJSONBool`, `TJSONNull`), die genau die sechs Werttypen widerspiegeln, die Sie gerade gelernt haben. Wir werden JSON erzeugen, parsen und durchlaufen — mit nichts als dem, was ab Werk dabei ist —, bevor wir die freien Drittanbieter-Bibliotheken hinzuziehen, die die häufigen Fälle deutlich weniger umständlich machen.

> **JSON ist größer als Delphi — und genau das ist der Punkt**

> Weil JSON sprachneutral ist, hat alles in dieser Serie ein Gegenstück in anderen Stacks: ein Node.js- oder TypeScript-Dienst nutzt das eingebaute `JSON.parse`/`JSON.stringify`, Python hat sein `json`-Modul in der Standardbibliothek, und Go, Rust und C# bringen alle vollwertige JSON-Unterstützung mit. Das ist kein Umweg weg von der Delphi-Geschichte — es ist der Grund, JSON gut zu lernen. Die Bytes, die Ihre Delphi-Anwendung sendet, sind dieselben Bytes, die ein Python-Dienst liest, sodass das Modell, das Sie hier aufbauen, Sie überallhin begleitet.

> Sechs Werttypen und eine rekursive Regel. Lernen Sie, sie zu sehen, und jede API-Antwort der Welt hört auf, eine Wand aus Satzzeichen zu sein, und wird zu etwas, das Sie lesen können.

Nächstes Mal: Datum und Uhrzeit in JSON — der Typ, den es nicht gibt, und die zwei Konventionen, die alle stattdessen verwenden.

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 —