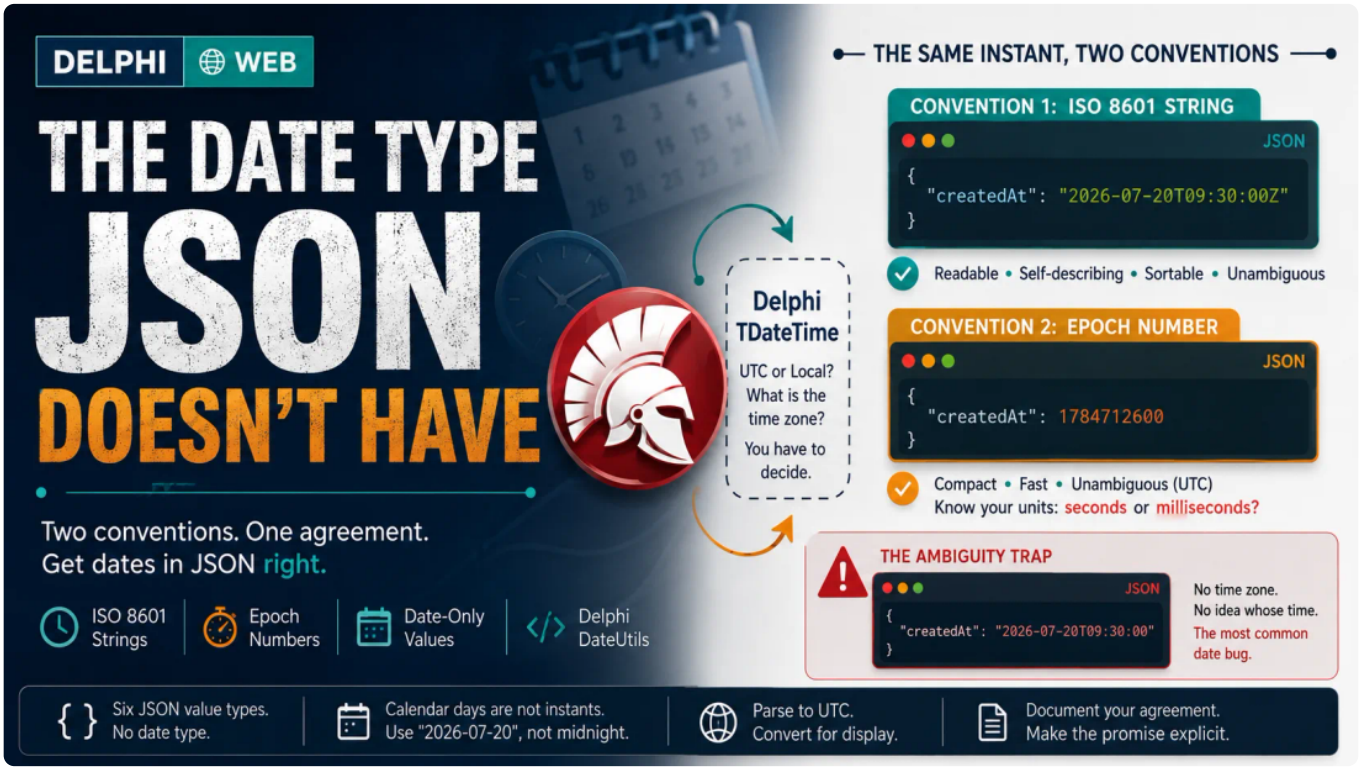
NEWS · JULY 20, 2026

Der Datumstyp, den JSON nicht hat

Teil zwei der JSON-Serie für Delphi-Entwickler — warum JSON keinen Datumstyp hat, welche zwei Konventionen stattdessen alle verwenden, was sie jeweils kosten und wie man ISO 8601 mit System.DateUtils korrekt liest und schreibt.

By Dr. Holger Flick



In Delphi ist ein Datum ein *Ding*. Sie deklarieren `Birthday: TDate`, der Compiler weiß, was das ist, der Debugger zeigt es Ihnen ordentlich an, und `IncMonth` macht am 31. das Richtige. `TDate`, `TTime` und `TDateTime` sind echte Typen der Sprache — Sie reichen sie herum wie Integer, und das Typsystem deckt Ihnen den Rücken.

Dann serialisieren Sie einen davon nach JSON, und all das löst sich in Luft auf. Denn JSON hat, wie wir am Ende von Teil eins gesehen haben, genau sechs Wertetypen — String, Number, Boolean, null, Object, Array — und keiner davon ist ein Datum. Es gibt kein `"date"`-Schlüsselwort, kein `2026-07-20`-Literal, nichts in der Grammatik, das sagt: „Dieser Wert ist ein Zeitpunkt." Das Format hat schlicht keine Meinung dazu.

Ein Datum in JSON ist also überhaupt kein Typ. Es ist eine **Konvention**: ein String oder eine Zahl, von der beide Programme außerhalb des Formats vereinbart haben, sie als Zeitpunkt zu interpretieren. Stimmt diese Vereinbarung, sind Daten langweilig. Stimmt sie nicht — und genau hier wohnen die Bugs —, dann trägt Ihre Rechnung für alle östlich von Ihnen ein Datum zu früh, oder Ihre Zeitstempel springen ins Jahr 2033, weil jemand Millisekunden als Sekunden gelesen hat.

In diesem Beitrag geht es um diese Vereinbarung: die zwei Konventionen, die die Welt tatsächlich verwendet, was sie jeweils kosten und wie Sie beide in Delphi mit `System.DateUtils` korrekt erzeugen und einlesen.

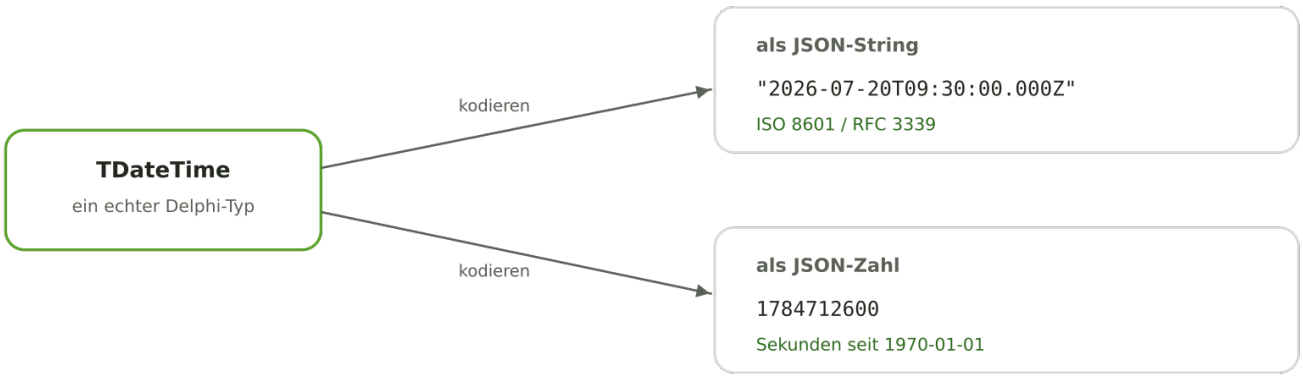
Warum es keinen Datumstyp gibt (und warum das Absicht war)

Es liegt nahe, den fehlenden Datumstyp für ein Versäumnis zu halten, aber er folgt direkt aus dem, was JSON sein sollte.

JSONs Grammatik ist bewusst winzig — klein genug, dass ein korrekter Parser die Arbeit eines Nachmittags ist, was genau der Grund ist, warum jede Sprache so schnell einen guten bekommen hat. Ein Datumstyp hätte das zunichtegemacht. Daten sind wirklich schwierig: Kalender, Schaltsekunden, Zeitzonendatenbanken, die sich mehrmals im Jahr ändern, und die Frage, ob „2026-07-20" ein Moment oder eine Beschriftung ist. Irgendetwas davon in das Wire-Format einzubacken hätte bedeutet, dass jeder Parser in jeder Sprache eine Kalenderimplementierung mitbringt — und sich für immer auf sie einigt.

Also erwähnt RFC 8259, der aktuelle JSON-Standard, Daten schlicht nicht. Er definiert sechs Wertetypen und hört auf. Die Darstellung eines Zeitstempels bleibt der Anwendung überlassen — das kostet wirklich etwas, ist aber ein bewusster Handel: JSON blieb klein, und die Datumsbehandlung wurde zu einer Entscheidung, die Sie treffen, statt zu einer, die Sie erben.

Damit bleiben Ihnen zwei Materialien, denn ein Datum muss als einer der sechs Typen ankommen. In der Praxis werden nur zwei verwendet.



Ein TDateTime, zwei Konventionen — und JSON selbst befürwortet keine davon

Beide Kästen rechts enthalten denselben Zeitpunkt. Für JSON ist keiner davon „korrekter" — das Format sieht einen String und eine Zahl, mehr nicht. Der Unterschied liegt vollständig darin, worauf sich die beiden Seiten geeinigt haben, und diese Einigung festzulegen ist Ihre Aufgabe.

Konvention 1: der ISO-8601-String

Das ist die Variante, zu der Sie standardmäßig greifen sollten, und die, die die überwältigende Mehrheit moderner APIs verwendet.

ISO 8601 ist der internationale Standard für Datum und Uhrzeit als Text. Die Form, die Ihnen überall begegnet, ist `2026-07-20T09:30:00Z`: Jahr-Monat-Tag, ein `T` als Trenner, Stunde:Minute:Sekunde und eine Zeitzonenangabe. Das abschließende `Z` — gesprochen „Zulu" — bedeutet UTC.

In der Praxis implementieren die meisten APIs tatsächlich RFC 3339, den die Spezifikation als „a profile of the ISO 8601 standard" und „a conformant subset of the ISO 8601 extended format" beschreibt. Dieser Unterschied wiegt schwerer, als er klingt. Vollständiges ISO 8601 ist ein ausuferndes Regelwerk, das auch Wochendaten (`2026-W30-1`), Ordinaldaten (`2026-201`) und das Weglassen von Trennzeichen erlaubt. RFC 3339 wirft fast all das weg und legt eine eindeutige Form fest — er „requires the 'T' to avoid ambiguity". Wenn jemand sagt „wir verwenden ISO 8601", meint er fast immer die RFC-3339-Teilmenge.

Auf den Offset lohnt sich besondere Sorgfalt. RFC 3339 zieht zudem eine semantische Linie, die die meisten übersehen: `Z` und `+00:00` bedeuten „UTC ist der bevorzugte Bezugspunkt", während `-00:00` gezielt heißt: „Die Zeit in UTC ist bekannt, der Offset zur Ortszeit aber nicht."

> **Der Fall, der wirklich zubeißt: gar kein Offset**
> Ein String wie `"2026-07-20T09:30:00"` — kein `Z`, kein `+02:00` — ist die häufigste Einzelursache für Datums-Bugs in JSON-APIs. Er ist ein *naiver* Zeitstempel: Er nennt eine Uhrzeit, ohne zu sagen, wessen
>
> Uhr. Der Sender meint meist seine eigene Ortszeit; der Empfänger nimmt meist UTC an oder seine eigene Ortszeit, und verschiedene Bibliotheken entscheiden unterschiedlich. Beide Systeme verhalten sich „vernünftig", und der Wert verschiebt sich stillschweigend um Stunden. Wenn Sie das Format bestimmen, geben Sie immer einen Offset mit.

Was es kostet. Strings sind größer als Zahlen — grob 24 Bytes gegenüber 10 — und Text zu parsen ist

langsamer, als eine Ganzzahl zu lesen. Für die allermeisten Anwendungen ist dieser Unterschied neben dem HTTP-Overhead bedeutungslos.

Was Sie bekommen. Eine ganze Menge, und deshalb hat sich diese Konvention durchgesetzt:

- **Er ist lesbar.** Sie sehen einer Antwort das Datum an. Wenn ein Support-Ticket ein rohes Payload enthält, zählt das.
- **Er ist eindeutig**, sofern der Offset vorhanden ist — der Wert trägt seinen Zeitzonenkontext selbst mit sich.
- **Er sortiert als Text korrekt.** Weil die Felder von groß nach klein mit fester Breite laufen, *ist* die lexikografische Reihenfolge bei UTC-Zeitstempeln die chronologische. Datenbanken, Log-Tools und ein schlichtes `ORDER BY` bekommen das geschenkt.
- **Er kann mehr ausdrücken als einen Zeitpunkt.** `"2026-07-20"` für sich ist ein legitimer reiner Datumswert — ein Konzept, das die Epochen-Zahl überhaupt nicht ausdrücken kann, wovon ein eigener Abschnitt weiter unten handelt.

Konvention 2: die Epochen-Zahl

Die Alternative ist eine schlichte JSON-Zahl: die Anzahl der seit einem festen Bezugspunkt vergangenen Zeiteinheiten, fast immer die Unix-Epoche 1970-01-01T00:00:00Z.

Das ist kompakt, trivial vergleichbar und hinsichtlich der Zeitzone von Natur aus eindeutig — ein Epochenwert wird immer von einem UTC-Zeitpunkt aus gemessen, es gibt also keinen Offset zu vergessen. Das passt gut zu Maschine-zu-Maschine-Verkehr, hochfrequenter Telemetrie und JWT-Claims, wo niemand das Payload von Hand liest.

Es hat zwei scharfe Kanten, und beide sind in der Produktion verbreitet.

Sekunden oder Millisekunden? Das ist die große Frage, denn JSON gibt Ihnen keine Möglichkeit, es zu

erkennen. Unix-Zeit wird klassisch in Sekunden gezählt, `1784712600` ist also ein Moment im Juli 2026. JavaScript aber arbeitet in Millisekunden — `Date.now()` liefert `1784712600000`. Beides ist „ein Unix-Zeitstempel". Geben Sie einen Millisekundenwert an etwas, das Sekunden erwartet, landen Sie zehntausende Jahre in der Zukunft; umgekehrt landen Sie im Januar 1970. Die Zahl trägt keinen Hinweis darauf, welche von beiden sie ist — Sie müssen es wissen, und der einzige Ort, an dem dieses Wissen lebt, ist Ihre Dokumentation.

Genauigkeit. JSON-Zahlen sind ein einziger Typ ohne Unterscheidung zwischen Ganzzahl und Gleitkomma,

und RFC 8259 ist deutlich in der Konsequenz: „numbers that are integers and are in the range [-(2\\-

53)+1, (2\\53)-1] are interoperable", weil Implementierungen verbreitet IEEE-754-Double-Precision verwenden.

Sekundengenaue Zeitstempel sind weit von dieser Grenze entfernt. Millisekunden- und Mikrosekundenwerte haben mehr Luft, als viele befürchten — aber sobald eine Sprache jede JSON-Zahl in ein Double parst, wie JavaScript es tut, beginnt genau bei großen Ganzzahlen der stille Präzisionsverlust.

ISO-8601-String	`"2026-07-20T09:30:00Z"`	selbstbeschreibend
Unix-Sekunden	`1784712600`	Einheit ungenannt
Unix-Millisekunden	`1784712600000`	Einheit ungenannt
naiver String	`"2026-07-20T09:30:00"`	wessen Uhr?

Derselbe Zeitpunkt, vier Kodierungen — und nur der String sagt, was er bedeutet

Die ersten drei Zeilen sind derselbe Zeitpunkt in drei Schreibweisen. Sehen Sie sich die rechte Spalte an: Nur die oberste Zeile sagt einem Leser — Mensch oder Maschine —, was sie tatsächlich bedeutet. Die beiden Zahlenzeilen sind ohne Dokumentation nicht unterscheidbar, und die unterste Zeile ist die Mehrdeutigkeitsfalle von oben, die am menschenfreundlichsten aussieht und am gefährlichsten ist.

> **Was also nehmen?**
> Meine Empfehlung, und es ist die, auf die sich der Großteil der Branche geeinigt hat: **standardmäßig**
>
> **ISO-8601-Strings mit explizitem Offset**. Lesbarkeit und Selbstbeschreibung zahlen sich beim ersten
>
> Debuggen eines Payloads aus, und der Größenunterschied fällt über HTTP selten ins Gewicht. Greifen Sie zu Epochen-Zahlen, wenn Sie einen konkreten Grund haben — sehr hohes Nachrichtenaufkommen, eingebettete oder bandbreitenbeschränkte Verbindungen, oder eine Spezifikation, die es vorschreibt, wie JWT es für `exp` und `iat` tut. Und wenn Sie eine API konsumieren, ist es ohnehin nicht Ihre Entscheidung:
>
> Lesen Sie deren Dokumentation und halten Sie sich exakt daran.

Wie Delphi Daten darstellt — und warum die Diskrepanz real ist

Vor dem Code lohnt sich Genauigkeit darüber, *wovon* Sie eigentlich konvertieren, denn Delphis Modell

unterscheidet sich wirklich von JSONs.

`TDateTime` ist in `System.pas` als `TDateTime = type Double` deklariert — eine Gleitkommazahl. Der ganzzahlige Teil zählt Tage seit dem Epochentag 30.12.1899, der Nachkommateil ist der verstrichene Bruchteil eines Tages: `0.5` ist Mittag, `0.75` ist 18 Uhr. `TDate` und `TTime` sind als eigenständige Typen über demselben Double deklariert (`TDate = type TDateTime`), was Ihnen Absicht zur Compile-Zeit sowie besseres Verhalten in Debugger und RTTI verschafft — zur Laufzeit sind es aber alles dieselben Zahlen.

Daraus folgt etwas, das man verinnerlichen sollte: **Ein `TDateTime` trägt keine Zeitzone.** Es ist eine nackte Zahl

auf einem Zeitstrahl, deren Bedeutung — ist das Ortszeit oder UTC? — ausschließlich in Ihrem Kopf und Ihren Namenskonventionen lebt. JSONs ISO-8601-Konvention hingegen *kann* einen Offset tragen. Die Konvertierung

ist also nicht bloß Formatierung; sie ist der Moment, in dem Sie eine Information beisteuern müssen, die der Wert nie hatte.

Die Konvertierung fügt Bedeutung hinzu, die das rohe Double nie trug

Der gestrichelte Kasten in der Mitte ist der springende Punkt. Jede Funktion im nächsten Abschnitt stellt Ihnen im Grunde diese Frage, meist über einen booleschen Parameter, den man leicht gedankenlos auf seinem Standardwert lässt. Genau daher kommen die um Stunden falschen Werte.

ISO 8601 schreiben und lesen mit System.DateUtils

Delphi liefert alles Nötige in `System.DateUtils` mit. Die beiden entscheidenden Funktionen sind `DateToISO8601` und `ISO8601ToDate`, dazu eine `Try…`-Variante für unsichere Eingaben.

Ihre Deklarationen sagen schon das meiste:

```
function DateToISO8601(const ADate: TDateTime; AInputIsUTC: Boolean = True): string;
function ISO8601ToDate(const AISODate: string; AReturnUTC: Boolean = True): TDateTime;
function TryISO8601ToDate(const AISODate: string; out Value: TDateTime;
  AReturnUTC: Boolean = True): Boolean;
```

Beachten Sie, dass beide auf `True` voreingestellt sind — beide gehen also davon aus, dass Sie in **UTC** arbeiten.

Diese Voreinstellung ist sinnvoll, aber sie ist nur dann korrekt, wenn der übergebene Wert tatsächlich UTC ist.

Einen Zeitstempel schreiben

Der Normalfall ist ein Wert, den Sie bereits in UTC halten und direkt hinausschreiben:

```
uses
  System.DateUtils, System.JSON;

var
  LNow: TDateTime;
  LObj: TJSONObject;
begin
  LNow := TTimeZone.Local.ToUniversalTime(Now);   // Now ist Ortszeit – erst konvertieren

  LObj := TJSONObject.Create;
  try
    LObj.AddPair('created_at', DateToISO8601(LNow));  // AInputIsUTC = True
    Writeln(LObj.ToJSON);
    // {"created_at":"2026-07-20T09:30:00.000Z"}
  finally
    LObj.Free;
  end;
end;
```

Die entscheidende Zeile ist die Konvertierung. `Now` liefert **Ortszeit**; sie direkt an `DateToISO8601` mit der Voreinstellung `AInputIsUTC = True` zu übergeben, würde einer lokalen Ablesung ein `Z` aufdrücken und behaupten, sie

sei UTC — der naive Zeitstempel, direkt an der Quelle erzeugt. `TTimeZone.Local.ToUniversalTime` konvertiert sauber und berücksichtigt die Sommerzeit für das jeweilige Datum.

Alternativ übergeben Sie den lokalen Wert und sagen die Wahrheit darüber:

```
LObj.AddPair('created_at', DateToISO8601(Now, False));
// z. B. {"created_at":"2026-07-20T11:30:00.000+02:00"}
```

Mit `AInputIsUTC = False` ermittelt die RTL den lokalen UTC-Offset für dieses Datum und hängt ihn statt `Z` an. Beide Strings bezeichnen denselben Zeitpunkt; beide sind gültiges RFC 3339. Wählen Sie einen und bleiben Sie konsistent.

Einen Zeitstempel lesen

Für Eingaben, die Sie kontrollieren, genügt `ISO8601ToDate`; bei fehlerhaftem Text löst es eine Exception aus. Für alles, was über das Netz kommt, ist `TryISO8601ToDate` die bessere Wahl:

```
var
  LValue: TJSONValue;
  LWhen: TDateTime;
begin
  LValue := LObj.GetValue('created_at');
  if (LValue <> nil) and TryISO8601ToDate(LValue.Value, LWhen) then
    // LWhen ist UTC (AReturnUTC ist auf True voreingestellt)
    Writeln(DateTimeToStr(TTimeZone.Local.ToLocalTime(LWhen)))
  else
    Writeln('Zeitstempel fehlt oder ist ungültig');
end;
```

Zwei Dinge sind hervorzuheben. `LValue.Value` liefert Ihnen den entschlüsselten String-Inhalt. Und das Parse-Ergebnis ist standardmäßig UTC — die Umrechnung in Ortszeit zur Anzeige ist ein eigener, bewusster Schritt, und genau diese Disziplin verhindert Zeitzonen-Drift: **in UTC parsen, in UTC speichern und rechnen, erst zur Anzeige für Menschen in Ortszeit umrechnen.**

Für feinere Kontrolle gibt es eine Überladung mit einer Menge:

```
LWhen := ISO8601ToDate(LText, [ioNoTZIsLocal]);
```

`ioNoTZIsLocal` behandelt den naiven Fall explizit — hat der eingehende Text keine Zeitzone, wird er als Ortszeit statt als UTC verstanden. `ioReturnUTC` steuert, ob das Ergebnis als UTC zurückkommt. Wenn Sie eine API konsumieren müssen, die naive Zeitstempel liefert, ist das der Schalter, mit dem Sie Ihre Annahme im Code aussprechen, statt sie implizit zu lassen.

Reine Datumswerte sind ein anderes Problem

Dieser Fall verdient eine eigene Behandlung, denn das falsche Werkzeug erzeugt hier einen realen und erstaunlich häufigen Bug.

Manche Werte sind keine Zeitpunkte. Ein Geburtstag, ein Rechnungsdatum, ein Feiertag, ein Vertragsbeginn — das sind **Kalendertage**, keine Momente auf einem Zeitstrahl. Niemand wurde um `1990-03-14T00:00:00Z` geboren; er wurde am 14. März geboren, und das gilt unabhängig davon, wo Sie gerade stehen.

Senden Sie einen solchen Wert als Zeitstempel, hängen Sie ihm eine Uhrzeit und eine Zeitzone an, die nie zu ihm gehörten. Mitternacht ist dabei die denkbar schlechteste Wahl, weil sie genau auf der Grenze liegt: `"1990-03-14T00:00:00Z"` in Ortszeit in Auckland ist der 14. mittags — in Ordnung —, aber dieselbe Mitternachts-UTC-Konvention auf einen in Berlin erfassten Wert angewandt kann bei der Rückrechnung auf

dem 13. landen. Das Datum verschiebt sich um einen Tag, je nachdem, wer hinsieht. Das ist der klassische Off-by-one-Day-Bug — und er ist vollständig hausgemacht.

Das korrekte Wire-Format ist die reine Datumsform `"2026-07-20"` — die RFC 3339 als `full-date` definiert und die weder Uhrzeit noch Offset trägt, eben weil es keine gibt.

Und genau hier lässt die RTL Sie allein. `DateToISO8601` hat keinen Nur-Datum-Modus: Es gibt immer die vollständige Form `yyyy-mm-ddThh:nn:ss.zzzZ` aus. Sein Formatstring ist in der Implementierung fest als `'%.4d-%.2d-%.2dT%.2d:%.2d:%.2d.%.3dZ'` verdrahtet, sodass selbst ein reines `TDate` — dessen Nachkommateil null ist — als Mitternachts-Zeitstempel mit `Z` und drei Null-Millisekunden herauskommt. Übergeben Sie ein `TDate`, bekommen Sie genau das zurück, was Sie nicht senden sollten.

Für reine Datumswerte verwenden Sie stattdessen `FormatDateTime`:

```
uses
  System.SysUtils, System.DateUtils;

var
  LBirthday: TDate;
begin
  LBirthday := EncodeDate(1990, 3, 14);
  Writeln(FormatDateTime('yyyy-mm-dd', LBirthday));   // 1990-03-14
end;
```

> **FormatDateTime ist locale-abhängig — pinnen Sie es fest**
> `FormatDateTime` verwendet die Formateinstellungen der Anwendung. `'yyyy-mm-dd'` wird zwar respektiert, aber die sichere Gewohnheit bei Wire-Formaten ist, invariante Einstellungen explizit zu übergeben, damit keine Benutzer-Locale je die Ausgabe beeinflussen kann: ``pascal var  LFmt: TFormatSettings; begin  LFmt := TFormatSettings.Invariant;  Writeln(FormatDateTime('yyyy-mm-dd', LBirthday, LFmt)); end; `` Verfahren Sie bei einem reinen Zeitwert genauso: FormatDateTime('hh:nn:ss', LTime, LFmt)`.

Das Zurücklesen eines reinen Datums-Strings ist unkompliziert, da die Form fest ist:

```
var
  LDate: TDate;
begin
  LDate := EncodeDate(
    StrToInt(Copy(LText, 1, 4)),
    StrToInt(Copy(LText, 6, 2)),
    StrToInt(Copy(LText, 9, 2)));
end;
```

Das wirkt grobschlächtig, und das ist es auch — aber für ein festes 10-Zeichen-Format ist es ehrlich und ohne Abhängigkeiten. (`TryISO8601ToDate` akzeptiert auch ein blankes `"1990-03-14"` und liefert Ihnen ein `TDateTime` um Mitternacht; wenn Sie diesen Weg gehen, achten Sie bewusst auf die UTC-Flags, damit Mitternacht nicht wandert.)

Epochen-Zahlen, wenn Sie sie brauchen

Wenn die API, mit der Sie sprechen, Epochen-Zahlen verwendet, deckt `System.DateUtils` auch das ab:

```
function DateTimeToUnix(const AValue: TDateTime; AInputIsUTC: Boolean = True): Int64;
function UnixToDateTime(const AValue: Int64; AReturnUTC: Boolean = True): TDateTime;
```

Diese arbeiten in **Sekunden**, der klassischen Unix-Konvention. Spricht Ihr Gegenüber JavaScript, sendet es sehr wahrscheinlich Millisekunden, und Sie müssen selbst überbrücken:

```
// Millisekunden schreiben
LObj.AddPair('ts', TJSONNumber.Create(DateTimeToUnix(LUtcNow) * 1000));

// Millisekunden lesen
LWhen := UnixToDateTime(LMillis div 1000);
```

Das Wegdividieren der Millisekunden verwirft die Genauigkeit unterhalb einer Sekunde. Brauchen Sie sie wirklich, verwenden Sie stattdessen `IncMilliSecond(UnixDateDelta, LMillis)` — `UnixDateDelta` ist die RTL-Konstante für die 1970er-Epoche als `TDateTime`.

Wo die RTL zu wünschen übrig lässt — und welche Bibliotheken die Lücke füllen

Die eingebauten Funktionen sind solide, korrekt und frei von Abhängigkeiten, und für handgebautes JSON reichen sie wirklich aus. Ihre Grenzen sollte man aber offen benennen, denn Sie werden allen dreien begegnen.

Sie schreibt immer Millisekunden. `DateToISO8601` gibt bedingungslos `.000` aus. Das ist gültiges RFC 3339 und entspricht dem, was JavaScripts `JSON.stringify` erzeugt — `Date.prototype.toJSON` liefert exakt `1975-08-19T23:15:30.000Z` —, die RTL befindet sich also in guter Gesellschaft. Manche APIs und manche strengen Validatoren erwarten jedoch `2026-07-20T09:30:00Z` ohne Nachkommaanteil, und die RTL bietet dafür keinen Schalter. Dann formatieren Sie den String selbst oder schneiden nachträglich zu.

Sie hat keinen Nur-Datum-Modus. Wie oben gezeigt, bekommt ein `TDate` einen vollständigen Zeitstempel, ob Sie wollen oder nicht. Die Typunterscheidung, die Delphi in der Sprache so sorgfältig pflegt, fällt an der JSON-Grenze weg.

Sie arbeitet Wert für Wert. Diese Funktionen konvertieren ein `TDateTime`. Einen Objektgraphen zu serialisieren heißt, die Traversierung selbst zu schreiben, pro Feld die Konvention zu entscheiden — und dasselbe für den Rückweg noch einmal.

Genau an diesem letzten Punkt verdienen Drittanbieter-Bibliotheken ihr Geld. Delphi Neon von Paolo Rossi (Apache-2.0) ist ein RTTI-basierter Serialisierer, der ganze Objekte, Records und generische Collections von und nach JSON abbildet. Seine Datumsbehandlung ist gerade an der obigen Lücke aufschlussreich: Sie hält *getrennte* Pfade für `TDate` und `TDateTime` bereit — `TJSONUtils.DateToJSON` formatiert ein `TDate` als `YYYY-MM-DD`, während Datums-/Zeitwerte über `DateToISO8601` laufen. Neon bewahrt also die Unterscheidung, die die RTL einebnet — und bietet eine `UseUTCDate`-Konfigurationsoption, sodass die UTC-Entscheidung einmal in der Konfiguration fällt statt an jeder Aufrufstelle.

> **Der größere Zusammenhang**
> Neon ist nicht die einzige Möglichkeit, und nichts davon ist Delphi-spezifisch. mORMot 2 enthält eine sehr schnelle JSON-Engine mit eigenen Datumskonventionen, und JsonDataObjects ist ein kompakter, performanter Parser in einer einzigen Unit — beide Open Source. Außerhalb von Delphi wird dasselbe Problem genauso gelöst: Jackson in Java und System.Text.Json in .NET setzen beide standardmäßig auf ISO-8601-Strings, Pythons `datetime.isoformat()` erzeugt dieselbe Form, und Gos `encoding/json` serialisiert `time.Time` als RFC 3339. Diese Konvergenz ist die eigentliche Beruhigung: Wählen Sie ISO 8601 mit explizitem Offset, und Sie sprechen denselben Dialekt wie alle anderen auf der Leitung.

Fazit

JSONs fehlender Datumstyp ist keine Lücke, die gestopft werden müsste — er ist eine Entscheidung, die an Sie weitergereicht wurde, und die ganze Kunst besteht darin, sie bewusst zu treffen statt versehentlich.

- **Ein Datum in JSON ist eine Konvention, kein Typ.** Zwei Programme einigen sich darauf, dass ein bestimmter String oder eine Zahl einen Moment bedeutet. Schreiben Sie diese Einigung auf; sie ist der einzige Ort, an dem die Bedeutung lebt.
- **Standardmäßig ISO 8601 mit explizitem Offset.** Lesbar, sortierbar, selbstbeschreibend und universell verstanden. Epochen-Zahlen, wenn Volumen oder eine Spezifikation es verlangen — und dokumentieren Sie Sekunden gegen Millisekunden, denn die Zahl selbst verrät es niemandem.
- **Senden Sie nie einen naiven Zeitstempel.** Kein `Z`, kein Offset, keine Ahnung. Der häufigste Datums-Bug in JSON-APIs und der am leichtesten vermeidbare.
- **Kalendertage sind keine Zeitpunkte.** Senden Sie einen Geburtstag als `"1990-03-14"`. Die RTL nimmt Ihnen das nicht ab — `DateToISO8601` schreibt immer einen vollständigen Zeitstempel —, verwenden Sie also `FormatDateTime('yyyy-mm-dd', …)` mit invarianten Einstellungen.
- **In UTC parsen, in UTC rechnen, erst zur Anzeige in Ortszeit umrechnen.** Die Parameter `AInputIsUTC` und `AReturnUTC` sind auf `True` voreingestellt; stellen Sie sicher, dass diese Voreinstellung über Ihre Daten die Wahrheit sagt.

> Delphi gibt Ihnen einen Typ, und der Compiler setzt ihn durch. JSON gibt Ihnen einen String und ein Versprechen. Die Bugs wohnen in der Lücke dazwischen, und das Mittel dagegen ist, das Versprechen laut auszusprechen — im Format und in Ihrer Dokumentation.

Nächstes Mal schreiben wir endlich echten Code gegen die Klassen selbst: `System.JSON` und die kleine Familie von Typen — `TJSONObject`, `TJSONArray`, `TJSONString`, `TJSONNumber`, `TJSONBool`, `TJSONNull` —, die exakt die sechs Wertetypen aus Teil eins abbilden. JSON bauen, parsen und durchlaufen mit nichts als dem, was schon im Karton liegt.

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 —