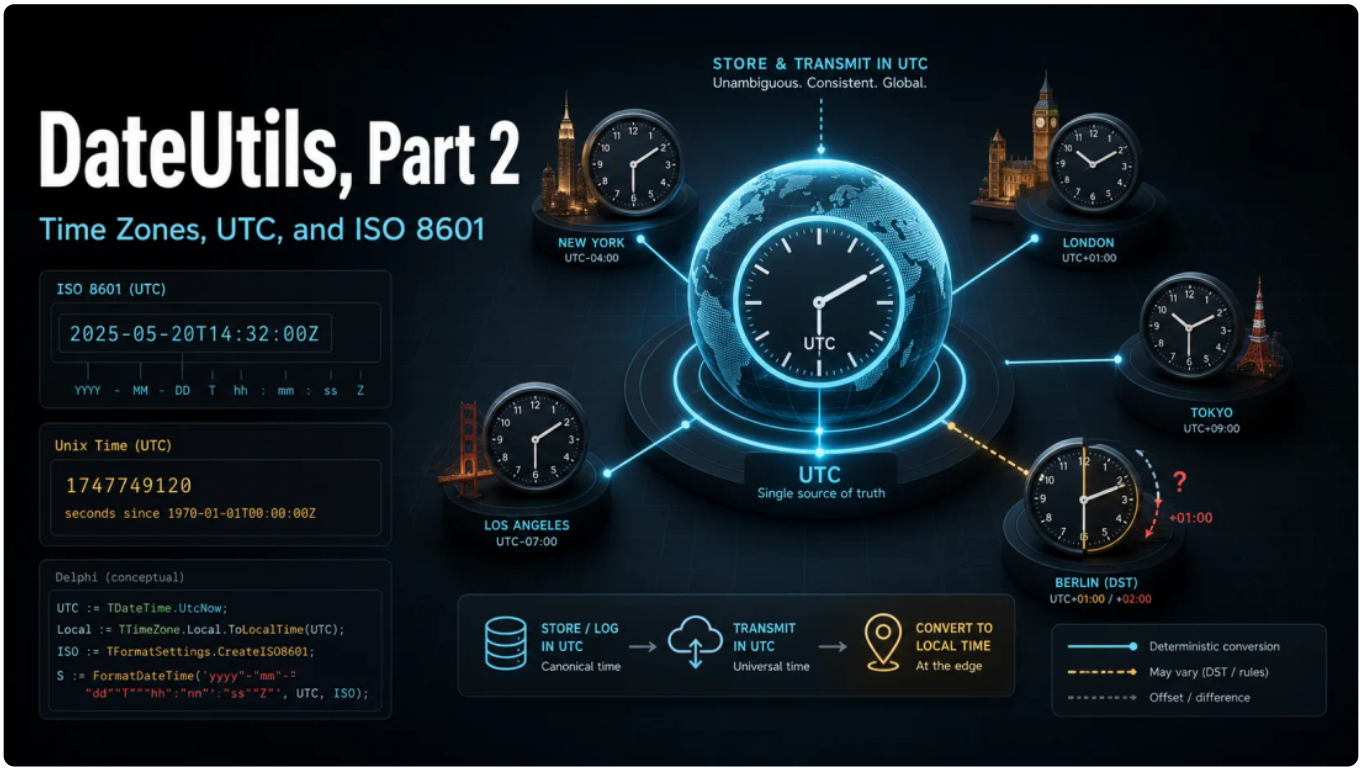
NEWS · JULY 30, 2026

# DateUtils, Teil 2: Zeitzonen, UTC und ISO 8601

By Dr. Holger Flick




Es gibt eine Kategorie von Fehlern, die fast schon ein Initiationsritus ist. Die App funktioniert einwandfrei auf Ihrem Rechner. Sie funktioniert im Test. Dann wird sie ausgeliefert, und ein Kunde drei Zeitzonen entfernt meldet, dass Termine eine Stunde daneben liegen, ein Bericht die Daten von gestern zeigt oder ein \"created at\"-Zeitstempel in der Zukunft liegt. Jeder, der Software gebaut hat, die Zeitzonen überquert, ist diesem Gespenst begegnet.

In Teil 1 haben wir die alltägliche Datumsarithmetik mit `System.DateUtils` sauber und lesbar gemacht — dabei aber stillschweigend angenommen, dass jedes Datum in *derselben* Zeitzone lebt. In dem Moment, in dem das nicht mehr stimmt — ein Server, der in UTC läuft, Benutzer über die ganze Welt verteilt, ein Zeitstempel aus einer REST-API — brauchen Sie einen disziplinierten Umgang mit Zonen, sonst ziehen diese Gespenster ein.

Die gute Nachricht: Die moderne RTL bietet genau dafür hervorragende Werkzeuge, und sobald Sie eine einfache Regel übernehmen, machen sie das gesamte Problem beherrschbar. **In Teil 2 geht es darum, Zeitzonen richtig zu behandeln**: die goldene Regel (\"UTC speichern, lokal anzeigen\"), die Klasse `TTimeZone` für die Konvertierung zwischen beiden und die Austauschformate — **ISO 8601** und **Unix-Zeit** —, die einen Zeitstempel eindeutig halten, während er zwischen Systemen unterwegs ist. Das ist der Teil der Datumsverarbeitung, der für die Korrektheit wirklich zählt. Treiben wir das Gespenst aus.

## Die eine Regel, die die meisten Zeitzonen-Bugs verhindert

Vor jeder API kommt das Prinzip — denn die Werkzeuge helfen nur, wenn Sie sie innerhalb eines soliden Modells einsetzen. Nahezu jedes Zeitzonen-Desaster lässt sich darauf zurückführen, dass eine \"nackte\" Lokalzeit gespeichert oder übertragen wurde, ohne festzuhalten, in *welcher* Zone sie galt. Die Lösung ist eine Disziplin, auf die sich die gesamte Branche geeinigt hat:

> **Speichern und übertragen Sie UTC. Konvertieren Sie in die Lokalzeit des Benutzers nur an den Rändern — bei der Anzeige oder bei der Eingabe.**

UTC — Coordinated Universal Time — ist die eine, globale, zonenlose Referenzuhr. Sie springt nie vor und stellt sich nie zurück. Wenn jeder Zeitstempel in Ihrer Datenbank und jeder Zeitstempel auf der Leitung UTC ist, dann *funktionieren* Vergleiche, Sortierungen und \"Tage dazwischen\" einfach, weil alles an derselben Uhr gemessen wird. Lokalzeit wird zu einer Frage der *Darstellung*, die ganz zum Schluss angewendet wird.




UTC überall speichern; nur an den Rändern für Anzeige und Eingabe in Lokalzeit konvertieren

Das Diagramm ist die gesamte Architektur: UTC lebt in der Mitte, und die Konvertierung in Lokalzeit passiert nur an den zwei Rändern, an denen ein Mensch beteiligt ist. Übernehmen Sie das, und der Rest dieses Beitrags ist nur noch \"wie man diese zwei Konvertierungen korrekt durchführt\".

## Die aktuelle Zeit in UTC ermitteln

Beginnen wir an der Quelle. Das vertraute `Now` liefert die *lokale* Zeit — gut für die Anzeige, falsch für die Speicherung. Die RTL liefert Ihnen UTC direkt. Die sauberste moderne Form ist der Record-Helper `TDateTimeHelper`, den die Unit zu `TDateTime` hinzufügt:

```
uses
  System.DateUtils;

var
  StampUtc: TDateTime;
begin
  StampUtc := TDateTime.NowUTC;   // aktueller Zeitpunkt, in UTC
end;
```

`TDateTime.NowUTC` ist Ihr Aufruf für \"Zeitstempel zum Speichern\" — verwenden Sie ihn überall dort, wo Sie sonst mit `Now` festgehalten hätten, wann etwas passiert ist. (Es gibt auch die klassische Funktionsform über das Betriebssystem, aber `NowUTC` liest sich am besten und steht direkt am Typ zur Verfügung.)

## Zwischen lokal und UTC konvertieren mit `TTimeZone`

Für die Konvertierung *vorhandener* Werte — ein UTC-Zeitstempel aus der Datenbank in Lokalzeit für die Anzeige oder eine Benutzereingabe in die andere Richtung — stellt die RTL die Klasse `TTimeZone` bereit. Ihre Klassen-Property `Local` liefert die Zone des aktuellen Benutzers, und zwei Methoden erledigen die Konvertierungen:

```
var
  Utc, Local: TDateTime;
begin
  // UTC aus der Datenbank ' Lokalzeit für die UI
  Local := TTimeZone.Local.ToLocalTime(Utc);

  // vom Benutzer eingegebene Lokalzeit ' UTC zum Speichern
  Utc := TTimeZone.Local.ToUniversalTime(Local);
end;
```

Das ist für die meisten Anwendungen bereits die ganze Konvertierungsgeschichte: `ToLocalTime` auf dem Weg nach draußen, `ToUniversalTime` auf dem Weg nach drinnen. `TTimeZone` legt außerdem den Offset selbst offen, was für Anzeige oder Logging praktisch ist:

```
ShowMessage(TTimeZone.Local.UtcOffset.ToString);      // z. B. '-04:00:00' als TTimeSpan
ShowMessage(TTimeZone.Local.Abbreviation);            // z. B. 'GMT-4'
ShowMessage(TTimeZone.Local.GetUtcOffset(SomeDate).ToString); // Offset *für ein bestimmtes Datum*
```

Beachten Sie, dass `GetUtcOffset` ein *Datum* entgegennimmt — was wichtiger ist, als es zunächst scheint, denn der Offset ist nicht konstant.

> **Die Sommerzeit ist der Grund, warum Sie *für ein bestimmtes Datum* konvertieren müssen**
> Der UTC-Offset einer Zone **ändert sich im Laufe des Jahres**, wo immer Sommerzeit (Daylight Saving Time) gilt — New York liegt im Januar bei `-05:00` und im Juli bei `-04:00`. Das ist die größte Einzelquelle subtiler Zeitfehler, und genau deshalb nehmen die `TTimeZone`-Konvertierungen den *konkreten* `TDateTime`-Wert entgegen, der konvertiert wird: Der Offset wird für den DST-Zustand *dieses* Datums berechnet, nicht als feste Zahl. Deshalb dürfen Sie auch nie \"konvertieren\", indem Sie einfach einen fest codierten Offset addieren — `SomeUtc + (5/24)` ist die halbe Jahreszeit über falsch. Gehen Sie immer über `ToLocalTime`/`ToUniversalTime`, die die Zonendatenbank des Betriebssystems nach dem richtigen Offset für das richtige Datum befragen. `TTimeZone` erlaubt Ihnen sogar, mit `GetLocalTimeType` die seltenen ungültigen/mehrdeutigen Zeiten zu erkennen, die exakt an den DST-Übergängen auftreten.




Der UTC-Offset ist nicht konstant — die Sommerzeit verschiebt ihn im Jahresverlauf

Das Diagramm zeigt, warum fest codierte Offsets scheitern: Dieselbe Zone liegt einen Teil des Jahres bei `-05:00` und einen anderen bei `-04:00`. Nur eine datumsbezogene Konvertierung wählt den richtigen Wert — und genau das erledigt `TTimeZone` für Sie.

## Austausch: ISO 8601, das Format für Datumsangaben auf der Leitung

Wenn ein Datum Ihr Programm verlässt — in JSON, eine URL, ein Log, ein anderes System —, ist der rohe `Double` eines `TDateTime` für den Empfänger bedeutungslos. Sie brauchen ein *textuelles* Format, das eindeutig, sortierbar und universell verstanden ist. Dieses Format ist ISO 8601 — das `2026-08-11T14:30:00.000Z`, das Sie aus jeder modernen API kennen. `DateUtils` konvertiert in beide Richtungen:

```
var
  S: string;
  D: TDateTime;
begin
  // TDateTime ' ISO-8601-String (Eingabe wird standardmäßig als UTC behandelt ' 'Z' am Ende)
  S := DateToISO8601(TDateTime.NowUTC);
  // z. B. '2026-08-11T14:30:00.000Z'

  // ISO-8601-String ' TDateTime (liefert standardmäßig UTC zurück)
  D := ISO8601ToDate('2026-08-11T14:30:00.000Z');
end;
```

Es gibt sogar eine noch elegantere Form am `TDateTime`-Helper — `SomeDate.ToISO8601` —, die die eigenständige Funktion spiegelt. Zwei Dinge daran lohnt es sich festzuhalten, denn genau hier stolpern die Leute.

> **Die UTC-Parameter sind das entscheidende Detail**
>
> Beide Funktionen nehmen einen Boolean entgegen, der die UTC-Eigenschaft des Wertes deklariert: `DateToISO8601(D, AInputIsUTC)` sagt, ob `D` *bereits* UTC ist, und `ISO8601ToDate(S, AReturnUTC)` sagt, ob Sie das *Ergebnis* in UTC wollen. Sie stehen standardmäßig auf `True`, was perfekt zur goldenen Regel passt — Sie halten ohnehin alles in UTC. Wenn Sie \"UTC speichern, lokal anzeigen\" befolgen, lassen Sie diese Parameter fast immer auf ihren Standardwerten und lassen das `Z`-Suffix mit dem String mitreisen. Für Strings ohne Zonenangabe erlaubt Ihnen die Überladung mit `TISO8601ToDateOptions` festzulegen, wie sie zu interpretieren sind.

Zur Validierung nicht vertrauenswürdiger Eingaben bevorzugen Sie die `Try`-Form, damit ein fehlerhafter String keine Exception auslöst:

```
if TryISO8601ToDate(UserSuppliedText, D) then
  UseIt(D)
else
  ComplainNicely;
```

## Austausch: Unix-Zeit, die Zahl, die niemals lügt

Das andere universelle Format ist die Unix-Zeit — eine einzelne Ganzzahl, die die Sekunden seit dem 1. Januar 1970 UTC zählt. Sie ist kompakt, sie ist von Natur aus UTC (keine Zone, die man falsch machen könnte), und ein riesiger Teil der Infrastruktur spricht sie. `DateUtils` konvertiert in beide Richtungen:

```
var
  Secs: Int64;
  D: TDateTime;
begin
  Secs := DateTimeToUnix(TDateTime.NowUTC);   // z. B. 1786552200
  D := UnixToDateTime(Secs);                   // zurück zu einem TDateTime (UTC)
end;
```

Wie die ISO-Funktionen tragen auch diese die Parameter `AInputIsUTC` / `AReturnUTC` (standardmäßig `True`) und fügen sich damit nahtlos in das UTC-überall-Modell ein. Greifen Sie zur Unix-Zeit, wenn Sie Kompaktheit brauchen oder mit einem System sprechen, das sie erwartet; greifen Sie zu ISO 8601, wenn Lesbarkeit für Menschen und selbstbeschreibende Zoneninformationen zählen (Logs, JSON-APIs). Beide sind korrekt — es ist eine Frage der Passung.

## Alles zusammen: die Rundreise

Hier ist die gesamte Disziplin in einem kurzen Ablauf — die Form, der fast jede wohlerzogene Anwendung folgt. Beachten Sie: UTC in der Mitte und lokal nur an den äußersten Enden.

```
// 1. Erfassen — Zeitstempel in UTC
Order.CreatedAt := TDateTime.NowUTC;

// 2. Übertragen — als ISO 8601 serialisieren (bleibt UTC, bekommt ein 'Z')
Json := DateToISO8601(Order.CreatedAt);

// 3. Anderswo empfangen — zurück nach UTC parsen
var Received := ISO8601ToDate(Json);

// 4. Anzeigen — zum Schluss in die lokale Zone des Betrachters konvertieren
ShowMessage(DateTimeToStr(TTimeZone.Local.ToLocalTime(Received)));
```

Jeder Schritt in der Mitte ist zonenfrei und eindeutig; die einzige Konvertierung ist das abschließende `ToLocal-Time` für den Menschen. Halten Sie sich daran, und das Gespenst \"die Termine sind eine Stunde daneben\" taucht schlicht nie auf.

## Die andere Seite: wo mehr als die RTL nötig ist

Ein Versprechen an die Leserschaft — ehrliche Eingrenzung. `TTimeZone.Local` liefert Ihnen die aktuelle Zone der *Maschine*, was für eine Desktop-Anwendung, die der Person davor Zeiten anzeigt, genau richtig ist. Zwei

Szenarien brauchen jedoch mehr:

- **Ein Server, der Zeiten für viele Benutzer in unterschiedlichen Zonen rendert.** Die lokale Zone der Maschine ist irrelevant; Sie brauchen die Zone jedes *Benutzers* (im Profil gespeichert) und müssen pro Benutzer konvertieren. Mit `TTimeZone.Local` aus der RTL allein geht das nicht — Sie brauchen die IANA-Zonen-ID jedes Benutzers und eine Bibliothek, die in eine *beliebige* benannte Zone konvertieren kann, nicht nur in die lokale.
- **Historische/zukünftige Genauigkeit über Änderungen der Zonenregeln hinweg.** Regierungen ändern DST-Regeln; robuste Verarbeitung bedeutet eine aktuelle IANA tz database.

> **Alternativen und Ergänzungen**

> Für die Konvertierung in beliebige benannte Zonen in Delphi bündelt die angesehene Open-Source-Bibliothek TZDB die IANA-Datenbank und klinkt sich in dieselbe `TTimeZone`-Abstraktion ein — eine hervorragende Ergänzung, wenn `TTimeZone.Local` nicht ausreicht. Über die Stacks hinweg erledigen dieselbe Aufgabe `TimeZoneInfo` und `DateTimeOffset` in .NET, Luxon oder die neue `Temporal`-API in JavaScript und `zoneinfo` in Python. Das *Prinzip* — UTC überall, Konvertierung an den Rändern, eine echte tz-Datenbank für benannte Zonen — ist in jedem davon identisch. Lernen Sie es einmal, hier.

## Fazit

Über beide Teile hinweg sind wir von lesbarer alltäglicher Datumsarithmetik zu korrekter zonenübergreifender Verarbeitung gekommen. Der Teil, der Sie vor echten Fehlern schützt:

- **Die goldene Regel: UTC speichern und übertragen; nur bei Anzeige und Eingabe in Lokalzeit konvertieren.** UTC ist die zonenlose Referenzuhr, die Vergleiche und Sortierungen vertrauenswürdig macht.
- `TDateTime.NowUTC` stempelt den aktuellen Zeitpunkt in UTC; `TTimeZone.Local.ToLocalTime` / `ToUniversalTime` konvertieren an den Rändern — immer *datumsbezogen*, weil die Sommerzeit den Offset im Jahresverlauf ändert. Addieren Sie niemals einen fest codierten Offset.
- **ISO 8601** (`DateToISO8601` / `ISO8601ToDate`, mit `Try` für nicht vertrauenswürdige Eingaben) und **Unix-Zeit** (`DateTimeToUnix` / `UnixToDateTime`) sind die eindeutigen Austauschformate; ihre `IsUTC`-Parameter passen standardmäßig zum UTC-überall-Modell.
- Für benutzerspezifische Zonen auf einem Server oder beliebige benannte Zonen kombinieren Sie die RTL mit einer Bibliothek auf Basis der IANA tz database.

> UTC speichern, lokal anzeigen, an den Rändern mit `TTimeZone` konvertieren — und ISO 8601 oder Unix-Zeit auf die Leitung legen. Diese Disziplin, zusammen mit den eingebauten Werkzeugen der RTL, macht Software über Zeitzonen hinweg tatsächlich korrekt.

Damit schließt die `DateUtils`-Serie — Teil 1 für die alltägliche Datumsarithmetik, dieser Teil für die zonenbewusste Korrektheit, die verteilte Anwendungen ehrlich hält. Ihr zukünftiges Ich, das ein \"warum ist das eine Stunde daneben\"-Ticket debuggt, das nie eingereicht wurde, wird es Ihnen danken.

> **Die Serie: Moderne Datumsverarbeitung in Delphi (DateUtils)**
> Zwei Teile: 1. Alltägliche Datums- und Zeitoperationen 2. Zeitzonen, UTC und ISO 8601 **— Sie sind hier**

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 —**