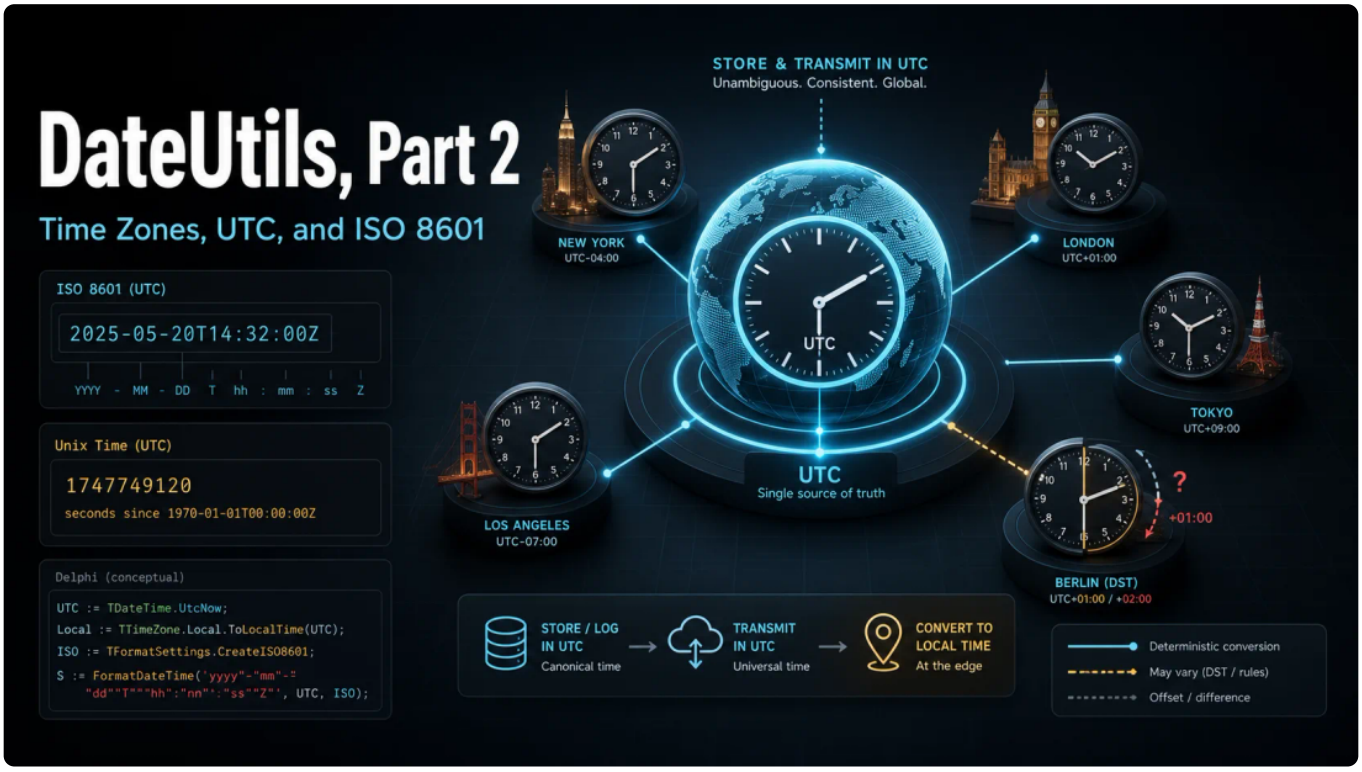
NEWS · JULY 30, 2026

DateUtils, Part 2: Time Zones, UTC, and ISO 8601

By Dr. Holger Flick



There's a category of bug that's almost a rite of passage. The app works perfectly on your machine. It works in testing. Then it ships, and a customer three time zones away reports that appointments are an hour off, or a report shows yesterday's data, or a \"created at\" timestamp is in the future. Everyone who has built software that crosses time zones has met this ghost.

In Part 1 we made everyday date math in `System.DateUtils` clean and readable — but we quietly assumed every date lived in the *same* time zone. The moment that stops being true — a server running in UTC, users spread across the world, a timestamp arriving from a REST API — you need a disciplined way to handle zones, or those ghosts move in.

The good news: the modern RTL has excellent tools for exactly this, and once you adopt one simple rule they make the whole problem tractable. **Part 2 is about doing time zones right**: the golden rule (\"store UTC, display local\"), the `TTimeZone` class for converting between them, and the interchange formats — **ISO 8601** and **Unix time** — that keep a timestamp unambiguous as it travels between systems. This is the part of date handling that actually matters for correctness. Let's exorcise the ghost.

The one rule that prevents most time-zone bugs

Before any API, the principle — because the tools only help if you use them within a sound model. Nearly every time-zone disaster traces back to storing or transmitting a \"naked\" local time with no record of *which* zone it was in. The fix is a discipline the whole industry has converged on:

> **Store and transmit UTC. Convert to the user's local time only at the edges — when you display it or accept input.**

UTC — Coordinated Universal Time — is the single, global, zone-less reference clock. It never springs forward or falls back. If every timestamp in your database and every timestamp on the wire is UTC, then comparisons, sorting, and \"days between\" all *just work*, because they're all measured against the same clock. Local time becomes a *presentation* concern, applied last.



Store UTC everywhere; convert to local only at the display and input edges

The diagram is the entire architecture: UTC lives in the middle, and conversion to local happens only at the two edges where a human is involved. Adopt this and the rest of this post is just \"how to do those two conversions correctly.\"

Getting the current time in UTC

Start at the source. The familiar `Now` returns *local* time — fine for display, wrong for storage. The RTL gives you UTC directly. The cleanest modern form is the `TDateTimeHelper` record helper the unit adds to `TDateTime`:

```
uses
  System.DateUtils;

var
  StampUtc: TDateTime;
begin
  StampUtc := TDateTime.NowUTC;   // current moment, in UTC
end;
```

`TDateTime.NowUTC` is your \"stamp it for storage\" call — use it wherever you'd have reached for `Now` to record when something happened. (There's also the classic function form via the OS, but `NowUTC` reads best and is right there on the type.)

Converting between local and UTC with `TTimeZone`

For converting *existing* values — a UTC timestamp from the database into local time for display, or user input the other way — the RTL provides the `TTimeZone` class. Its `Local` class property hands you the current user's zone, and two methods do the conversions:

```
var
  Utc, Local: TDateTime;
begin
  // UTC from the database ' local time for the UI
  Local := TTimeZone.Local.ToLocalTime(Utc);

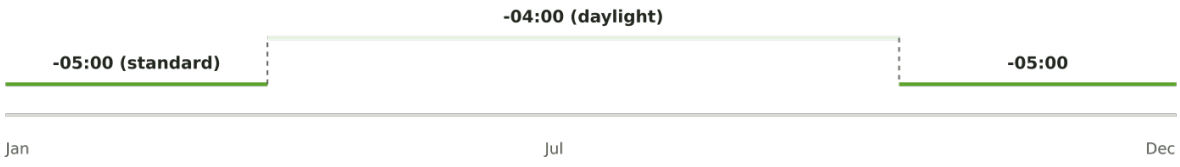
  // local time the user typed ' UTC to store
  Utc := TTimeZone.Local.ToUniversalTime(Local);
end;
```

That's the whole conversion story for most apps: `ToLocalTime` on the way out, `ToUniversalTime` on the way in. `TTimeZone` also exposes the offset itself, which is handy for display or logging:

```
ShowMessage(TTimeZone.Local.UtcOffset.ToString);      // e.g. '-04:00:00' as a TTimeSpan
ShowMessage(TTimeZone.Local.Abbreviation);            // e.g. 'GMT-4'
ShowMessage(TTimeZone.Local.GetUtcOffset(SomeDate).ToString); // offset *for a specific date*
```

Notice that `GetUtcOffset` takes a *date* — which matters more than it first appears, because the offset isn't constant.

> **Daylight Saving Time is why you must convert *for a specific date***
>
> A zone's offset from UTC **changes across the year** wherever Daylight Saving Time is observed — New York is `-05:00` in January and `-04:00` in July. This is the single biggest source of subtle time bugs, and it's why `TTimeZone` conversions take the *specific* `TDateTime` being converted: the offset is computed for *that* date's DST state, not a fixed number. It's also why you must never \"convert\" by just adding a hardcoded offset — `SomeUtc + (5/24)` is wrong half the year. Always go through `ToLocalTime`/`ToUniversalTime`, which consult the OS's zone database for the right offset on the right date. `TTimeZone` even lets you ask `GetLocalTimeType` to detect the rare invalid/ambiguous times that occur exactly at the DST transitions.



The UTC offset is not constant — DST shifts it during the year

The diagram is why hardcoded offsets fail: the same zone sits at `-05:00` for part of the year and `-04:00` for another. Only a date-aware conversion picks the right one — which is exactly what `TTimeZone` does for you.

Interchange: ISO 8601, the format for dates on the wire

When a date leaves your program — into JSON, a URL, a log, another system — a `TDateTime`'s raw `Double` is meaningless to the receiver. You need a *textual* format that's unambiguous, sortable, and universally understood.

That format is ISO 8601 — the `2026-08-11T14:30:00.000Z` you've seen in every modern API. `DateUtils` converts both directions:

```
var
  S: string;
  D: TDateTime;
begin
  // TDateTime ' ISO 8601 string (input treated as UTC by default ' trailing 'Z')
  S := DateToISO8601(TDateTime.NowUTC);
  // e.g. '2026-08-11T14:30:00.000Z'

  // ISO 8601 string ' TDateTime (returns UTC by default)
  D := ISO8601ToDate('2026-08-11T14:30:00.000Z');
end;
```

There's an even tidier form on the `TDateTime` helper — `SomeDate.ToISO8601` — mirroring the standalone function. Two things about these are worth pinning down, because they're where people trip.

> **The UTC parameters are the important detail**
> Both functions take a Boolean that declares the UTC-ness of the value: `DateToISO8601(D, AInputIsUTC)` says whether `D` is *already* UTC, and `ISO8601ToDate(S, AReturnUTC)` says whether you want the *result* in UTC. They default to `True`, which pairs perfectly with the golden rule — you're keeping everything in UTC anyway. If you follow \"store UTC, display local,\" you'll almost always leave these at their defaults and let the `Z` suffix travel with the string. For strings that lack a zone, the overload taking `TISO8601ToDateOptions` lets you say how to interpret them.

For validation of untrusted input, prefer the `Try` form so a malformed string doesn't raise:

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

Interchange: Unix time, the number that never lies

The other universal format is Unix time — a single integer counting seconds since 1 January 1970 UTC. It's compact, it's inherently UTC (no zone to get wrong), and it's what a huge amount of infrastructure speaks. `DateUtils` converts both ways:

```
var
  Secs: Int64;
  D: TDateTime;
begin
  Secs := DateTimeToUnix(TDateTime.NowUTC);   // e.g. 1786552200
  D := UnixToDateTime(Secs);                   // back to a TDateTime (UTC)
end;
```

Like the ISO functions, these carry `AInputIsUTC` / `AReturnUTC` parameters (defaulting to `True`), so they slot straight into the UTC-everywhere model. Reach for Unix time when you need compactness or you're talking to a system that expects it; reach for ISO 8601 when human-readability and self-describing zone info matter (logs, JSON APIs). Both are correct — it's a fit choice.

Putting it together: the round trip

Here's the whole discipline in one short flow — the shape almost every well-behaved app follows. Notice UTC in the middle and local only at the very ends.

```
// 1. Capture — stamp in UTC
Order.CreatedAt := TDateTime.NowUTC;

// 2. Transmit — serialize as ISO 8601 (stays UTC, gets a 'Z')
Json := DateToISO8601(Order.CreatedAt);

// 3. Receive elsewhere — parse back to UTC
var Received := ISO8601ToDate(Json);

// 4. Display — convert to the viewer's local zone, last
ShowMessage(DateTimeToStr(TTimeZone.Local.ToLocalTime(Received)));
```

Every step in the middle is zone-free and unambiguous; the only conversion is the final `ToLocalTime` for the human. Follow this and the \"appointments are an hour off\" ghost simply never appears.

The other side: where this needs more than the RTL

Guarantee to the reader — honest scoping. `TTimeZone.Local` gives you the *machine's* current zone, which is exactly right for a desktop app showing times to the person sitting at it. But two scenarios need more:

- **A server rendering times for many users in different zones.** The machine's local zone is irrelevant; you need each *user's* zone (stored in their profile), and you must convert per-user. The RTL's `TTimeZone.Local` alone won't do that — you need each user's IANA zone id and a library that can convert to an *arbitrary* named zone, not just the local one.
- **Historical/future accuracy across zone-rule changes.** Governments change DST rules; robust handling means an up-to-date IANA tz database.

> **Alternatives and companions**
> For arbitrary-named-zone conversion in Delphi, the well-regarded open-source TZDB library bundles the IANA database and plugs into the same `TTimeZone` abstraction — a great complement when `TTimeZone.Local` isn't enough. Across stacks, the same job is done by .NET's `TimeZoneInfo` and `DateTimeOffset`, JavaScript's Luxon or the new `Temporal` API, and Python's `zoneinfo`. The *principle* — UTC everywhere, convert at the edges, use a real tz database for named zones — is identical in every one of them. Learn it once here.

Takeaways

Across both parts, we moved from readable everyday date math to correct cross-zone handling. The part that protects you from real bugs:

- **The golden rule: store and transmit UTC; convert to local only at display and input.** UTC is the zone-less reference clock that makes comparisons and sorting trustworthy.
-

`TDateTime.NowUTC` stamps the current moment in UTC; `TTimeZone.Local.ToLocalTime` / `ToUniversalTime` convert at the edges — always *date-aware*, because DST makes the offset change through the year. Never add a hardcoded offset.

- **ISO 8601** (`DateToISO8601` / `ISO8601ToDate`, with `Try` for untrusted input) and **Unix time** (`DateTimeToUnix` / `UnixToDateTime`) are the unambiguous interchange formats; their `IsUTC` parameters default to the UTC-everywhere model.
- For per-user zones on a server or arbitrary named zones, pair the RTL with an IANA tz-database library.

> Store UTC, display local, convert at the edges with `TTimeZone` — and put ISO 8601 or Unix time on the wire. That discipline, plus the RTL's built-in tools, is what makes cross-time-zone software actually correct.

That closes the `DateUtils` series — Part 1 for everyday date math, this part for the zone-aware correctness that keeps distributed apps honest. Your future self, debugging a \"why is this an hour off\" ticket that never got filed, will thank you.

> **The series: Modern Dates in Delphi (DateUtils)**
> Two parts: 1. Everyday date and time operations 2. Time zones, UTC, and ISO 8601 **— you are here**

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 —