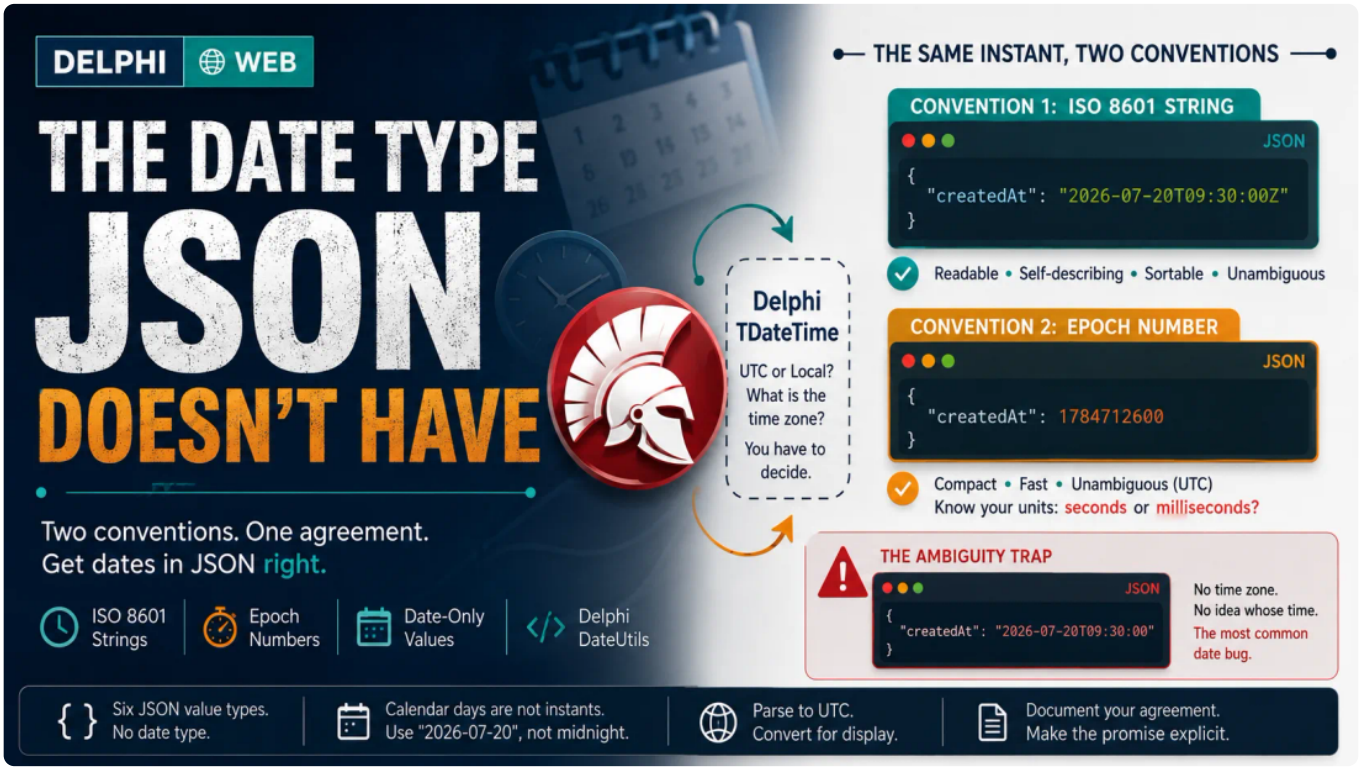
NEWS · JULY 20, 2026

# The Date Type JSON Doesn't Have

Part two of the JSON series for Delphi developers — why JSON has no date type, the two conventions everyone uses instead, what each one costs you, and how to read and write ISO 8601 correctly with System.DateUtils.

By Dr. Holger Flick




In Delphi, a date is a *thing*. You declare `Birthday: TDate`, the compiler knows what it is, the debugger shows it to you properly, and `IncMonth` does the right thing on the 31st. `TDate`, `TTime`, and `TDateTime` are real types in the language — you pass them around like integers, and the type system has your back the whole way.

Then you serialise one to JSON and all of that evaporates. Because JSON, as we saw at the end of part one, has exactly six value types — string, number, boolean, null, object, array — and not one of them is a date. There is no `"date"` keyword, no `2026-07-20` literal, nothing in the grammar that says "this value is a moment in time." The format simply has no opinion.

So a date in JSON isn't a type at all. It's a **convention**: a string or a number that both programs have agreed, out of band, to interpret as a point in time. Get that agreement right and dates are boring. Get it wrong — and this is where the bugs live — and your invoice is dated a day early for everyone east of you, or your timestamps jump to 2033 because someone read milliseconds as seconds.

This post is about that agreement: the two conventions the world actually uses, what each one costs, and how to produce and consume them correctly in Delphi with `System.DateUtils`.

## Why there is no date type (and why that was on purpose)

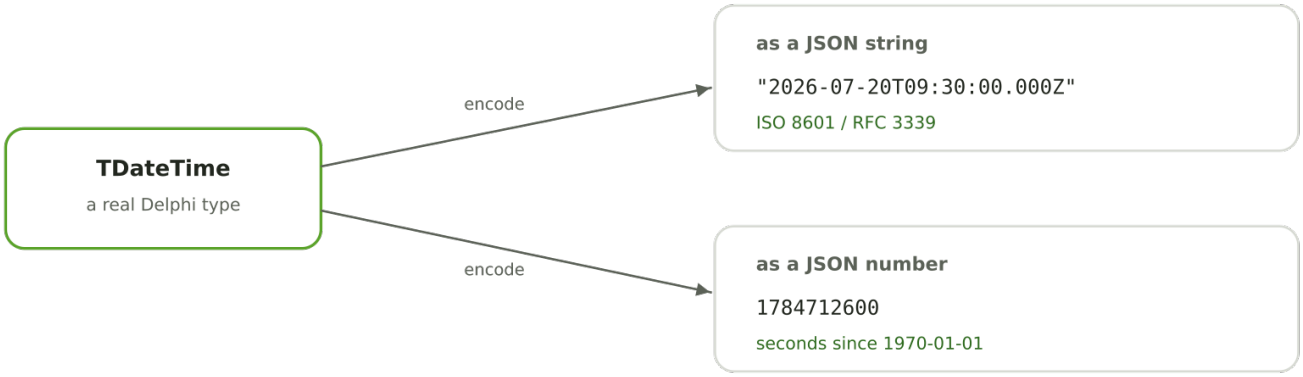
It's tempting to read the missing date type as an oversight, but it follows directly from what JSON was built to be.

JSON's grammar is deliberately tiny — small enough that a correct parser is an afternoon's work, which is exactly why every language got a good one so quickly. A date type would have broken that. Dates are genuinely hard: calendars, leap seconds, timezone databases that change several times a year, the question of whether "2026-07-20" is a moment or a label. Baking any of that into the wire format would have meant every parser in every language shipping a calendar implementation and agreeing on it forever.

So RFC 8259, the current JSON standard, simply doesn't mention dates. It defines six value types and stops.

Representing a timestamp is left to the application — which is a real cost, but a deliberate trade: JSON stayed small, and date handling became a decision you make rather than one you inherit.

That leaves you two materials to build with, since a date has to arrive as one of the six types. In practice, only two are ever used.




One TDateTime, two conventions — and JSON itself endorses neither

Both boxes on the right hold the same instant. Neither is more "correct" as far as JSON is concerned — the format sees a string and a number, nothing more. The difference is entirely in what the two ends agreed the bytes mean, and that agreement is yours to specify.

## Convention 1: the ISO 8601 string

This is the one you should reach for by default, and the one the overwhelming majority of modern APIs use.

ISO 8601 is the international standard for writing dates and times as text. The shape you'll see everywhere is

`2026-07-20T09:30:00Z`: year-month-day, a `T` separator, hour:minute:second, and a timezone designator. The trailing `Z` — spoken "Zulu" — means UTC.

In practice, what most APIs actually implement is RFC 3339, which the spec describes as "a profile of the ISO 8601 standard" and "a conformant subset of the ISO 8601 extended format." That distinction matters more than it sounds. Full ISO 8601 is a sprawling standard that also permits week dates (`2026-W30-1`), ordinal dates (`2026-201`), and omitting separators. RFC 3339 throws nearly all of that away and pins down one unambiguous shape — it explicitly "requires the 'T' to avoid ambiguity." When someone says "we use ISO 8601," they almost always mean the RFC 3339 subset.

The offset is the part worth being careful about. RFC 3339 also draws a semantic line most developers miss: `Z` and `+00:00` mean "UTC is the preferred reference point," whereas `-00:00` specifically means "the time in UTC is known, but the offset to local time is unknown."

> **The one that actually bites: no offset at all**
>
> A string like `"2026-07-20T09:30:00"` — no `Z`, no `+02:00` — is the single most common source of date bugs in JSON APIs. It's a *naive* timestamp: it names a wall-clock reading without saying whose wall. The sender usually means their own local time; the receiver usually assumes UTC, or its own local time, and different libraries pick differently. Two systems both behave "reasonably" and the value silently shifts by hours. If you control the format, always emit an offset.

**What it costs.** Strings are bigger than numbers — roughly 24 bytes versus 10 — and parsing text is slower than reading an integer. For the vast majority of applications that difference is irrelevant next to HTTP overhead.

**What you get.** A great deal, and it's why this convention won:

- **It's readable.** You can eyeball a response and see the date. When a support ticket includes a raw payload, that matters.
- **It's unambiguous** when the offset is present — the value carries its own timezone context.
- **It sorts correctly as text.** Because the fields run largest to smallest with fixed widths, lexicographic string order *is* chronological order for UTC timestamps. Databases, log tools, and plain `ORDER BY` all get this for free.
- **It expresses more than an instant.** `"2026-07-20"` on its own is a legitimate date-only value — a concept the epoch-number convention cannot express at all, which is the subject of its own section below.

## Convention 2: the epoch number

The alternative is to send a plain JSON number: a count of time units elapsed since a fixed reference point, almost always the Unix epoch of 1970-01-01T00:00:00Z.

This is compact, trivially comparable, and inherently unambiguous about timezone — an epoch count is always measured from a UTC instant, so there's no offset to forget. It's a natural fit for machine-to-machine traffic, high-frequency telemetry, and JWT token claims, where nobody reads the payload by hand.

It has two sharp edges, and both are common in production.

**Seconds or milliseconds?** This is the big one, because JSON gives you no way to tell. Unix time is classically counted in seconds, so `1784712600` is a moment in July 2026. But JavaScript works in milliseconds — `Date.now()` returns `1784712600000`. Both are "a Unix timestamp." Feed a milliseconds value to something expecting seconds and you land tens of thousands of years in the future; feed seconds to something expecting milliseconds and you get January 1970. The number carries no clue about which it is — you have to know, and the only place that knowledge lives is your documentation.

**Precision.** JSON numbers are a single type with no integer/float distinction, and RFC 8259 is direct about the consequence: "numbers that are integers and are in the range [-(2\\53)+1, (2\\53)-1] are interoperable," because implementations widely use IEEE 754 double precision. Second-level timestamps are nowhere near that limit. Millisecond and microsecond timestamps have more headroom than people fear, but once a language parses every JSON number into a double — as JavaScript does — large integer values are exactly where silent precision loss starts.

| | | |
|---|---|---|
| **ISO 8601 string** | `"2026-07-20T09:30:00Z"` | self-describing |
| **Unix seconds** | `1784712600` | unit not stated |
| **Unix milliseconds** | `1784712600000` | unit not stated |
| **naive string** | `"2026-07-20T09:30:00"` | whose clock? |

Same instant, four encodings — and only the string says what it means

The first three rows are the same instant written three ways. Look at the right-hand column: only the top row tells a reader — human or machine — what it actually means. The two numeric rows are indistinguishable without documentation, and the bottom row is the ambiguity trap from earlier, which looks the most human-friendly and is the most dangerous.

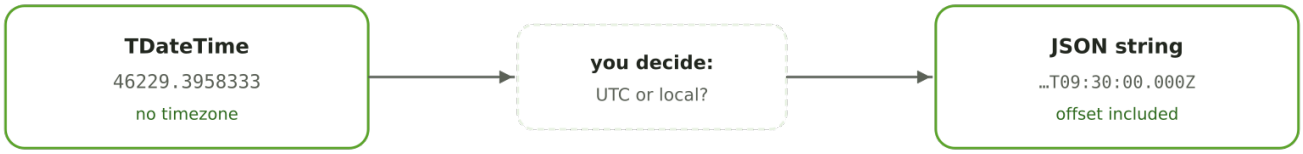
> **So which should you use?**
> My recommendation, and it's the one most of the industry has converged on: **default to ISO 8601 strings with an explicit offset**. The readability and self-description pay for themselves the first time you debug a payload, and the size difference rarely matters over HTTP. Reach for epoch numbers when you have a specific reason — very high message volume, embedded or bandwidth-constrained links, or a spec that mandates them, as JWT does for `exp` and `iat`. And if you consume an API, this isn't your choice at all: read their documentation and match it exactly.

## How Delphi represents dates, and why the mismatch is real

Before the code, it's worth being precise about what you're converting *from*, because Delphi's model is genuinely different from JSON's.

`TDateTime` is declared in `System.pas` as `TDateTime = type Double` — a floating-point number. The integral part counts days since the epoch day 12/30/1899, and the fractional part is the fraction of a day elapsed: `0.5` is noon, `0.75` is 6pm. `TDate` and `TTime` are declared as distinct types over the same Double (`TDate = type TDateTime`), which gives you compile-time intent and better debugger and RTTI behaviour — but at runtime they're all the same number.

That has a consequence worth internalising: **a `TDateTime` carries no timezone.** It's a bare number on a timeline whose meaning — is this local time or UTC? — lives entirely in your head and your naming conventions. JSON's ISO 8601 convention, by contrast, *can* carry an offset. So the conversion isn't just formatting; it's the moment you have to supply information the value never held.




The conversion adds meaning the raw Double never carried

The dashed box in the middle is the whole point. Every function in the next section is really asking you that question, usually through a boolean parameter that's easy to leave at its default without thinking. That's where wrong-by-hours bugs come from.

## Writing and reading ISO 8601 with System.DateUtils

Delphi ships everything you need in `System.DateUtils`. The two functions that matter are `DateToISO8601` and `ISO8601ToDate`, with a `Try…` variant for parsing untrusted input.

Their declarations tell you most of what you need:

```
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;
```

Note that both default to `True` — that is, both assume you are working in **UTC**. That default is a sensible one, but it is only correct if the value you hand over really is UTC.

### Writing a timestamp

The common case is a value you already hold in UTC, which you write straight out:

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

var
  LNow: TDateTime;
  LObj: TJSONObject;
begin
  LNow := TTimeZone.Local.ToUniversalTime(Now);   // Now is local — convert first

  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;
```

The important line is the conversion. `Now` returns **local** time, so passing it directly to `DateToISO8601` with the default `AInputIsUTC = True` would stamp a `Z` onto a local reading and claim it was UTC — the naive-timestamp

bug, committed at the source. `TTimeZone.Local.ToUniversalTime` converts properly, honouring daylight saving for that date.

Alternatively, hand the local value over and tell the truth about it:

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

With `AInputIsUTC = False`, the RTL looks up the local UTC offset for that date and appends it instead of `Z`. Both strings above denote the same instant; both are valid RFC 3339. Pick one and be consistent.

## Reading a timestamp

For input you control, `ISO8601ToDate` is fine; it raises an exception on malformed text. For anything arriving over the network, prefer `TryISO8601ToDate`:

```
var
  LValue: TJSONValue;
  LWhen: TDateTime;
begin
  LValue := LObj.GetValue('created_at');
  if (LValue <> nil) and TryISO8601ToDate(LValue.Value, LWhen) then
    // LWhen is UTC (AReturnUTC defaults to True)
    Writeln(DateTimeToStr(TTimeZone.Local.ToLocalTime(LWhen)))
  else
    Writeln('missing or invalid timestamp');
end;
```

Two things are worth calling out. `LValue.Value` gives you the unescaped string content. And the parse result is UTC by default — converting to local for display is a separate, deliberate step, which is exactly the discipline that prevents timezone drift: **parse to UTC, store and compute in UTC, convert to local only when showing it to a human.**

For finer control there's an overload taking a set:

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

`ioNoTZIsLocal` handles the naive-string case explicitly — when the incoming text has no timezone, treat it as local rather than UTC. `ioReturnUTC` controls whether the result comes back as UTC. If you must consume an API that emits naive timestamps, this is the knob that lets you state your assumption in code rather than leaving it implicit.

## Date-only values are a different problem

Here's a case that deserves its own treatment, because using the wrong tool causes a real and surprisingly common bug.

Some values are not instants. A birthday, an invoice date, a public holiday, a contract start — these are **calendar days**, not moments on a timeline. Nobody was born at `1990-03-14T00:00:00Z`; they were born on the 14th of March, and that fact is true regardless of where you're standing.

If you send such a value as a timestamp, you attach a time and a timezone that were never part of it. Midnight is the worst possible choice, because it sits directly on the boundary: `"1990-03-14T00:00:00Z"` converted to local time in Auckland is the 14th at noon — fine — but the same midnight-UTC convention applied to a value someone entered in Berlin can land on the 13th once it's converted back. The date shifts by a day depending on who's looking. This is the classic off-by-one-day bug, and it's entirely self-inflicted.

The correct wire format is the date-only form, `"2026-07-20"` — which RFC 3339 defines as `full-date` and which carries no time and no offset, precisely because there isn't one.

And this is where the RTL leaves you to it. `DateToISO8601` has no date-only mode: it always emits the full `yyyy-mm-ddThh:nn:ss.zzzZ` shape. Its format string is fixed in the implementation as `'%.4d-%.2d-%.2dT%.2d:%.2d:%.2d.%.3dZ'`, so even a pure `TDate` — whose fractional part is zero — comes out as a midnight timestamp with a `Z` and three zero milliseconds. Hand it a `TDate` and you get back exactly the thing you shouldn't send.

For date-only values, use `FormatDateTime` instead:

```
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 is locale-sensitive — pin it**
>
> `FormatDateTime` uses the application's format settings, and `'yyyy-mm-dd'` will still be honoured, but the safe habit for wire formats is to pass invariant settings explicitly so no user locale can ever influence the output: ``pascal var  LFmt: TFormatSettings; begin  LFmt := TFormatSettings.Invariant;- Writeln(FormatDateTime('yyyy-mm-dd', LBirthday, LFmt)); end; `  Use the same discipline for a time-only value:` FormatDateTime('hh:nn:ss', LTime, LFmt)`.

Reading a date-only string back is straightforward, since the shape is fixed:

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

That looks blunt, and it is — but for a fixed 10-character format it's honest and dependency-free. (`TryISO8601To-Date` will also accept a bare `"1990-03-14"` and hand you back a `TDateTime` at midnight; if you go that route, be deliberate about the UTC flags so midnight doesn't drift.)

## Epoch numbers, when you need them

If the API you're talking to uses epoch numbers, `System.DateUtils` covers that too:

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

These work in **seconds**, which is the classic Unix convention. If your counterpart speaks JavaScript, it is very likely sending milliseconds, and you need to bridge that yourself:

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

// reading milliseconds
LWhen := UnixToDateTime(LMillis div 1000);
```

Dividing away the milliseconds discards sub-second precision, so if you genuinely need it, use `IncMilliSecond(UnixDateDelta, LMillis)` instead — `UnixDateDelta` is the RTL's constant for the 1970 epoch expressed as a `TDateTime`.

## Where the RTL falls short — and the libraries that fill the gap

The built-in functions are solid, correct, and free of dependencies, and for hand-built JSON they're genuinely all you need. But it's worth being straight about their limits, because you'll meet all three.

**It always writes milliseconds.** `DateToISO8601` emits `.000` unconditionally. That's valid RFC 3339 and matches what JavaScript's `JSON.stringify` produces — `Date.prototype.toJSON` yields exactly `1975-08-19T23:15:30.000Z` — so the RTL is in good company. But some APIs and some strict validators expect `2026-07-20T09:30:00Z` with no fractional part, and the RTL gives you no switch for it. You're formatting the string yourself, or trimming afterwards.

**It has no date-only mode.** As shown above, `TDate` gets a full timestamp whether you want one or not. The type distinction that Delphi maintains so carefully in the language is dropped at the JSON boundary.

**It's value-at-a-time.** These functions convert one `TDateTime`. Serialising an object graph means writing the traversal, deciding per field which convention applies, and doing it again on the way back.

That last point is where third-party libraries earn their keep. Delphi Neon by Paolo Rossi (Apache-2.0) is an RTTI-based serialiser that maps whole objects, records, and generic collections to and from JSON. Its date handling is instructive precisely on the gap above: it keeps *separate* paths for `TDate` and `TDateTime`, with `TJSONUtils.DateToJSON` formatting a `TDate` as `YYYY-MM-DD` while datetimes go through `DateToISO8601`. In other words, it preserves the distinction the RTL flattens — and it exposes a `UseUTCDate` configuration option so the UTC decision is made once, in configuration, instead of at every call site.

> **The wider landscape**
> Neon isn't the only option, and none of this is Delphi-specific. mORMot 2 includes a very fast JSON engine with its own date conventions, and JsonDataObjects is a compact, high-performance single-unit parser — both open source. Outside Delphi the same problem gets solved the same way: Jackson in Java and System.Text.Json in .NET both default to ISO 8601 strings, Python's `datetime.isoformat()` produces the same shape, and Go's `encoding/json` marshals `time.Time` as RFC 3339. That convergence is the real reassurance: choose ISO 8601 with an explicit offset and you're speaking the same dialect as everyone else on the wire.

## Takeaways

JSON's missing date type isn't a gap to be patched — it's a decision that got pushed to you, and the whole game is making that decision explicitly rather than by accident.

- **A date in JSON is a convention, not a type.** Two programs agree that a particular string or number means a moment. Write that agreement down; it's the only place the meaning lives.
- **Default to ISO 8601 with an explicit offset.** Readable, sortable, self-describing, and universally understood. Use epoch numbers when volume or a spec demands it — and document seconds versus milliseconds, because the number can't tell anyone which it is.
- **Never emit a naive timestamp.** No `Z`, no offset, no idea. It's the most common date bug in JSON APIs and the easiest to avoid.
- **Calendar days are not instants.** Send a birthday as `"1990-03-14"`. The RTL won't do this for you — `DateToISO8601` always writes a full timestamp — so use `FormatDateTime('yyyy-mm-dd', …)` with invariant settings.
- **Parse to UTC, compute in UTC, convert to local only for display.** The `AInputIsUTC` and `AReturnUTC` parameters default to `True`; make sure that default is telling the truth about your data.

> Delphi gives you a type and the compiler enforces it. JSON gives you a string and a promise. The bugs live in the gap, and the fix is to state the promise out loud — in the format, and in your docs.

Next time we finally write real code against the classes themselves: `System.JSON`, and the small family of types — `TJSONObject`, `TJSONArray`, `TJSONString`, `TJSONNumber`, `TJSONBool`, `TJSONNull` — that mirror exactly the six value types from part one. Building, parsing, and walking JSON with nothing but what's already in the box.

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 —**