NEWS · AUGUST 13, 2026

Zwei Länder, zwanzig Feiertage, eine Datenstruktur

Teil eins einer zweiteiligen Serie über die Berechnung gesetzlicher Feiertage in Delphi — warum ein Feiertagskalender in eine Datenstruktur gehört und nicht in eine case-Anweisung, drei kleine Datums-Primitive auf Basis von System.DateUtils und der komplette US-Bundesfeiertagskalender in einem lesbaren Array.

By Dr. Holger Flick



Jede Geschäftsanwendung muss irgendwann wissen, ob ein bestimmter Tag ein Arbeitstag ist. Die Lohnbuchhaltung braucht es. Lieferterminschätzungen brauchen es. Alles, was ein SLA in Werktagen berechnet, braucht es. Und die erste Version dieses Codes ist in nahezu jeder Codebasis, die ich gesehen habe, eine `case`-Anweisung, die jeden Dezember einen neuen Zweig bekommt.

Das Problem sind nicht die Zweige. Das Problem ist, dass ein Feiertag kein Datum ist — er ist eine *Regel*.

„Weihnachten" ist ein festes Datum und wird es immer bleiben. Aber „Thanksgiving" ist der vierte Donnerstag im November, „Memorial Day" der *letzte* Montag im Mai, und „Ostermontag" ist der Tag nach einer lunisolaren

Berechnung, die dem Computer um rund fünfzehn Jahrhunderte vorausgeht. Kodieren Sie die Daten fest, sind Sie nächstes Jahr wieder dran. Kodieren Sie die Regeln, sind Sie für immer fertig.

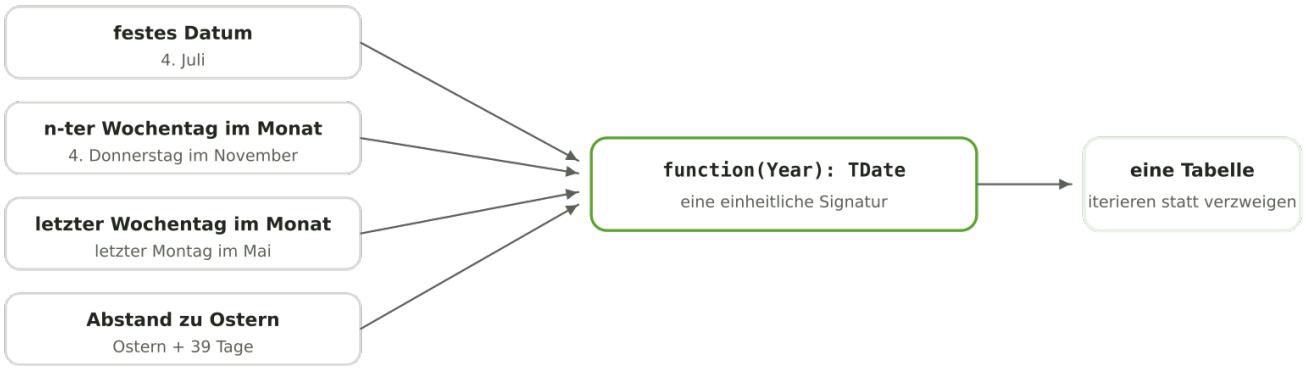
Dies ist eine zweiteilige Serie darüber, genau das in Delphi zu tun, vollständig aufgebaut auf `System.DateUtils` — der RTL-Unit, die ich in Teil 1 der DateUtils-Serie vorgestellt habe. **Dieser Teil** baut die Datenstruktur

und die drei Datums-Primitive, aus denen jeder regelbasierte Feiertag besteht, und drückt damit alle elf US-Bundesfeiertage in einem einzigen lesbaren Array aus. Teil 2 nimmt sich die interessante Hälfte vor —

Ostern, die vier deutschen Feiertage, die daran hängen, und das Zusammenführen zweier nationaler Kalender. Am Ende haben Sie eine Feiertags-Engine von etwa hundert Zeilen, die nie aktualisiert werden muss.

Ein Feiertag ist eine Funktion von einem Jahr auf ein Datum

Die Einsicht, die das alles zusammenfallen lässt, ist klein: Was ein Feiertag kulturell auch sein mag — rechnerisch ist er eine Funktion, die ein Jahr entgegennimmt und ein Datum zurückgibt. `2026 ' 2026-11-26`. Mehr nicht. Sobald jeder Feiertag dieselbe Form hat, passen alle in eine Liste, und der Code, der die Liste *benutzt*, muss nie wissen, welche Art von Regel er gerade vor sich hat.



Jeder Feiertag, wie auch immer definiert, reduziert sich auf dieselbe Signatur: Jahr rein, Datum raus

Lesen Sie das Diagramm von links nach rechts, und Sie haben die gesamte Architektur. Vier strukturell verschiedene Arten von Regeln — und *weil* sie eine Signatur teilen, hat der aufrufende Code genau einen

Pfad hindurch. Es gibt im Folgenden nirgends ein `if Kind = hkFixed then … else if Kind = hkNthWeekday then …`. Genau dieser Zweig ist das, was wir wegkonstruieren.

Delphi drückt „eine Funktion, die man in einer Variablen ablegen kann" mit einem anonymen Methodentyp aus,

deklariert mit `reference to`:

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

type
  /// Berechnet das Datum, auf das dieser Feiertag in einem Jahr fällt.
  TComputeHoliday = reference to function(const AYear: Word): TDate;

  THolidayRule = record
    UID: string;      // stabiler Bezeichner – sicher in einer Datenbank speicherbar
    Name: string;     // Anzeigename
    Compute: TComputeHoliday;
    constructor Create(const AUID, AName: string; const ACompute: TComputeHoliday);
  end;

constructor THolidayRule.Create(const AUID, AName: string;
  const ACompute: TComputeHoliday);
begin
  UID := AUID;
  Name := AName;
  Compute := ACompute;
end;
```

Die `UID` verdient ein Wort, denn sie ist das Feld, das man weglässt und dann bereut. Anzeigenamen ändern sich — sie werden übersetzt, anders geschrieben, und in einem denkwürdigen Fall wurde der Name eines

US-Feiertags per Gesetz geändert. Ein stabiler maschineller Bezeichner sorgt dafür, dass ein Datensatz mit „dieser Mitarbeiter hatte an `memorial-day` frei" auch danach noch etwas bedeutet.

> **Warum ein Record und keine Klasse**
> Ein `record` mit einem Feld vom Typ einer anonymen Methode wird vom Compiler verwaltet — keine `Create`/`Free`-Paare, keine Eigentümerfrage, und das Array räumt sich selbst auf. Die Closure, die jedes `Compute` einfängt, ist wie ein Interface referenzgezählt. Eine Klasse würde genauso funktionieren; sie würde Ihnen lediglich eine Speicherverwaltungsaufgabe aufbürden, ohne dafür etwas zu bieten.

Die drei Primitive — und warum jedes ein Einzeiler ist

Die elf US-Bundesfeiertage sind in 5 U.S.C. § 6103 definiert, und das Gesetz genau zu lesen lohnt sich hier wirklich: Es verwendet exakt drei Formulierungen. Manche Feiertage sind ein Kalenderdatum („1. Januar"). Manche sind ein Ordinal-Wochentag („der dritte Montag im Januar"). Einer ist ein terminaler Wochentag („der letzte Montag im Mai"). Das ist das gesamte Vokabular — also genau drei Hilfsfunktionen.

Die erste muss man kaum schreiben, aber sie zu benennen zahlt sich darin aus, wie die Regeltabelle später zu lesen ist:

```
function Fixed(const AYear, AMonth, ADay: Word): TDate;
begin
  Result := EncodeDate(AYear, AMonth, ADay);
end;
```

Die zweite ist die interessante. Um den *n*-ten Montag eines Monats zu finden, startet man am 1., fragt, welcher Wochentag das ist, geht vorwärts bis zum ersten Montag und addiert dann ganze Wochen. `DayOfTheWeek` aus `System.DateUtils` ist das richtige Werkzeug, weil es nach ISO 8601 nummeriert ist — Montag ist 1 bis Sonntag ist 7 — was die Arithmetik ehrlich hält:

```
/// Das ANth-te Auftreten von ADayOfWeek in einem Monat.
/// ADayOfWeek nutzt die DateUtils-Konstanten: DayMonday = 1 .. DaySunday = 7.
function NthWeekdayOfMonth(const AYear, AMonth, ADayOfWeek, ANth: Word): TDate;
var
  FirstOfMonth: TDate;
  Offset: Integer;
begin
  FirstOfMonth := EncodeDate(AYear, AMonth, 1);
  // Wie viele Tage vom 1. vorwärts bis zum ersten ADayOfWeek?
  Offset := (Integer(ADayOfWeek) - DayOfTheWeek(FirstOfMonth) + 7) mod 7;
  Result := IncDay(FirstOfMonth, Offset + 7 * (Integer(ANth) - 1));
end;
```

Beim `+ 7) mod 7` lohnt sich ein kurzer Halt: Es verwandelt eine möglicherweise negative Differenz in einen Vorwärtsschritt von 0 bis 6 Tagen, sodass die Funktion funktioniert, egal ob der 1. vor oder nach dem Zielwochentag liegt. Lassen Sie es weg, und der Kalender stimmt etwa in der Hälfte der Fälle — genau die Art Fehler, die im Januar durch alle Tests kommt und im September auffliegt.

Das dritte Primitiv arbeitet vom Monatsende rückwärts, und hier verdient sich `EndOfTheMonth` aus Teil 1 seinen Platz, weil es die Länge jedes Monats kennt, Schaltjahre eingeschlossen:

```
/// Das letzte Auftreten von ADayOfWeek in einem Monat.
function LastWeekdayOfMonth(const AYear, AMonth, ADayOfWeek: Word): TDate;
var
  LastOfMonth: TDate;
  Offset: Integer;
begin
  LastOfMonth := DateOf(EndOfTheMonth(EncodeDate(AYear, AMonth, 1)));
  // Wie viele Tage vom letzten Tag rückwärts bis zum letzten ADayOfWeek?
  Offset := (DayOfTheWeek(LastOfMonth) - Integer(ADayOfWeek) + 7) mod 7;
  Result := IncDay(LastOfMonth, -Offset);
end;
```

Beachten Sie das `DateOf` um `EndOfTheMonth`: `EndOfTheMonth` liefert die letzte *Millisekunde* des Monats,

`23:59:59.999`, wir wollen aber ein sauberes Mitternachtsdatum. Ein Wort, das verhindert, dass ein `TDate` einen Zeitanteil in jeden späteren Vergleich hineinträgt.

!!! warning "„Letzter" ist nicht „fünfter" — und auch nicht „vierter"" Das ist der häufigste Fehler in handgeschriebenem Feiertagscode. Memorial Day ist der *letzte* Montag im Mai — und der Mai hat in manchen

Jahren fünf Montage und in anderen vier, also ist weder `NthWeekdayOfMonth(…, 4)` noch `NthWeekdayOf-Month(…, 5)` allgemein korrekt. 2026 ist der letzte Montag im Mai der 25.; 2027 ist es der 31., was tatsächlich der fünfte *ist*. Fragen Sie nach dem fünften Montag eines Mais, der nur vier hat, liefert `NthWeekdayOfMonth`

bereitwillig ein Datum im Juni. Deshalb zählt `LastWeekdayOfMonth` vom Monatsende rückwärts statt vom Anfang vorwärts — die Frage „welcher ist der letzte" hat jedes Jahr eine Antwort, die Frage „gibt es einen fünften" nicht.

> **Die RTL hat davon auch eine Variante**
>
> `System.DateUtils` liefert `EncodeDayOfWeekInMonth` mit (und ein `TryEncodeDayOfWeekInMonth`, das `False` zurückgibt statt `EConvertError` auszulösen), das die n-ter-Wochentag-Aufgabe direkt erledigt. Das ist eine völlig gute Wahl und eine Funktion weniger, die Ihnen gehört. Ich habe die Arithmetik hier ausgeschrieben, weil sie das Randverhalten sichtbar macht — und weil der Fall „letzter Wochentag" ohnehin die rückwärts zählende Variante braucht. Nutzen Sie für den n-ten Fall gern die RTL-Funktion; der Regeltabelle unten ist es egal, welche Implementierung hinter dem Namen steht.

Die US-Tabelle: elf Feiertage, elf Zeilen

Mit den Primitiven wird der gesamte US-Bundesfeiertagskalender zu einer Liste von Regeln, die sich fast genau wie das Gesetz liest, das sie umsetzt. Das ist der Lohn für die Vorarbeit — und beachten Sie, dass hier überhaupt keine Logik mehr steht, nur Daten:

```
function USRules: TArray<THolidayRule>;
begin
  Result := [
    THolidayRule.Create('new-years-day', 'New Year''s Day',
      function(const Y: Word): TDate begin Result := Fixed(Y, 1, 1); end),
    THolidayRule.Create('mlk-day', 'Martin Luther King Jr. Day',
      function(const Y: Word): TDate begin Result := NthWeekdayOfMonth(Y, 1, DayMonday, 3); end),
    THolidayRule.Create('presidents-day', 'Washington''s Birthday',
      function(const Y: Word): TDate begin Result := NthWeekdayOfMonth(Y, 2, DayMonday, 3); end),
    THolidayRule.Create('memorial-day', 'Memorial Day',
      function(const Y: Word): TDate begin Result := LastWeekdayOfMonth(Y, 5, DayMonday); end),
    THolidayRule.Create('juneteenth', 'Juneteenth National Independence Day',
      function(const Y: Word): TDate begin Result := Fixed(Y, 6, 19); end),
    THolidayRule.Create('independence-day', 'Independence Day',
      function(const Y: Word): TDate begin Result := Fixed(Y, 7, 4); end),
    THolidayRule.Create('labor-day', 'Labor Day',
      function(const Y: Word): TDate begin Result := NthWeekdayOfMonth(Y, 9, DayMonday, 1); end),
    THolidayRule.Create('columbus-day', 'Columbus Day',
      function(const Y: Word): TDate begin Result := NthWeekdayOfMonth(Y, 10, DayMonday, 2); end),
    THolidayRule.Create('veterans-day', 'Veterans Day',
      function(const Y: Word): TDate begin Result := Fixed(Y, 11, 11); end),
    THolidayRule.Create('thanksgiving', 'Thanksgiving Day',
      function(const Y: Word): TDate begin Result := NthWeekdayOfMonth(Y, 11, DayThursday, 4);
end),
    THolidayRule.Create('christmas-day', 'Christmas Day',
      function(const Y: Word): TDate begin Result := Fixed(Y, 12, 25); end)
  ];
end;
```

Zwei Anmerkungen zu den Namen, denn Präzision zählt, wenn die Quelle ein Gesetz ist. Der Feiertag, den die meisten Amerikaner „Presidents' Day" nennen, heißt im Bundesrecht **Washington's Birthday** — das Gesetz wurde nie entsprechend geändert, auch wenn viele Bundesstaaten den anderen Namen verwenden. Und Juneteenth heißt vollständig **Juneteenth National Independence Day**, 2021 durch den Juneteenth National Independence Day Act in die Liste aufgenommen — weshalb jede vor 2021 geschriebene Feiertagstabelle eine Zeile zu wenig hat. Ich habe die juristischen Namen in `Name` und die geläufigen in der `UID` belassen; wenn Sie das Ergebnis Anwendern anzeigen, ist eine Anzeigenamen-Zuordnung der richtige Ort, um dem Gesetzgeber zu widersprechen.

> **Was diese Tabelle bewusst nicht leistet**
>
> Dies sind die **Bundes**feiertage — die Tage, an denen Bundesbedienstete frei haben. Die Vereinigten Staaten haben keine nationalen Feiertage, die private Arbeitgeber binden, und die Bundesstaaten ergänzen eigene (Patriots' Day in Massachusetts, Cesar Chavez Day in Kalifornien und so weiter). Diese Tabelle ist der gemeinsame Nenner, dem jeder US-Kalender zustimmt — damit die richtige Voreinstellung und die falsche endgültige Antwort, wenn Sie Lohnabrechnung für einen bestimmten Bundesstaat machen. Dasselbe gilt für Deutschland, und darauf komme ich in Teil 2 zurück.

Die Tabelle auswerten

Weil jede Regel dieselbe Signatur hat, ist die Berechnung eines Jahreskalenders eine Schleife, in der von Regeltypen keine Rede mehr ist:

```
type
  THoliday = record
    UID: string;
    Name: string;
    Date: TDate;
  end;

function ComputeYear(const ARules: TArray<THolidayRule>;
  const AYear: Word): TArray<THoliday>;
var
  I: Integer;
begin
  SetLength(Result, Length(ARules));
  for I := 0 to High(ARules) do
  begin
    Result[I].UID  := ARules[I].UID;
    Result[I].Name := ARules[I].Name;
    Result[I].Date := ARules[I].Compute(AYear);   // die einzige Zeile, die variiert
  end;
end;
```

Dieser eine Aufruf `ARules[I].Compute(AYear)` ist der Ort, an dem die `case`-Anweisung geblieben ist. Einen zwölften Feiertag hinzuzufügen heißt, eine zwölfte Zeile in die Tabelle zu schreiben — diese Funktion ändert sich nie, und alles danach ebenso wenig.

Den Kalender 2026 auszugeben ist dann Formsache:

```
var
  H: THoliday;
begin
  for H in ComputeYear(USRules, 2026) do
    Writeln(Format('%-45s %s', [H.Name, FormatDateTime('ddd, dd mmm yyyy', H.Date)]));
end;
```

was die elf Tage unten liefert. Ein Stichprobenabgleich mit einem Kalender lohnt sich, denn nur geprüfte Ausgabe ist veröffentlichenswert:

```
New Year's Day                              Thu, 01 Jan 2026
Martin Luther King Jr. Day                  Mon, 19 Jan 2026
Washington's Birthday                       Mon, 16 Feb 2026
Memorial Day                                Mon, 25 May 2026
Juneteenth National Independence Day        Fri, 19 Jun 2026
Independence Day                            Sat, 04 Jul 2026
Labor Day                                   Mon, 07 Sep 2026
Columbus Day                                Mon, 12 Oct 2026
Veterans Day                                Wed, 11 Nov 2026
Thanksgiving Day                            Thu, 26 Nov 2026
Christmas Day                               Fri, 25 Dec 2026
```

Der Independence Day 2026 fällt auf einen Samstag — eine gute Erinnerung daran, dass „ist ein Feiertag" und „ist ein freier Tag" verschiedene Fragen sind: Die Bundespraxis verlegt einen Samstagsfeiertag auf den vorangehenden Freitag, gemäß § 6103(b). Das ist eine *Beobachtungsregel* über der *Auftretensregel*, und beide getrennt zu halten ist es, was Ihnen erlaubt, beide Fragen aus einer Tabelle zu beantworten. Das verdrahte ich in Teil 2, denn Deutschland macht es bewusst nicht so.

Lohnt sich eine Nachschlagefunktion?

Wenn das Jahr einmal berechnet ist, lauten die Fragen, die Sie tatsächlich stellen: „Ist dieses Datum ein Feiertag?" und „Welcher?" Ein lineares Durchsuchen von elf Einträgen ist völlig in Ordnung, aber wenn Sie

das in einer Werktagsschleife über einen Datumsbereich aufrufen, ist ein nach Datum indiziertes Dictionary angenehmer und macht aus der Antwort einen Hash-Zugriff:

```
function HolidayIndex(const ARules: TArray<THolidayRule>;
  const AYear: Word): TDictionary<TDate, string>;
var
  H: THoliday;
begin
  Result := TDictionary<TDate, string>.Create;
  for H in ComputeYear(ARules, AYear) do
    Result.AddOrSetValue(H.Date, H.Name);   // AddOrSetValue: zwei Feiertage können auf ein Datum
fallen
end;
```

`AddOrSetValue` statt `Add` ist keine vorsorgliche Polsterung — es ist tragend. Dass zwei Feiertage auf dasselbe Datum fallen, ist ein realer Fall, und er tritt in dem Moment auf, in dem Sie die Kalender zweier Länder zusammenführen. Diese Zusammenführung ist die Aufgabe von Teil 2, und dort beginnt sich diese Datenstruktur wirklich auszuzahlen.

> **Sie müssen das nicht selbst bauen**
> Wenn Sie einen Feiertagskalender wollen statt eines Verständnisses davon, gibt es ausgereifte Open-Source-Optionen, die man kennen sollte. In der Delphi-Welt löst TZDB das verwandte Zeitzonen-problem mit echten IANA-Daten. Außerhalb decken python-holidays und Nager.Date (.NET) weit über hundert Länder inklusive Untergliederungen ab, und date-holidays tut dasselbe für JavaScript. Sie tragen die Landes- und Regionalregeln, die diese Serie bewusst auslässt. Der Grund, es selbst zu schreiben, ist, dass die Regeln für ein oder zwei Länder wirklich überschaubar sind — wie die Tabelle oben zeigt — und eine Abhängigkeit, die Sie in neunzig Zeilen vollständig verstehen, oft der bessere Handel ist. Brauchen Sie vierzig Länder, nehmen Sie die Bibliothek.

Fazit

In Teil 1 ging es eigentlich um ein einziges Refactoring: Verhalten in Daten zu verwandeln, damit der konsumierende Code aufhört zu verzweigen.

- **Ein Feiertag ist eine Funktion von einem Jahr auf ein Datum.** Geben Sie jeder Regel diese Signatur, und der aufrufende Code hat genau einen Pfad.
- **Drei Primitive decken das gesamte US-Gesetz ab**: ein festes Datum, der *n*-te Wochentag eines Monats und der *letzte* Wochentag eines Monats. `DayOfTheWeek` und `EndOfTheMonth` erledigen die schwierigen Teile.
- **„Letzter" ist weder „vierter" noch „fünfter".** Zählen Sie vom Monatsende rückwärts; Vorwärtszählen liefert stillschweigend Daten im Folgemonat.
- **Eine stabile `UID` neben dem Anzeigenamen** hält gespeicherte Daten aussagekräftig, wenn Namen übersetzt, umgeschrieben oder per Gesetz geändert werden.

> Kodieren Sie die Regel, nicht das Datum. Eine aus Regeln gebaute Feiertagstabelle wird einmal geschrieben und ist in jedem Jahr korrekt, das Sie nie getestet haben.

Der US-Kalender war das einfache Land: elf Feiertage, alle im gregorianischen Kalender verankert, in dem wir ohnehin rechnen. Deutschland ist das interessante — vier der neun bundeseinheitlichen Feiertage sind über ihren Abstand zu Ostern definiert, und Ostern ist eine lunisolare Berechnung ganz ohne geschlossene Formel.

Teil 2 setzt sie um, ergänzt die deutsche Tabelle und führt beide Länder mit derselben dreizeiligen Schleife zu einem Kalender zusammen. Bis dort.

> **Die Serie: Gesetzliche Feiertage in Delphi berechnen**
> Zwei Teile: 1. Regeln als Daten und der US-Bundeskalender **— Sie sind hier** 2. Ostern, der deutsche Kalender und zwei Länder zusammenführen Beide bauen auf dem Tutorial Datum und Uhrzeit in Delphi auf.

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 —