TUTORIAL · AUGUST 15, 2026

Feiertagskalender in Delphi

Ein zweiteiliger Kurs zur Berechnung gesetzlicher Feiertage in Delphi — wie aus Feiertagsregeln Daten statt einer case-Anweisung werden, die drei Datums-Primitiven, aus denen der gesamte US-Bundeskalender besteht, der Osteralgorithmus nach Gauß/Meeus in fünfzehn Zeilen und die Zusammenführung zweier Länder zu einem Kalender, der nie aktualisiert werden muss.

By Dr. Holger Flick

Jede Geschäftsanwendung muss irgendwann eine trügerisch einfache Frage beantworten: *Ist dieser Tag ein Arbeitstag?* Die Lohnabrechnung fragt danach. Liefertermine fragen danach. Alles, was eine Antwort „innerhalb von fünf Werktagen" zusagt, fragt danach. Und die erste Implementierung ist in nahezu jeder Codebasis eine `case`-Anweisung, die jeden Dezember einen neuen Zweig bekommt — weil jemand die Daten für das nächste Jahr fest eingetragen hat und das Jahr darauf dann ausgegangen ist.

Die Lösung ist keine größere `case`-Anweisung. Sie besteht in der Einsicht, dass ein Feiertag gar kein Datum ist — er ist eine *Regel*. Weihnachten ist immer der 25. Dezember. Thanksgiving ist der vierte Donnerstag im November. Der Memorial Day ist der *letzte* Montag im Mai, und das ist weder der vierte noch der fünfte. Und der Ostermontag ist der Tag nach einer lunisolaren Berechnung, die fünfzehn Jahrhunderte vor dem ersten Computer standardisiert wurde. Wer die Daten fest einträgt, sitzt im nächsten Jahr wieder daran. Wer die Regeln kodiert, ist ein für alle Mal fertig.

Dieses Tutorial ist genau diese Engine, ausgeliefert als zweiteilige Artikelserie und vollständig auf der RTL-Unit `System.DateUtils` aufgebaut — ohne Fremdbibliotheken, insgesamt rund einhundertzwanzig Zeilen. Teil 1 baut

die Datenstruktur und behandelt die Vereinigten Staaten. Teil 2 nimmt sich Ostern vor, Deutschland und den Moment, in dem zwei nationale Kalender in derselben Tabelle leben müssen.

Lesen Sie die beiden Teile der Reihe nach. Teil 2 erweitert die Struktur aus Teil 1, ohne eine einzige Zeile daran zu ändern — und genau das ist die Aussage.

Teil 1 — Regeln als Daten und der US-Bundeskalender

Zwei Länder, zwanzig Feiertage, eine Datenstruktur

Am Anfang steht eine Beobachtung, die das ganze Problem in sich zusammenfallen lässt: Was ein Feiertag kulturell auch sein mag — rechnerisch ist er eine Funktion, die ein Jahr entgegennimmt und ein Datum zurückgibt. Geben Sie jeder Regel diese Signatur — ein `reference to function(Year): TDate` in einem kleinen Record, dazu eine stabile UID, die es übersteht, wenn Anzeigenamen übersetzt oder per Gesetz geändert werden — und der auswertende Code hat exakt einen Pfad, ganz ohne Verzweigung nach Regeltyp. Von dort sind es drei winzige Primitiven: ein festes Datum, der *n*-te Wochentag eines Monats und der letzte Wochentag

eines Monats. Diese drei decken das gesamte US-Gesetz ab, und alle elf Bundesfeiertage ergeben sich als ein einziges, gut lesbares Array, das sich fast wie der Gesetzestext liest, den es umsetzt. Der Beitrag benennt außerdem schonungslos den Fehler, der jeden Test im Januar übersteht und im September zuschlägt — jenen, bei dem „letzter Montag im Mai" als „vierter" oder „fünfter" geschrieben wird und Ihnen klammheimlich ein Datum im Juni liefert.

Teil 2 — Ostern, der deutsche Kalender und die Zusammenführung zweier Länder

Vier deutsche Feiertage hängen an einer einzigen Zahl

An Deutschland wird der Entwurf auf die Probe gestellt, denn vier der neun bundeseinheitlichen Feiertage sind überhaupt nicht am gregorianischen Kalender verankert — sie hängen an Ostern, das sich über ein Fenster von 35 Tagen bewegt und keine geschlossene Form hat, die Sie irgendwo nachschlagen könnten. Die gute Nachricht: Der anonyme gregorianische Algorithmus passt in fünfzehn Zeilen Ganzzahlarithmetik, und sobald Sie diese eine Zahl haben, sind Karfreitag, Ostermontag, Christi Himmelfahrt und Pfingstmontag jeweils ein einziger `IncDay`-Aufruf — wobei zwei dieser Abstände nicht die Zahlen sind, die die Überlieferung nennt, und der Fehler um eins bleibt lautlos. Dann kommen die Teile, vor denen Sie niemand warnt: Deutschlands neun Feiertage sind eine *Schnittmenge* aus sechzehn Landesgesetzen und keine Bundesliste, beide Länder

behandeln Feiertage am Wochenende genau entgegengesetzt, und beim Zusammenführen beider Kalender tritt eine Datumskollision zutage, die es nur in bestimmten Jahren gibt — genug, um einen naiven Index in der Produktion abstürzen zu lassen, und zwar in genau dem einen Jahr. Die Auswertungsschleife aus Teil 1 nimmt das alles unverändert auf.

Was Sie danach können

- Jeden Feiertag — festes Datum, n-ter Wochentag, letzter Wochentag oder osterbezogen — als einheitliches `function(Year): TDate` modellieren und den gesamten Kalender in einer Schleife ohne Typverzweigung auswerten.
- Den vollständigen US-Bundeskalender und den bundeseinheitlichen deutschen Kalender für ein beliebiges Jahr erzeugen, mit `System.DateUtils` und sonst nichts.
- Den Ostersonntag von Grund auf berechnen — und wissen, warum das Ergebnis das westliche Datum ist und wonach Sie greifen müssen, wenn Sie das orthodoxe brauchen.
-

Eintritt und *Begehung* als getrennte Datumsfelder führen, damit „Ist heute Weihnachten?" und „Ist das Büro geschlossen?" zwei beantwortbare Fragen bleiben statt eines verlustbehafteten Feldes.

- Mehrere Länder zu einem Kalender zusammenführen, zwei Feiertage am selben Tag überstehen und darauf aufbauend mit Werktagen rechnen.
- Ehrlich beurteilen, wann Schluss ist: Der Beitrag nennt die gepflegten Open-Source-Alternativen und den Punkt, ab dem eine davon die bessere Wahl ist, als den Code selbst zu besitzen.

> 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.

Beide Teile bauen auf dem Tutorial Datum und Uhrzeit in Delphi auf — lesenswert vorab, falls `TDateTime` Sie schon einmal überrascht hat.

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 —