NEWS · AUGUST 3, 2026

Was ist RTTI? Delphis Reflection, Teil 1

By Dr. Holger Flick

Hier ist eine Frage, die zunächst unmöglich klingt: **Kann ein Programm seinen eigenen Code untersuchen, während es läuft?** Kann es ein Objekt betrachten, das es nie zuvor gesehen hat, fragen „Welche Properties hast du, wie heißen sie, welche Typen haben sie?" — und sie dann lesen und schreiben, ohne eine einzige Zeile, die speziell für die Klasse dieses Objekts geschrieben wurde?

Die Antwort lautet in modernem Delphi: ja. Und die Maschinerie, die das möglich macht, heißt **RTTI — Run-Time Type Information**. Sie ist der stille Motor hinter einem enormen Teil des Toolings, das Sie bereits verwenden: die Art, wie der Object Inspector der IDE die published Properties einer Komponente auflistet, wie FireDAC und ORMs Objektfelder auf Datenbankspalten abbilden, wie ein JSON-Serializer ein beliebiges Objekt in Text verwandelt und zurück. Keines dieser Werkzeuge wurde in Kenntnis *Ihrer* Klassen geschrieben — sie entdecken sie zur Laufzeit, über RTTI.

Die meisten Delphi-Entwickler profitieren seit Jahren von RTTI, ohne es je direkt aufzurufen. Das ist ein wenig schade, denn die moderne Unit `System.Rtti` — eingeführt bereits mit Delphi 2010 — macht Reflection wirklich zugänglich, und sobald der Groschen fällt, eröffnet sie eine ganze Kategorie eleganter, generischer Lösungen. Diese dreiteilige Serie ist eine behutsame, ehrliche Tour. **Teil 1 beantwortet das „Was" und „Warum"**: was RTTI eigentlich ist, die zwei Records, mit denen alles beginnt (`TRttiContext` und `TValue`), und — wichtig — wann man *nicht* danach greifen sollte. Teil 2 liest Typen, Properties und Attribute; Teil 3 ruft Methoden auf und baut ein kleines, wirklich nützliches Werkzeug. Beginnen wir mit der Idee.

Reflection, in einem Satz

Das übergeordnete Konzept trägt einen Namen, den Sie in vielen Sprachen wiederfinden: **Reflection** — die Fähigkeit eines Programms, seine eigene Struktur zur Laufzeit zu untersuchen und zu manipulieren. Java hat sie, C# hat sie, Python hat sie. In Delphi heißt das Feature RTTI, und `System.Rtti` ist die moderne, objektorientierte API dafür.

Normalerweise ist Ihr Code *spezifisch*: Um den `Name` eines `TCustomer` zu lesen, schreiben Sie `Customer.Name`, und der Compiler backt exakt ein, wo dieses Feld liegt. RTTI dreht das um. Es erlaubt Ihnen, Code zu schreiben, der mit einem Typ arbeitet, über den er bis zur Laufzeit *nichts* weiß — indem er den Typ bittet, sich selbst zu beschreiben, und dann auf die Antwort reagiert.

Normaler Code kennt den Typ zur Compile-Zeit; RTTI entdeckt ihn zur Laufzeit

Das Diagramm bringt den wesentlichen Unterschied auf den Punkt. Links ist der Property-Name `Name` *einkompiliert* — überlegen Sie es sich anders, ändern Sie den Quelltext und kompilieren neu. Rechts ist `'Name'` ein **String** — Daten, die Ihr Programm aus einer Konfigurationsdatei, einem JSON-Schlüssel oder einer Datenbankspalte erhalten kann, und auf die es reagiert, ohne je von `TCustomer` gehört zu haben. Diese Indirektion ist die gesamte Superkraft.

Warum sollte ich das jemals wollen?

Die ehrliche Frage. Reflection hat einen Preis (wir werden weiter unten offen darüber sprechen), also muss sie sich ihren Platz verdienen. Die Aufgaben, bei denen sie das tut, haben überwiegend eine gemeinsame Form: **Sie schreiben Code, der einheitlich über viele Typen funktionieren muss, die Sie nicht kontrollieren.** Ein paar konkrete Beispiele, jedes davon etwas, das Menschen tatsächlich mit `System.Rtti` bauen:

- **Serialisierung** — ein *beliebiges* Objekt in JSON/XML verwandeln und zurück, indem man seine Properties durchläuft. Die RTL-eigene Unit `REST.Json` macht genau das.
- **Objektrelationales Mapping** — auslesen, welche Felder eine Klasse hat, um `INSERT`-/`SELECT`-Anweisungen automatisch zu erzeugen.
- **Dependency Injection & IoC-Container** — den Konstruktor einer Klasse inspizieren, um herauszufinden, was zu übergeben ist.
- **Validierung & UI-Generierung** — Properties (und *Attribute* darauf) lesen, um ein Formular automatisch aufzubauen oder eine Regel zu prüfen.
-

Objekte generisch kopieren / vergleichen — eine „Klone ein beliebiges Objekt"-Routine, der egal ist, was das Objekt ist.

Der rote Faden: Ohne RTTI zwingt Sie jede dieser Aufgaben zu handgeschriebenem Boilerplate pro Klasse — eine `ToJSON`-Methode auf jeder einzelnen Klasse, für immer. Mit RTTI schreiben Sie den Walker *einmal*.

Der Startpunkt: `TRttiContext`

Alles in `System.Rtti` beginnt mit einem einzigen Record: `TRttiContext`. Stellen Sie ihn sich als Ihre Sitzung mit dem Reflection-System vor — Sie öffnen eine, stellen Fragen und schließen sie. Er ist im Quelltext als Record mit `Create` und `Free` deklariert:

```
uses
  System.Rtti;

var
  Ctx: TRttiContext;
  Typ: TRttiType;
begin
  Ctx := TRttiContext.Create;
  try
    Typ := Ctx.GetType(TButton);      // die Klasse TButton reflektieren
    ShowMessage(Typ.QualifiedName);    // 'Vcl.StdCtrls.TButton'
  finally
    Ctx.Free;
  end;
end;
```

Ein paar Dinge sind gleich hier hervorhebenswert, denn die Lebensdauerregeln sind etwas ungewöhnlich und sollten vom ersten Beispiel an richtig sitzen.

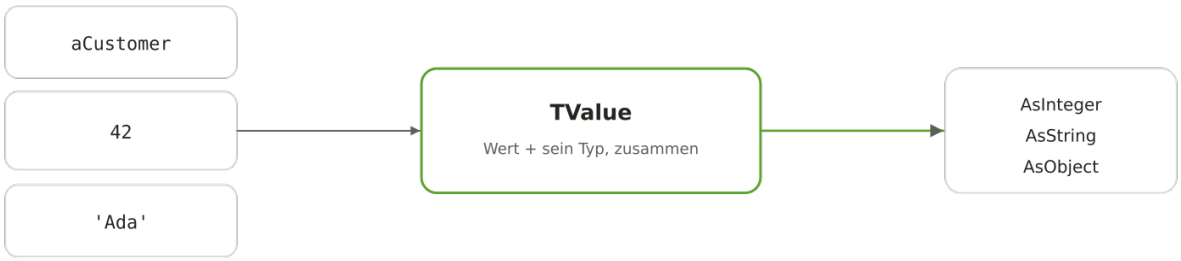
> **TRttiContext ist ein Record — trotzdem Create/Free. Und zwar deshalb**
> `TRttiContext` ist ein **Record**, das Muster aus unseren Beiträgen zu IOUtils und Regex (Records, die man nie freigibt) gilt hier also nicht. Der Grund: Der Context hält eine interne Referenz, die einen *Pool von `TRtti-Objekten am Leben und im Cache hält**. Create/Free verwalten diesen Pool. Sie *können* beides weglassen — der Context nutzt intern ein referenzgezähltes Token, es ist also kein hartes Leak — aber die idiomatische, vorhersagbare Form ist Create in einem try, Free im finally. Die Objekte, die er Ihnen aushändigt (TRttiType, TRttiProperty`, …), **gehören dem Context** — die geben Sie niemals selbst frei.

Die meistgenutzte Methode des Contexts ist `GetType`: Sie nimmt entweder eine Klassenreferenz oder einen Type-Info-Zeiger und liefert einen `TRttiType` — die Laufzeitbeschreibung dieses Typs. Dieser `TRttiType` ist das Tor zu allem in Teil 2: seinen Properties, Methoden, Feldern und Attributen. `GetType` hat einen Partner, `GetTypes`, der *jeden* Typ zurückgibt, für den das Programm RTTI besitzt, und `FindType` schlägt einen Typ über seinen qualifizierten Namen als String nach.

Der universelle Wert: `TValue`

Es gibt einen zweiten Record, den Sie kennen müssen, bevor irgendetwas anderes Sinn ergibt, denn er ist überall in der API: `TValue`. Das Problem, das er löst: Wenn Sie eine Property lesen, deren Typ Sie zur Compile-Zeit nicht kennen — was *liefert* dieser Lesevorgang dann zurück? Einen `Integer`? Einen `string`?

Ein `TObject`? Das können Sie erst zur Laufzeit wissen — RTTI braucht also einen Container, der **einen Wert beliebigen Typs** aufnehmen kann und sich dabei merkt, welcher Typ das tatsächlich ist. Dieser Container ist `TValue`.



TValue ist eine typisierte Box: sie hält einen Wert beliebigen Typs und merkt sich den Typ

Das Bild ist das Konzept: Was immer Sie hineinlegen — einen Integer, einen String, eine Objektreferenz — kommt als `TValue` heraus, das *seinen eigenen Typ kennt*, und Sie holen es mit einem typisierten Accessor wieder heraus. Hinein geht es mit `TValue.From<T>` (oder einer einfachen Zuweisung, dank impliziter Operatoren, die der Quelltext für gängige Typen deklariert), heraus mit `AsInteger`, `AsString`, `AsObject`, dem generischen `AsType<T>` oder dem sicheren `ToString`:

```
var
  V: TValue;
begin
  V := TValue.From<Integer>(42);   // einen Integer einpacken
  // ... oder einfach:  V := 42;   // der implizite Operator tut dasselbe

  if V.IsEmpty then Exit;
  ShowMessage(V.AsString);          // '42'  (Konvertierung im ToString-Stil)
  ShowMessage(V.AsInteger.ToString);// '42'  (typisierte Extraktion)
end;
```

`TValue` ist das, was eine einzige Serialisierungsroutine eine Property verarbeiten lässt, egal ob sie eine Zahl, ein Name oder ein verschachteltes Objekt ist — der „Beliebiger-Typ"-Klebstoff, der generischen Code erst möglich macht.

> **TValue ist auch für sich genommen nützlich**
>
> Sie brauchen nicht den gesamten Reflection-Apparat, um `TValue` zu schätzen. Es ist eine praktische, typsichere Alternative zu `Variant`, wann immer Sie eine Variable brauchen, die verschiedene Typen halten kann — und anders als `Variant` kann es *jeden* Typ halten, einschließlich Objekten und Records, und merkt sich den exakten Typ, statt zu koerzieren. Wenn Sie je zu `Variant` gegriffen haben, um „irgendeinen Wert irgendeines Typs" herumzureichen, ist `TValue` oft die sauberere, moderne Wahl.

Die andere Seite: Wann Sie RTTI *nicht* verwenden sollten

Ein Versprechen an die Leserschaft — Reflection ist mächtig, und genau deshalb wird sie leicht überstrapaziert. Lassen Sie mich das ehrlich einordnen.

RTTI tauscht ein wenig Laufzeitkosten und etwas Compile-Zeit-Sicherheit gegen enorme Flexibilität. Dieser Tausch ist ein *Schnäppchen*, wenn Sie eine generische Framework-Komponente schreiben (einen Serializer,

einen Mapper), die über viele Typen hinweg genutzt wird. Er ist ein *schlechtes Geschäft*, wenn Sie damit etwas erledigen wollen, das Sie direkt tun könnten.

> **Wo Reflection das falsche Werkzeug ist**
> - **Sie kennen den Typ zur Compile-Zeit.** Wenn Sie `Customer.Name` schreiben können, schreiben Sie `Customer.Name` — das ist schneller, sicherer und klarer als `GetProperty('Name')`. RTTI ist für die Fälle, in denen Sie das Member im Quelltext *nicht* benennen können. - **Heiße innere Schleifen.** Reflektiver Zugriff bedeutet Lookups und Boxing in `TValue`; für das Verdrahten beim Aufbau ist das fein, aber reflektieren Sie nicht in einer engen Pro-Pixel- oder Pro-Zeile-Schleife, wenn ein direkter Aufruf genügen würde. Einmal reflektieren, Ergebnis cachen. - **Binärgröße / Linking.** Die RTTI-Erzeugung fügt Ihrer ausführbaren Datei Metadaten hinzu, und der Compiler kann nur entfernen, was sicher ungenutzt ist. Für die meisten Desktop- und Server-Anwendungen ist das kein Thema; auf stark größenbeschränkten Zielen lohnt es sich zu wissen, dass es die Direktive `{$RTTI}` gibt, um zu steuern, was emittiert wird (ein Thema für Teil 2). Nichts davon heißt „meiden Sie RTTI". Es heißt: *Setzen Sie es dort ein, wo sich seine Flexibilität bezahlt macht* — auf Framework-Ebene, in typübergreifendem Code — und nutzen Sie überall sonst direkten Member-Zugriff.

> **Alternativen, kurz gefasst**
> Reflection ist eine ökosystemübergreifende Idee. Falls Ihre Arbeit mehrere Stacks umfasst: .NETs `System.Reflection` und Javas `java.lang.reflect` lösen dieselben Probleme mit sehr ähnlichen Formen (ein „Typ"-Objekt, das Properties/Methoden/Attribute offenlegt). Delphis `System.Rtti` wird Ihnen sofort vertraut vorkommen, wenn Sie eines von beiden benutzt haben. Die Konzepte übertragen sich; nur die Namen ändern sich.

Was Teil 1 festgehalten hat

Wir wollten beantworten, *was RTTI ist und warum man es haben will* — ehrlich eingeordnet. Das Fundament für Teil 2:

- **RTTI (Run-Time Type Information)** ist Delphis **Reflection**: Code, der Typen untersucht und manipuliert, die er zur Compile-Zeit nicht kannte. Sie treibt Serializer, ORMs, DI-Container und den Object Inspector selbst an.
- Alles beginnt mit dem Record `TRttiContext` — per `Create` anlegen, per `GetType` nach einem `TRttiType` fragen, am Ende `Free` aufrufen. Die zurückgegebenen `TRttiType`-Objekte **gehören dem Context**.
- `TValue` ist der universelle typisierte Container, mit dem generischer Code einen Wert *beliebigen* Typs halten und weiterreichen kann, ohne zu vergessen, welcher Typ es ist.
- Reflektieren Sie dort, wo sich Flexibilität lohnt — in typübergreifendem Framework-Code. Nutzen Sie direkten Member-Zugriff, wenn Sie den Typ bereits kennen. Cachen Sie reflektierte Lookups; reflektieren Sie nicht in heißen Schleifen.

> RTTI lässt Ihr Programm *Typen selbst* als Daten behandeln — es entdeckt die Properties und Methoden eines Objekts zur Laufzeit und agiert darauf, ohne eine einzige Zeile Code, die für genau diese Klasse geschrieben wurde.

Mit dem „Was" und „Warum" im Gepäck wird Teil 2 konkret: einen `TRttiType` öffnen, seine Properties und Felder durchlaufen, Werte darüber lesen und schreiben, und die **Custom Attributes** lesen, die moderne Delphi-Frameworks so deklarativ machen. Dort beginnt Reflection, sich wie Magie anzufühlen. Bis dahin!

> **Die Serie: Delphis Reflection (RTTI)**
> Drei Teile: 1. Was ist RTTI? **— Sie sind hier** 2. Typen, Properties und Attribute lesen 3. Methoden aufrufen — und ein echtes Werkzeug

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 —