NEWS · AUGUST 4, 2026

RTTI Teil 2: Typen, Properties und Attribute auslesen

By Dr. Holger Flick



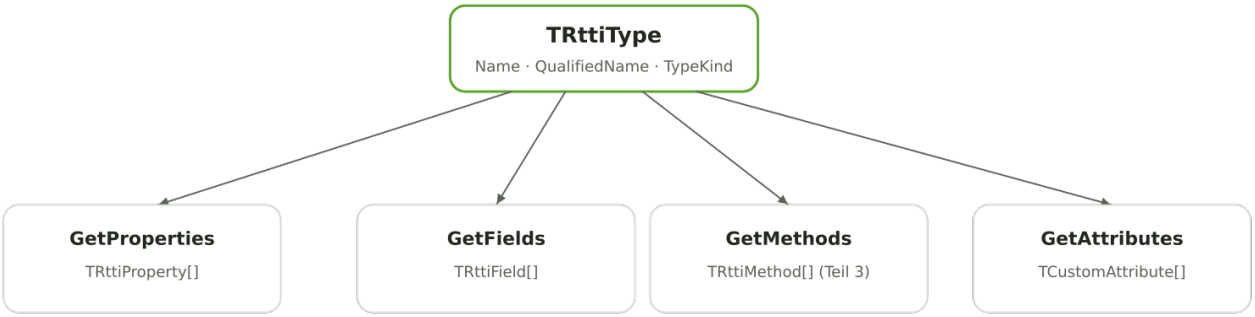
In Teil 1 haben wir die Grundidee etabliert: **RTTI ist Reflection** — Code, der Typen inspiziert und manipuliert, die er zur Compile-Zeit nicht kannte — und alles beginnt mit zwei Records: `TRttiContext` (Ihre Sitzung mit dem Reflection-System) und `TValue` (eine Box, die einen Wert beliebigen Typs aufnimmt und sich dabei merkt, welcher Typ das ist).

Jetzt bringen wir den Typ zum *Sprechen*. Das ist der Teil, in dem Reflection aufhört, eine abstrakte Idee zu sein, und zu etwas wird, dem Sie bei der Arbeit zusehen können: Sie übergeben `System.Rtti` ein Objekt, und es nennt Ihnen jede Property des Objekts, wie sie heißt, welchen Typ sie hat — und lässt Sie diese Properties dann per Name, als String, lesen und schreiben. Machen Sie das einmal, und der Reiz liegt auf der Hand; genau so kann eine einzige Routine *jede beliebige* Klasse serialisieren, validieren oder kopieren.

Teil 2 behandelt das Auslesen der Struktur: einen `TRttiType` öffnen, seine **Properties** und **Fields** durchlaufen, ihre Werte über `TValue` lesen und setzen — und dann das Feature, das moderne Delphi-Frameworks so elegant macht: **Custom Attributes**, die kleinen `[eckig geklammerten]` Annotationen, die Sie an eine Klasse hängen und zur Laufzeit wieder auslesen können. Teil 3 wird all das nutzen, um *Methoden aufzurufen* und ein kleines, echtes Werkzeug zu bauen. Lesen wir ein paar Typen.

Die Landkarte: Was ein `TRttiType` bereitstellt

Wenn Sie `Ctx.GetType(...)` aufrufen, erhalten Sie einen `TRttiType` — die Drehscheibe für alles Weitere. Bevor der Code kommt, hier die Struktur dessen, was er bietet, damit sich die folgenden Methodennamen wie eine Landkarte anfühlen und nicht wie eine Liste.



Ein TRttiType ist die Drehscheibe: von ihm aus erreichen Sie Properties, Fields, Methoden und Attribute

Das Diagramm ist Ihr Inhaltsverzeichnis für Teil 2: Von einem `TRttiType` aus erreichen Sie `GetProperties`, `GetFields`, `GetMethods` und `GetAttributes` (plus die Singular-Varianten `GetProperty`/`GetField`/`GetMethod`, um ein einzelnes Element per Name nachzuschlagen). Properties und Fields nehmen wir uns jetzt vor, Attribute danach — und Methoden heben wir uns für Teil 3 auf.

Die Properties eines Objekts durchlaufen

Machen wir es konkret — mit einer kleinen Klasse und einer Routine, die *jede beliebige* Instanz *jeder beliebigen* Klasse beschreibt: das „Hello World" der Reflection. Angenommen, wir haben:

```
type
  TCustomer = class
  private
    FName: string;
    FAge: Integer;
  public
    property Name: string read FName write FName;
    property Age: Integer read FAge write FAge;
  end;
```

Hier ist eine Routine, die von jeder Property Name, Typ und aktuellen Wert ausgibt — und beachten Sie: `TCustomer` kommt darin *nirgends* vor. Sie nimmt ein schlichtes `TObject` entgegen:

```
uses
  System.Rtti;

procedure Describe(const Obj: TObject);
var
  Ctx: TRttiContext;
  Typ: TRttiType;
  Prop: TRttiProperty;
begin
  Ctx := TRttiContext.Create;
  try
    Typ := Ctx.GetType(Obj.ClassType);
    for Prop in Typ.GetProperties do
      Writeln(Format('%s: %s = %s',
        [Prop.Name,
         Prop.PropertyType.Name,
         Prop.GetValue(Obj).ToString]));
  finally
    Ctx.Free;
  end;
end;
```

Rufen Sie `Describe(SomeCustomer)` auf, und Sie erhalten:

```
Name: string = Ada Lovelace
Age: Integer = 36
```

Sehen Sie sich an, was passiert ist. `GetProperties` lieferte ein `TArray<TRttiProperty>`; jede `TRttiProperty` kennt ihren eigenen `Name`, ihren `PropertyType` (selbst wieder ein `TRttiType`) und — der entscheidende Zug — `GetValue(Obj)` liest diese Property *aus dem lebenden Objekt* aus und reicht sie als `TValue` zurück, dessen `ToString` wir ausgeben. Diese eine Routine funktioniert jetzt für `TCustomer`, `TInvoice`, `TButton` — oder eine Klasse, die es noch gar nicht gibt.

Lesen *und Schreiben* per Name

Reflection liest und schreibt. Das Spiegelbild von `GetValue` ist `SetValue`, und zusammen erlauben sie Ihnen, eine Property zu setzen, wenn ihr Name als *Daten* ankommt — aus einem JSON-Key, einem Konfigurationseintrag, einer Datenbankspalte. Hier ein kleiner generischer Setter:

```
procedure SetProp(const Obj: TObject; const PropName: string; const Value: TValue);
var
  Ctx: TRttiContext;
  Prop: TRttiProperty;
begin
  Ctx := TRttiContext.Create;
  try
    Prop := Ctx.GetType(Obj.ClassType).GetProperty(PropName);
    if (Prop <> nil) and Prop.IsWritable then
      Prop.SetValue(Obj, Value);
  finally
    Ctx.Free;
  end;
end;

// Verwendung – der Property-Name ist ein Laufzeit-String:
SetProp(Customer, 'Name', 'Grace Hopper');
SetProp(Customer, 'Age', 45);
```

`GetProperty(PropName)` schlägt eine einzelne Property über ihren String-Namen nach (und liefert `nil`, wenn es keine gibt), `IsWritable` schützt vor schreibgeschützten Properties, und `SetValue` schreibt den neuen Wert in das Objekt. Der Parameter `Value: TValue` ist der Grund, warum die Box aus Teil 1 so wichtig war: Sie erlaubt einer einzigen Routine, hier einen Integer und dort einen String entgegenzunehmen.



GetValue liest eine Property aus dem lebenden Objekt in ein TValue; SetValue schreibt sie zurück

Das Diagramm zeigt den kompletten Lese-/Schreibzyklus: `GetValue` holt den aktuellen Inhalt der Property in ein `TValue`; `SetValue` schiebt ein `TValue` wieder hinein. Jeder generische Serializer und Object-Mapper ist im Kern eine Schleife um diese beiden Aufrufe.

Fields vs. Properties — eine kurze, wichtige Unterscheidung

Ihnen wird auch `GetFields` begegnen, und es lohnt sich ein Satz dazu, wann Sie was verwenden — denn dasselbe ist es nicht.

Properties sind die öffentliche Oberfläche (`Name`, `Age`) — oft von Gettern/Settern getragen, und fast immer das, was Sie für Serialisierung wollen, weil sie den logischen Zustand des Objekts repräsentieren. **Fields** sind der rohe Speicher (`FName`, `FAge`) dahinter. `GetFields` funktioniert identisch — `Field.FieldType`, `Field.GetValue(Obj)`, `Field.SetValue(Obj, V)` — greift aber direkt auf die zugrunde liegenden Daten zu. Bevorzugen Sie Properties, es sei denn, Sie brauchen gezielt den Speicher (manche Serializer lesen absichtlich Fields, um Setter mit Seiteneffekten zu umgehen). Die API ist symmetrisch — wer die eine kennt, kennt auch die andere.

> **Sichtbarkeit: Was RTTI sehen kann**
> Member tragen eine Sichtbarkeit — die RTL enumeriert sie als `mvPrivate`, `mvProtected`, `mvPublic`, `mvPublished`. Standardmäßig erzeugt der Compiler vollständige RTTI für **public- und published-Member** von Klassen, was für die meiste Reflection ausreicht. Wenn Sie private/protected Member reflektieren müssen — oder die erzeugten Metadaten aus Größengründen *reduzieren* wollen — steuert die Compiler-Direktive `{$RTTI}` genau, was generiert wird; siehe die Dokumentation zur RTTI-Direktive. Für die alltägliche Serialisierung von public/published Properties funktionieren die Voreinstellungen einfach.

Attribute: Annotationen, die Sie zur Laufzeit auslesen können

Hier kommt das Feature, das RTTI von „nützlich" zu „elegant" erhebt. Ein **Custom Attribute** ist ein kleines Stück Metadaten, das Sie in eckigen Klammern an eine Klasse, Property, ein Field oder eine Methode hängen — und über RTTI zur Laufzeit wieder auslesen können. Wenn Sie schon einmal mit einem modernen Delphi-ORM oder -Serializer etwas wie `[Column('customer_name')]` über eine Property geschrieben haben, haben Sie Attribute benutzt; so werden deklarative Frameworks gebaut.

Ein Attribut ist einfach eine Klasse, die von `TCustomAttribute` abstammt (deklariert in `System.pas` als Basis von allen). Sie definieren eines mit einem Konstruktor und wenden es dann in Klammern an:

```
type
  // 1. Ein Attribut definieren – eine gewöhnliche Klasse mit Konstruktor
  ColumnAttribute = class(TCustomAttribute)
  private
    FName: string;
  public
    constructor Create(const AName: string);
    property Name: string read FName;
  end;

  // 2. Auf Properties anwenden – und dabei zusätzliche Metadaten mitgeben
  TCustomer = class
  private
    FName: string;
    FAge: Integer;
  public
    [Column('customer_name')]
    property Name: string read FName write FName;
    [Column('customer_age')]
    property Age: Integer read FAge write FAge;
  end;
```

Und jetzt die Belohnung — diese Attribute wieder auslesen, um etwa den Datenbank-Spaltennamen jeder Property zu ermitteln, ohne ein einziges Mapping fest zu verdrahten:

```
var
  Ctx: TRttiContext;
  Prop: TRttiProperty;
  Attr: TCustomAttribute;
begin
  Ctx := TRttiContext.Create;
  try
    for Prop in Ctx.GetType(TCustomer).GetProperties do
      for Attr in Prop.GetAttributes do
        if Attr is ColumnAttribute then
          Writeln(Format('%s -> column "%s"',
            [Prop.Name, ColumnAttribute(Attr).Name]));
  finally
    Ctx.Free;
  end;
end;
```

Ausgabe:

```
Name -> column "customer_name"
Age -> column "customer_age"
```

Das ist die Mapping-Schicht eines ORMs im Miniaturformat. Die Klasse *deklariert* ihre Datenbankspalten direkt neben den Properties, und eine generische Routine *liest* diese Deklarationen, um SQL zu bauen — keine separate Mapping-Datei, kein Code pro Klasse.

> **Die typisierte Abkürzung: GetAttribute<T> und HasAttribute<T>**
> Die Schleife mit Typtest per `is` funktioniert, aber die RTL bietet auf jedem reflektierten Objekt sauberere generische Helfer: `HasAttribute<ColumnAttribute>` liefert einen Boolean, und `GetAttribute<ColumnAttribute>` liefert das Attribut bereits auf seinen Typ gecastet (oder `nil`). Die innere Prüfung wird damit zu:- ``pascal var Col := Prop.GetAttribute<ColumnAttribute>; if Col <> nil then  Writeln(Prop.Name + ' -> ' + Col.Name); `` Sauberer, und kein manuelles Casten. Bevorzugen Sie diese Helfer, wenn Sie einen bestimmten Attributtyp suchen.

Die andere Seite: Attribute sind Metadaten, keine Magie

Ehrlichkeit gegenüber dem Leser — Attribute sind wunderbar, aber man sollte nüchtern sehen, was sie sind.

Ein Attribut tut *von sich aus nichts*. `[Column('customer_name')]` erzeugt keine Spalte und ändert nicht das Verhalten der Property; es sind inerte Daten, bis irgendein Code sie gezielt über `GetAttributes` ausliest und handelt. Das ist ein Feature — es hält Annotation und Verhalten entkoppelt — aber es bedeutet: Der Wert eines Attributs liegt vollständig im Framework, das es interpretiert. Definieren Sie Attribute, wenn Sie ein solches Framework *bauen* (oder eines konfigurieren, das sie liest). Streuen Sie keine Attribute, die nichts ausliest; das ist Dokumentation mit Laufzeitkosten, kein Verhalten.

Fazit

Teil 2 hat einen `TRttiType` von einem bloßen Namen in eine vollständig erkundbare Struktur verwandelt. Was Sie in Teil 3 mitnehmen sollten:

- Von einem `TRttiType` aus enumerieren `GetProperties`/`GetFields` (und die Singular-Varianten `GetProperty`/`GetField`) die Member eines Objekts; jedes kennt seinen `Name` und seinen Typ.
- `GetValue(Obj)` liest ein Member als `TValue` aus einem lebenden Objekt; `SetValue(Obj, V)` schreibt eines zurück. Dieses Lese-/Schreibpaar ist der Kern jedes generischen Serializers und Mappers.
- Bevorzugen Sie **Properties** für den logischen Zustand; greifen Sie nur dann zu **Fields**, wenn Sie den rohen Speicher brauchen. Die Standard-RTTI deckt public/published Member ab; `{$RTTI}` regelt den Rest.
- **Custom Attributes** (Subklassen von `TCustomAttribute`) hängen deklarative Metadaten an, die Sie mit `GetAttributes` oder dem typisierten `GetAttribute<T>` auslesen — das Fundament von ORMs, Validatoren und Serializern. Sie sind inert, bis Code sie liest.

> Reflection liest und schreibt: `GetValue`/`SetValue` lassen eine einzige Routine die Properties jedes Objekts per Name behandeln, und Custom Attributes lassen eine Klasse *deklarieren*, wie ein Framework sie behandeln soll.

Teil 3 ergänzt das letzte Verb — **Methoden per Name aufrufen** mit `TRttiMethod.Invoke` — und rundet die Serie ab, indem wir ein kleines, wirklich nützliches Reflection-Werkzeug bauen. Bis zum Finale.

> **Die Serie: Delphis Reflection (RTTI)**
> Drei Teile: 1. Was ist RTTI? 2. Typen, Properties und Attribute auslesen **— Sie sind hier** 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 —