NEWS · AUGUST 5, 2026

RTTI Teil 3: Methoden aufrufen — und ein echtes Werkzeug

By Dr. Holger Flick

Wir haben einen weiten Weg zurückgelegt. Teil 1 hat erklärt, was **RTTI** ist — Reflection, also Code, der Typen inspiziert, die er zur Compile-Zeit nicht kannte — und `TRttiContext` und `TValue` eingeführt. Teil 2 hat die **Properties, Felder und Attribute** eines Objekts gelesen und Werte per Name mit `GetValue`/`SetValue` zurückgeschrieben. Ein Verb fehlt noch, und es ist das spektakulärste: **eine Methode über ihren Namen aufrufen — als String.**

Wenn sich das Lesen einer Property per Name wie ein kleiner Zaubertrick anfühlte, ist das Aufrufen einer Methode per Name die ganze Show. Es bedeutet, dass Sie `DoThing` auf einem Objekt aufrufen können, wenn der String `'DoThing'` erst zur Laufzeit eintrifft — aus einem Skript, einer Konfigurationsdatei, einer Nachricht von einem Server, einem Plugin. Genau so funktionieren Command-Dispatcher, Scripting-Brücken und Test-Runner.

Teil 3 behandelt die Invocation — `TRttiMethod.Invoke`, das Übergeben von Argumenten und das Auslesen von Rückgabewerten als `TValue` — und dann, um die ganze Serie abzurunden, bauen wir mit allem Gelernten ein kleines, wirklich nützliches Werkzeug: einen **generischen Objekt-zu-CSV-Exporter**, der mit jeder Liste jeder Klasse funktioniert — allein mit Reflection. Bringen wir es stark zu Ende.

Eine Methode über ihren Namen aufrufen

Der Aufbau spiegelt Teil 2 exakt wider: Context öffnen, den Typ holen — aber diesmal fragen wir nach einer **Methode** und rufen sie per `Invoke` auf. Angenommen, wir haben eine Klasse mit einer Methode, die etwas tut und ein Ergebnis zurückgibt:

```
type
  TCalculator = class
  public
    function Add(A, B: Integer): Integer;
    procedure Reset;
  end;
```

So rufen Sie `Add(2, 3)` auf, wenn `'Add'` ein Laufzeit-String ist — beachten Sie, dass die Argumente als **Array von** `TValue` hineingehen und das Ergebnis *als* `TValue` zurückkommt:

```
uses
  System.Rtti;

var
  Ctx: TRttiContext;
  Calc: TCalculator;
  Method: TRttiMethod;
  Res: TValue;
begin
  Calc := TCalculator.Create;
  Ctx := TRttiContext.Create;
  try
    Method := Ctx.GetType(TCalculator).GetMethod('Add');
    Res := Method.Invoke(Calc, [2, 3]);      // Add(2, 3) aufrufen
    ShowMessage(Res.AsInteger.ToString);      // '5'
  finally
    Ctx.Free;
    Calc.Free;
  end;
end;
```

Betrachten Sie die Signatur, die uns die RTL gibt: `Invoke(Instance: TObject; const Args: array of TValue): TValue`. Jedes Argument ist ein `TValue` (die Integer `2` und `3` boxen sich selbst über die impliziten Operatoren aus Teil 1), und der Rückgabewert ist ein `TValue`, den Sie mit `AsInteger`, `AsString` oder dem jeweils passenden Accessor auspacken. Bei einer Prozedur wie `Reset`, die nichts zurückgibt, ignorieren Sie das (leere) Ergebnis einfach.

Invoke: Name + geboxte Argumente gehen hinein, ein geboxtes Ergebnis kommt zurück

Das Diagramm ist der gesamte Mechanismus: Ein Methodenname plus geboxte Argumente gehen hinein, die Methode läuft auf Ihrem lebenden Objekt, und ein geboxtes Ergebnis kommt zurück. Ein Command-Dispatcher ist genau dieser Aufruf, bei dem Name und Argumente aus einer eingehenden Nachricht stammen.

> **Invoke konstruiert auch Objekte und ruft Klassenmethoden auf**
> `Invoke` hat auch Overloads für eine `TClass` und eine `TValue`-Instanz — das heißt, Sie können **Konstruktoren** reflektiv aufrufen. Holen Sie die `Create`-Methode über `GetMethod('Create')` und rufen Sie sie per `Invoke` auf der *Klasse* auf, um eine Instanz zu erzeugen, wenn Sie die Klasse erst zur Laufzeit kennen.
>
> Genau so instanziieren Dependency-Injection-Container die Typen, nach denen sie gefragt werden. Wir bauen hier keinen vollständigen Container, aber es ist dasselbe `Invoke`, das Sie gerade gesehen haben.

> **Invoke braucht die richtigen Argumente — Abweichungen knallen zur Laufzeit**
> Das Sicherheitsnetz, das der Compiler Ihnen normalerweise spannt, fehlt hier: Wenn Sie die falsche Anzahl oder die falschen Typen von Argumenten übergeben, bekommen Sie keinen Compile-Fehler, sondern einen `EInvocationError` (oder einen Konvertierungsfehler) *zur Laufzeit*. Das ist der fundamentale Tausch der Reflection: Flexibilität gegen Compile-Zeit-Prüfung. Sichern Sie Invocations ab, die Sie aus externen Eingaben bauen, und nutzen Sie die typisierten Helfer der RTL, wo es geht. Genau deshalb greifen Sie — wie Teil 1 betont hat — nur dann zu `Invoke`, wenn Sie die Methode wirklich nicht im Quelltext benennen können.

Das Finale: ein generischer CSV-Exporter

Setzen wir alle drei Teile für etwas ein, das Sie tatsächlich ausliefern könnten. Das Ziel: **jede `TObjectList<T>` nach CSV exportieren** — Kopfzeile aus den Property-Namen, eine Zeile pro Objekt — mit einer einzigen Routine, die nichts über die Klassen weiß, die man ihr übergibt. Das ist genau die Art kleiner Gewinn, für die Reflection perfekt ist.

Wir lassen eine Klasse eine Property sogar per Custom-Attribut *vom Export ausnehmen* — ein Rückgriff auf Teil 2. Zuerst das Attribut:

```
type
  // Markiert eine Property, die der Exporter überspringen soll (Attribute aus Teil 2)
  NoExportAttribute = class(TCustomAttribute);

  TEmployee = class
  private
    FName: string;
    FSalary: Currency;
    FPasswordHash: string;
  public
    property Name: string read FName write FName;
    property Salary: Currency read FSalary write FSalary;
    [NoExport]                                  // niemals exportieren
    property PasswordHash: string read FPasswordHash write FPasswordHash;
  end;
```

Nun der Exporter. Er läuft einmal über die Properties, um die Kopfzeile zu bauen (und überspringt alles, was mit `[NoExport]` markiert ist), dann über die Werte jedes Objekts, um die Zeilen zu erzeugen — mit `GetProperties`, `GetValue`, `HasAttribute<T>` und `TValue.ToString` aus der ganzen Serie:

```
uses
  System.Rtti, System.SysUtils, System.Classes, System.Generics.Collections;

function ExportToCsv<T: class>(const Items: TObjectList<T>): string;
var
  Ctx: TRttiContext;
  Typ: TRttiType;
  Props: TArray<TRttiProperty>;
  Prop: TRttiProperty;
  Item: T;
  Line: TArray<string>;
  SB: TStringBuilder;

  // Property nur aufnehmen, wenn sie lesbar und nicht mit [NoExport] markiert ist
  function Exported(const P: TRttiProperty): Boolean;
  begin
    Result := P.IsReadable and not P.HasAttribute<NoExportAttribute>;
  end;

begin
  Ctx := TRttiContext.Create;
  SB := TStringBuilder.Create;
  try
    Typ := Ctx.GetType(TClass(T));
    Props := Typ.GetProperties;

    // 1. Kopfzeile – Property-Namen
    Line := [];
    for Prop in Props do
      if Exported(Prop) then
        Line := Line + [Prop.Name];
    SB.AppendLine(string.Join(',', Line));

    // 2. Eine Zeile pro Objekt – Property-Werte per Reflection
    for Item in Items do
    begin
      Line := [];
      for Prop in Props do
        if Exported(Prop) then
          Line := Line + [Prop.GetValue(Item).ToString];
      SB.AppendLine(string.Join(',', Line));
    end;

    Result := SB.ToString;
  finally
    SB.Free;
    Ctx.Free;
  end;
end;
```

Geben Sie ihm eine Liste von Mitarbeitern, und Sie erhalten:

```
Name,Salary
Ada Lovelace,95000
Grace Hopper,102000
```

`PasswordHash` fehlt — das `[NoExport]`-Attribut hat seinen Dienst getan — und, entscheidend: **Diese Routine exportiert `TCustomer`, `TInvoice` oder jede Klasse, die Sie nächstes Jahr schreiben, ohne eine einzige Änderung.** Das ist das ganze Versprechen der Reflection, eingelöst in rund vierzig Zeilen: den Walker einmal schreiben, für immer profitieren. Hier der Ablauf auf einen Blick.

Ein generischer Walker: Property-Namen werden zur Kopfzeile, Werte zu Zeilen

Das Diagramm ist das Werkzeug in einer Zeile: Eine Liste unbekannter Objekte geht hinein, der Reflection-Walk macht aus Property-Namen eine Kopfzeile und aus Property-Werten Zeilen, und CSV kommt heraus — nirgendwo klassenspezifischer Code.

> **Machen Sie es produktionsreif, bevor Sie es ausliefern**
> Um das Beispiel fokussiert zu halten, lässt es reale Feinheiten weg: **CSV-Escaping** (Werte mit Kommas, Anführungszeichen oder Zeilenumbrüchen müssen gemäß RFC 4180 in Anführungszeichen gesetzt werden), kulturabhängige Zahlen-/Datumsformatierung und die Wahl zwischen Properties und Feldern. Ergänzen Sie das, und es ist wirklich auslieferbar. Der *Reflection*-Teil — der schwierige Teil — ist bereits vollständig und korrekt; der Rest ist gewöhnliche Formatierung.

Die andere Seite: Ist Reflection hier die richtige Wahl?

Ein Versprechen an die Leserin und den Leser — auch bei einem schönen Werkzeug: Fragen Sie, ob Reflection ihren Platz verdient. Für einen CSV-Exporter, der *beliebige* Klassen quer durch Ihre Codebasis verarbeiten muss: ja. Die Alternative wäre ein handgeschriebener Exporter pro Klasse — genau der Boilerplate, den Reflection ausradieren soll. Wenn Sie aber nur je *eine* bekannte Klasse exportieren, ist eine schlichte handgeschriebene Funktion einfacher, schneller und sicherer — kein `TValue`-Boxing, volle Compile-Zeit-Prüfung. Die Regel aus Teil 1 gilt bis zum Schluss: **Reflektieren Sie, wenn Ihr Code viele Typen abdecken muss, die Sie nicht kontrollieren; schreiben Sie direkten Code, wenn Sie den Typ kennen.** Der Exporter qualifiziert sich, weil „jede Klasse" schon in seiner Signatur steht.

> **Alternativen, die man kennen sollte**
> Sie müssen Serialisierung selten von Grund auf selbst bauen. Die RTL-Unit `REST.Json` erledigt Objekt”JSON bereits per RTTI, und ausgereifte Open-Source-Bibliotheken wieDelphi-JSON / JsonDataObjects sowie ORMs wie mORMot stützen sich für Serialisierung und Mapping stark auf Reflection. Über die Stacks hinweg machen .NETs `System.Text.Json` und Javas Jackson denselben Job mit denselben reflektiven Ideen. Greifen Sie zu einer bewährten Bibliothek, wenn eine passt; bauen Sie Ihr eigenes reflektives Werkzeug, wenn Ihr Bedarf speziell ist — jetzt wissen Sie genau, wie sie von innen funktionieren.

Fazit — die ganze Serie

Drei Teile haben uns von „Was ist RTTI" zu einem funktionierenden Werkzeug geführt. Das vollständige Bild:

- **Eine Methode per Name aufrufen** mit `TRttiMethod.Invoke(Instance, [args])` — Argumente gehen als `TValue` hinein, das Ergebnis kommt als `TValue` zurück. Die Overloads erlauben sogar den reflektiven Aufruf von Konstruktoren und Klassenmethoden (das Herz von DI-Containern).
- Reflection **verliert die Compile-Zeit-Prüfung**: Falsche Argumente lösen zur Laufzeit einen `EInvocation-Error` aus. Sichern Sie Invocations ab, die aus externen Eingaben gebaut werden.
- Der **Exporter als Finale** zeigt den Ertrag: `GetProperties` + `GetValue` + `HasAttribute<T>` ergeben einen CSV-Writer, der mit *jeder* Klasse funktioniert, einmal geschrieben — die Essenz jedes Serializers und Mappers.
- Der rote Faden aller drei Teile: **Reflektieren Sie, wenn Ihr Code Typen abdecken muss, die Sie nicht kontrollieren; schreiben Sie direkten Code, wenn Sie den Typ kennen.**

> Die Invocation vervollständigt das Trio — Lesen (`GetValue`), Schreiben (`SetValue`) und Aufrufen (`Invoke`) jedes Members per Name — und macht aus „Typen als Daten" funktionierende Werkzeuge wie einen klassenunabhängigen CSV-Exporter in wenigen Dutzend Zeilen.

Damit schließt die RTTI-Serie. Wenn Sie die Grundlagen noch einmal brauchen: Teil 1 ist das „Was und Warum", und Teil 2 behandelt Properties und Attribute. Sie verstehen jetzt die Maschinerie hinter Delphis Serializern, ORMs und DI-Containern — und Sie können Ihre eigene bauen, wenn der Bedarf wirklich Ihrer ist. Reflektieren Sie mal drüber.

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

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 —