Extended RTTI support in Free Pascal

Michaël Van Canneyt

March 3, 2024

Abstract

For a long time, Free Pascal lacked support for Extended RTTI. Recently, the Extended RTTI support has been merged to the main development tree, making Free Pascal again more compatible to Delphi. In this article we take a closer look.

1 Introduction

Introspection is the ability to examine the structure of data (or types) at runtime, without knowing exactly what type the data is: it allows you to examine the type using some API provided by the programming language. In Delphi and Free Pascal, the introspection mechanism is labelled RTTI: Run-Time Type Information.

Since its inception, Delphi has supported a limited form of RTTI: The published properties of a class could (and still can) be examined with the aid of the RTTI API: at first the TypInfo unit, later the System.Rtti unit. This mechanism is used to implement streaming, and streaming is what makes form files work, both for FMX and the VCL.

Free Pascal has supported this form of RTTI for a very long time. Delphi 2010 extended the RTTI system from not only published properties to all elements of a class: fields, properties, and methods could be examined using this "Extended RTTI", not just for published properties, but also for private, protected and public fields. Additionally, it introduced Atttributes: a system to annotate the class and its fields, properties and methods with extra information.

Free Pascal introduced support for Attributes quite early on, but they were limited to published properties. It took more than 10 years to catch up and implement, and several more years to merge the support for Extended RTTI in the main development compiler. The largest part of the work has now come to an end, and extended RTTI and attributes can now be used.

2 Why Extended RTTI ?

Before diving into the details of the RTTI Api and the possibilities, why would you want to use RTTI ? One of the main uses of RTTI is streaming: it allows you to examine a class and write the contents of this class to file or database. Later the information in file or database can be read and used to reconstruct the object. All this can be done by a standalone system, that does not need to know the details of the classes it writes or reads: it can use the RTTI to examine the classes and to manipulate the classes.

To make this more specific, take the `TTimer` class:

```
TComponent= class (TPersistent)
published
  Property Name : string Read FName Write SetName;
end;

TTimer = class (TCustomTimer)
published
  property Enabled: Boolean read FEnabled write SetEnabled default True;
  property Interval: Cardinal read FInterval write SetInterval default 1000;
  property OnTimer: TNotifyEvent read FOnTimer write SetOnTimer;
  property OnStartTimer: TNotifyEvent read FOnStartTimer write FOnStartTimer;
  property OnStopTimer: TNotifyEvent read FOnStopTimer write FOnStopTimer;
end;
```

Through the RTTI, a streaming system knows that the `TTimer` class has 6 properties that it can stream. It knows the type of the property (for instance `Cardinal` for Interval), how it must be read (directly from the field `FInterval`) or written (through the `SetInterval` method), and what the default value is (1000): the default value is a way to optimize the writing by not writing the value of a property if it matches the default value.

The same mechanism allows the property inspector to show objects that it does not know.

The RTTI above had quite some limitations: Only published properties can be examined. The type of information that could be published was also limited: classes descendent from TPersistent, ordinal types, set types (up to 32 elements) and float types. Methods had to be published as well in order to be usable as event handlers.

So no records, or no arrays, no public or protected properties or simply fields. The absence of annotations meant also that the streaming system had no possibility to store extra information (for instance an alternative name for a property).

With the introduction of Extended RTTI and Attributes, these restrictions were lifted: Information of any type can be examined, fields can be examined. All visibilities (public, protected, private) are now subject to inspection.

3 Why Attributes ?

To see why annotations can be useful, assume you have a external REST service which serves and consumes data through JSON. For example a JSON contact person object

```
{
  "first-name" : "michael",
  "last-name" : "Van Canneyt"
  "date-of-birth" : "19700707"
}
```

If you wish to model this data in an object, you need to use pascal identifiers, and pascal types:

```
TPerson = record
  firstname : string;
  lastname : string;
```

```
  birthdate : TDateTime;
end;
```

Two problems become immediatly apparent: the fieldnames (keys) in the JSON are not valid pascal identifiers, so you need to map somehow the names used in JSON to the fieldnames and vice versa. The date field needs to be read (and written) in a format which is not handled by a simple `StrToDate` or `DateToStr` method. If multiple REST resources are used, then you need a mapping between the class and the REST resource to be consumed.

One way of doing this is creating some data structure which contains the mapping and which provide the format. Another way would be to have all data structures descend from a basic class which provides some methods to create the mapping:

```
TPerson = Class
  class function ResourceName : String;
  class function FieldToJSONName(const aField : String): String;
  class function FormatDate(aDate : TDatetime) : String;
  firstname : string;
  lastname : string;
  birthdate : TDateTime;
end;
```

If you additionally wish to save the data into a database using some ORM technology, then it becomes even more complicated: The names of the database fields can also be different from the fields used in the class. A similar technique can be used to introduce a second mapping for the database;

```
TPerson = Class
  Class function TableName : string;
  class function FieldToJSONName(const aField : String): String;
  class function ResourceName : String;
  class function FieldToDBField(const aField : String): String;
  class function FormatDate(aDate : TDatetime) : String;
  firstname : string;
  lastname : string;
  birthdate : TDateTime;
end;
```

The result usually is a multitude of functions in the declaration (and their implementation, of course) to take care of the necessary mappings and provide information on how and what to stream.

Attributes can make this a lot simpler: The same information can be provided with simple annotations on the fields themselves:

```
[Table('People')]
[Resource('/REST/Contact')]
TPerson = Record
  [DBKeyField]
  id : integer;

  [JSONKey('first-name')]
  [DBField('pe_name')]
  firstname : string;
```

```
  [JSONKey('last-name')]
  [DBField('pe_lastname')]
  lastname : string;

  [JSONKey('date-of-birth')]
  [DateFormat('yyyymmdd')]
  [DBField('pe_birthdate')]
  birthdate : TDateTime;
end;
```

The above uses 5 attributes:

Table specifies the database table in which to save the object.

Resource specifies the URL at which the resource can be retrieved

JSONKey specifies the key to use when reading/writing the JSON Object.

DBField specifies the database fieldname to use when loading/saving from/to the database.

DateFormat specifies the formatting used in JSON for a date.

A system that consumes REST services and a second system that loads/saves objects to database can inspect these attributes using RTTI and use them to load or save the objects in the correct location and with the correct format. In a realistic system, more attributes will be needed, of course.

If you have only 1 service to create and one database table to maintain, using attributes may be overkill. But with many tables and many REST resources to create or consume, the use of attributes avoids writing a lot of functions to provide the necessary metadata for the systems that interact with the REST service or database.

From the declaration of the object the programmer can also immediatly see how the mappings are defined, without needing to dive into functions that provide the same information.

Of course, it can be argued that this meta-information should not be inside the object itself, that these mappings should be constructed outside the business objects: the above approach introduces a coupling between the used JSON streaming framework and the business objects. Similarly for the ORM (Object-Relational Mapping: the framework to save objects to a database). Additionally, it does not cover all possible use scenarios, for example what if an object needs to be loaded from one database and saved in another database with a different structure ? This questions are legitimate, but are the subject of a separate discussion: here we just want to illustrate a possible use of attributes.

4 The System.Rtti unit

The classical RTTI as it existed since Delphi 1 (or Free Pascal) could be examined through the routines in the `TypInfo` unit. These were low-level routines, requiring manually allocating memory and using pointers.

With the introduction of Extended RTTI, a new API to consult and use the RTTI was also introduced:

The extended RTTI needs to be examined using the `System.Rtti` unit. It provides the same functionality as the `TypInfo` unit offered, but offers more convenient APIs: record, objects.

For every type and kind of data, the RTTI unit contains a dedicated object which represents the RTTI data for that type. Here are the main classes:

TRttiType This is the parent class for all types.

TRttiInstanceType Represents the RTTI for a class type.

TRttiRecordType Represents the RTTI for a record.

TRttiField Represents the RTTI for a field of a record or class.

TRttiProperty Represents the RTTI for a property of a record or class.

TRttiMethod Represents the RTTI for a method of a record or class.

These classes offer methods to find related RTTI: for a class, its list of fields, for a method, its parameters. Almost all classes have a method to get the Attributes associated with the data or type they represent. The property and field classes have a method to set the value of the field or propery, given an instance. The method classes can be invoked, allowing you to call any method without knowing the exact calling mechanism. To make all this possible without using the actual types, a `TValue` type has to be used: this new type is much like a variant, but offers an exacter type match than a variant. It it used to set/get property or field values, it is used to return values of functions you call and has to be used to supply parameter values for methods that need parameters.

The APIs in this unit use classes, but the lifetime of these classes are completely managed by the unit: there is no need to free them. To make this possible a `TRttiContext` must be used when calling the APIs of the Rtti unit: when the context is freed, all instances of RTTI classes that are no longer used will be released.

Note that in Free Pascal the routines in the Rtti unit are just a convenience layer on top of the routines in the `TypInfo` unit: everything that can be done with the Rtti unit can also be done with the routines in the `TypInfo` unit (except the `Invoke` functionality to call a method). In Delphi, this is not the case: some functonalities are only present in the `Rtti` unit. So if you have code that you want to run in FPC and in Delphi, you should stick to using the `Rtti` unit.

5 Using the extended RTTI and Attributes

So, how can we use the Extended RTTI and the Attributes? We'll demonstrate this by expanding on the example given above: We'll create some routines that make a JSON object from data in any record, after reading this record from the database. We've seen that 5 attributes were used. We'll start by introducing the complete code that defines the person record.

```
unit contact;

{$mode objfpc}
{$H+}
{$modeswitch prefixedattributes}
{$modeswitch advancedrecords}
```

```
interface

uses
  Classes, SysUtils, rest.types, rttitojson.types, objtodb.types;

{$RTTI EXPLICIT Fields[vcPublic]}

Type
  [Table('People')]
  [Resource('/REST/Contact')]
  TPerson = Record
    [DBKeyField]
    id : integer;

    [JSONKey('first-name')]
    [DBField('pe_name')]
    firstname : string;

    [JSONKey('last-name')]
    [DBField('pe_lastname')]
    lastname : string;

    [JSONKey('date-of-birth')]
    [DateFormat('yyyymmdd')]
    [DBField('pe_birthdate')]
    birthdate : TDateTime;
  end;
```

An extra attribute (DBKeyField) is used to indicate the key field for the record.

The block with modeswitch directives instruct the Free Pascal compiler to allow prefixed attributes - this is needed because traditionally, Free Pascal uses the attribute notation to specify modifiers for procedures and functions. The second directive allows the use of advanced records.

The {`$RTTI` } directive deserves a mention: This directive controls how much RTTI is generated for your classes and records.

```
{$RTTI EXPLICIT FIELDS[vcPublic]}
```

The `EXPLICIT` keyword tells the compiler that what follows is the only kind of information that should be generated, and it should ignore the RTTI settings for parent classes. Alternatively `INHERITED` can be used to indicate that what follows should begeneration in addition to what information is used for the parent class. Finally, the `FIELDS` keyword tells the compiler what kind of information should be generated for fields. The other possibilities are `METHODS` and `PROPERTIES`, which are used to specify what RTTI should be generated for methods and properties, respectively. Finally a list of visibilities must be specified: the visibility sections for which RTTI information must be generated.

The above directive therefore tells the compiler that it should only generate RTTI for public fields. The following directive tells the compiler to generate all possible RTTI information

```
{$RTTI EXPLICIT
```

```
        FIELDS[vcPrivate,vcProtected,vcPublic,vcPublished]
        METHODS[vcPrivate,vcProtected,vcPublic,vcPublished]
        PROPERTIES[vcPrivate,vcProtected,vcPublic,vcPublished]
}
```

The next thing to note is the presence of the `rest.types` `rtitojson.types` and `objtodb.types` units. These units define the various attributes used in the definition. An attribute in Delphi is any class which descends from `TCustomAttribute`.

The attribute notation

```
[JSONKey('first-name')]
```

Tells the compiler that it should construct an attribute of class `JSONKey` or `JSONKeyAttribute`, (the compiler will optionally strip off the `Attribute` from the classname when looking for the class) and that it should pass the string `'first-name'` to the constructor of the class.

The following unit defines the attributes for the JSON streaming:

```
unit rttitojson.types;

{$mode ObjFPC}{$H+}

interface

uses
  Classes, SysUtils;

Type

  { JSONKeyAttribute }

  JSONKeyAttribute = class(TCustomAttribute)
  private
    FKey: string;
  public
    constructor create(aKey : String);
    property Key : string Read FKey;
  end;

  { DateFormatAttribute }

  DateFormatAttribute = class(TCustomAttribute)
    FFormat : String;
  Public
    constructor create(aFormat : String);
    Property DateFormat : String Read FFormat;
  end;
```

The implementation is quite simple:

```
implementation

{ JSONKeyAttribute }
```

```
constructor JSONKeyAttribute.create(aKey: String);
begin
  FKey:=aKey;
end;

{ DateFormatAttribute }

constructor DateFormatAttribute.create(aFormat: String);
begin
  FFormat:=aFormat;
end;

end.
```

The `GetAttributes` call can be used to get all attributes on an identifier. The `GetAttribute` call can be used to get a single attribute, by specifying the class for the attribute.

How can we use this to convert a record to a JSON object ? This is done with a `TRTTIJSONWriter` in a separate unit (we could also have used a class obviously).

Note that the `TPerson` record in no way references the writer: only the attributes are needed to define our `TPerson` record.

```
unit rttitojson.writer;

{$mode ObjFPC}
{$H+}
{$modeswitch advancedrecords}

interface

uses
  Classes, SysUtils, fpJSON, typinfo, rtti, rttitojson.types;

Type
  TRTTIJSONWriter = record
  private
    Ctx : TRTTIContext;
    function DateFieldToJSON(aDate: TDateTime; Fld: TRTTIField): TJSONData;
    function FieldToJSON(aData: Pointer; Fld: TRTTIField): TJSONData;
  Public
    Procedure ToJSON(aData : Pointer; aTypeInfo : PTypeInfo; aJSON : TJSONObject);
    Generic function ToJSON<T>(var aData : T) : TJSONObject;
    class function create : TRTTIJSONWriter; static;
    procedure Free;
  end;
```

The create and free methods create and free a `TRttiContext` which is needed to call the various functions from the `Rtti` unit:

```
class function TRTTIJSONWriter.create: TRTTIJSONWriter;

begin
```

```
  Result.Ctx:=TRttiContext.Create(False);
end;

procedure TRTTIJSONWriter.Free;
begin
  Ctx.Free;
end;
```

The main entry point is the generic `ToJSON` function. In Free Pascal notation:

```
Generic function ToJSON<T>(var aData : T) : TJSONObject;
```

This is actually just a convenience function with a quite simple implementation. It creates the result JSON Object and calls an overloaded version of the `ToJSON` function, passing it the type information of the type T (a record):

```
generic function TRTTIJSONWriter.ToJSON<T>(var aData : T) : TJSONObject;

begin
  Result:=TJSONObject.Create;
  try
    ToJSON(@aData,TypeInfo(T),Result);
  except
    Result.Free;
    Raise;
  end;
end;
```

The actual work is done in the other `ToJSON` function.

In essence this gets the Rtti for the record and loops over all fields: the list of fields is retrieved with the `GetFields` call, which returns an array of `TRttiField` instances. For each field it determines the key name to use when writing the JSON, and then calls `FieldToJSON` to convert the value of the field to a JSON data element, which is then added to the JSON object.

```
procedure TRTTIJSONWriter.ToJSON(aData: Pointer;
                                 aTypeInfo: PTypeInfo;
                                 aJSON: TJSONObject);

Var
  R : TRttiRecordType;
  Fld : TRttiField;
  Key : JSONKeyAttribute;
  KeyName : String;

begin
  R:=Ctx.GetType(aTypeInfo) as TRttiRecordType;
  For Fld in R.GetFields do
    begin
    KeyName:=Fld.Name;
    Key:=JSONKeyAttribute(Fld.GetAttribute(JSONKeyAttribute));
    if Assigned(Key) then
      KeyName:=Key.Key;
    aJSON.Add(KeyName,FieldToJSON(aData,Fld));
```

```
    end;
end;
```

Note the use of the `GetAttribute` call with the `JSONKeyAttribute` class name. If no attribute of this class exists for the field, `Nil` will be returned, and in that case the actual field name is used.

The `FieldToJSON` call starts by getting the value of the field using the `GetValue` name, passing the function a pointer to the record. The `GetValue` function will use the field's RTTI to calculate the correct memory address to retrieve the value:

```
function TRTTIJSONWriter.FieldToJSON(aData: Pointer; Fld : TRTTIField): TJSONData;

var
  V : TValue;
begin
  V:=Fld.GetValue(aData);
  case V.Kind of
    tkFloat : if V.TypeInfo=TypeInfo(TDateTime) then
                Result:=DateFieldToJSON(V.AsDateTime,Fld)
              else
                Result:=TJSONFloatNumber.Create(V.AsDouble);
    tkInt64 : Result:=TJSONInt64Number.Create(V.AsInt64);
    tkInteger : Result:=TJSONIntegerNumber.Create(V.AsInteger);
    tkEnumeration : Result:=TJSONString.Create(GetEnumName(V.TypeInfo,V.AsInteger));
    tkAString,
    tkString : Result:=TJSONString.Create(V.AsString);
    tkWString,
    tkUString : Result:=TJSONString.Create(UTF8Encode(V.AsUnicodeString));
  else
    Raise Exception.Create('Unsupported type');
  end;
end;
```

After the value was retrieved, the type of the value is examined, and an appropriate JSON data structure is created using the data. For a `TDateTime` field, a special routine is called which uses the DateFormat attribute to create a correcty formed JSON string:

```
function TRTTIJSONWriter.DateFieldToJSON(aDate : TDateTime; Fld : TRTTIField) : TJSONData;

var
  Res,Fmt : string;
  DateFormat : DateFormatAttribute;

begin
  Fmt:='';
  DateFormat:=DateFormatAttribute(Fld.GetAttribute(DateFormatAttribute));
  if Assigned(DateFormat) then
    Fmt:=DateFormat.DateFormat;
  if Fmt='' then
    Res:=DateToISO8601(aDate)
  else
    Res:=FormatDateTime(Fmt,aDate);
  Result:=TJSONString.Create(Res);
```

```
end;
```

With this, we're all done. The following program tests our code:

```
uses sysutils, contact, fpjson, rttitojson.types, rttitojson.writer;

Procedure WriteJSON(P :  TPerson);

var
  Writer : TRTTIJSONWriter;
  JSON : TJSONObject;
begin
  JSON:=Nil;
  Writer:=TRTTIJSONWriter.Create;
  try
    JSON:=Writer.ToJSON<TPerson>(P);
    Writeln('JSON : ',JSON.FormatJSON());
  finally
    JSON.Free;
    Writer.Free;
  end;
end;

var Person : TPerson;

begin
  With Person do
    begin
    id:=1;
    FirstName:='Kirth';
    LastName:='Gersen';
    birthdate:=EncodeDate(1486,5,14);
    end;
  WriteJSON(Person);
end.
```

The output is shown in figure 1 on page 12. If there was only one object, the whole operation can of course be done much simpler:

```
function TPerson.ToJSON : TJSONObject;
begin
  Result:=TJSONObject.Create([
    'id',id,
    'first-name',firstname,
    'last-name',LastName,
    'date-of-birth',FormatDateTime('yyyymmdd',birthdate)
  ]);
end;
```

But as remarked earlier, the technique is best suited when dealing with a large number of objects: it avoids having to write a oJSON

Figure 1: The generated JSON