X²XMLDataBinding Documentation

Mark van Renswoude

Jun 16, 2022

TABLE OF CONTENTS

CHAPTER

ONE

INTRODUCTION

X²XMLDataBinding is an alternative to the Delphi built-in XML data binding generator with a few benefits.

The tool has been compiled and tested with Delphi 10.2, your mileage may vary with other versions. It was originally compiled with Delphi XE2 and probably still works on many versions.

The files it generates are most likely compatible with Delphi XE2 and up as well, perhaps even lower versions, let me know if you have experience with them!

1.1 Key features

- Proper namespace support
- Output to a single file or multiple files, allowing types to be reused when included from several root schemas
- Has<Name> properties for optional elements and attributes
- <Name>IsNil properties for xsi:nil elements
- <Name>Def methods to safely read optional values with a default
- Read and write enumeration properties as a typed enum or raw text
- Proper boolean support
- Much improved support for date/time values
- Support for Base64 encoded values
- Basic validation for outputting XML documents which applies element order for sequences and checks for the presence of required elements
- Influence the generator by using a *Hints file*

Note: some features might already be present in the built-in XML data binding generator, but have been buggy in older Delphi versions and some still are.

This tool does not implement the full XSD specification, as some parts are obviously a design by committee. However, feel free to enhance the parsing if you have a particular schema which can not be processed!

CHAPTER

TWO

HINTS FILE

2.1 Introduction

There are a few cases where the generated output can be influenced by providing a hints file. You can generate a skeleton hints file by opening X^{2}XMLDataBinding, selecting a schema file and clicking the "Generate blank hints file" button. The filename must be <XSD filename without extension>.hints.xml to be recognized.

The empty file looks like this:

```
<?xml version="1.0"?>
<DataBindingHints xmlns="http://www.x2software.net/xsd/databinding/DataBindingHints.xsd">
</DataBindingHints>
```

In the sections below each available hint will be described. All hints will require at least two parameters: a Schema and an XPath expression.

The Schema can either be an empty string or the name of the XSD file (without the .xsd extension) which contains the element you want to modify. If the Schema is empty, the root schema (the one you select as the Schema file in the tool) will be used.

The XPath expression identifies the element *as present in the XSD file* on which the hint will be applied.

2.2 Specifying document elements

The data binding will try to determine the document element(s) from the XSD by creating Get/Load/New functions for global elements in the root schema. If the actual element you require is in an included schema, or if you want define a different set of document elements as to not get confused when using the binding, you can use the *DocumentElements* hint to describe those.

For example, this schema with two global elements:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="ChildElement">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Value" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="RootElement">
```

(continues on next page)

(continued from previous page)

```
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="ChildElement"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

This will result in the following functions:

```
{ Document functions }
function GetChildElement(ADocument: XMLIntf.IXMLDocument): IXMLChildElement;
function LoadChildElement(const AFileName: String): IXMLChildElement;
function LoadChildElementFromStream(AStream: TStream): IXMLChildElement;
function LoadChildElementFromString(const AString: String{$IF CompilerVersion >= 20};␣
↪AEncoding: TEncoding = nil; AOwnsEncoding: Boolean = True{$IFEND}): IXMLChildElement;
function NewChildElement: IXMLChildElement;


function GetRootElement(ADocument: XMLIntf.IXMLDocument): IXMLRootElement;
function LoadRootElement(const AFileName: String): IXMLRootElement;
function LoadRootElementFromStream(AStream: TStream): IXMLRootElement;
function LoadRootElementFromString(const AString: String{$IF CompilerVersion >= 20};␣
↪AEncoding: TEncoding = nil; AOwnsEncoding: Boolean = True{$IFEND}): IXMLRootElement;
function NewRootElement: IXMLRootElement;
```

To define only the RootElement as a document element, use the following hint:

```
<?xml version="1.0"?>
<DataBindingHints xmlns="http://www.x2software.net/xsd/databinding/DataBindingHints.xsd">
  <DocumentElements>
    <DocumentElement Schema="" XPath="//xs:element[@name='RootElement']" />
  </DocumentElements>
</DataBindingHints>
```

2.3 Renaming interfaces

The interface names are generated based on various factors, like the name of the complex type or the element, and will be made unique if required by prefixing them with the parent element's name and/or adding a number. This can result in some unexpected names. For example, using this XSD:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Order">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Info" type="Info"/>
        <xs:element name="Customer">
          <xs:complexType>
            <xs:sequence>
```

(continues on next page)

(continued from previous page)

```
            <xs:element name="Info">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="Name" type="xs:string"/>
                  <xs:element name="DateOfBirth" type="xs:date"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="Salesperson">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Info">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="Name" type="xs:string"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:complexType name="Info">
  <xs:sequence>
    <xs:element name="OrderNumber" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
</xs:schema>
```

When generated, due to the order in which the interfaces are resolved, the interface for Order.Info will be named 'IXMLInfo3', whereas the other two will be aptly named 'IXMLCustomerInfo' and 'IXMLSalespersonInfo'.

This is not an issue if you directly assign the Order.Info property, but if you need a variable or parameter of that type it is not immediately clear from your code which Info element it corresponds to.

To change this you can use an InterfaceName hint:

```
<?xml version="1.0"?>
<DataBindingHints xmlns="http://www.x2software.net/xsd/databinding/DataBindingHints.xsd">
  <Interfaces>
    <InterfaceName Schema="" XPath="//xs:complexType[@name='Info']">OrderInfo</
↪InterfaceName>
  </Interfaces>
</DataBindingHints>
```

This will result in the interface to be named IXMLOrderInfo instead.

2.4 Renaming properties

Much like renaming interfaces you can also rename properties. This is useful if you don't like the underscore added for reserved words, or if the XML you are trying to parse or write has confusing element names. For example, we had a supplier who 'minified' the XML by naming all elements "V1" through "V185".

Example XSD:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Order">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="V1" type="xs:dateTime"/>
        <xs:element name="V2" type="xs:string"/>
        <xs:element name="V3" type="xs:string"/>
        <xs:element name="V4" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Example hints file:

```
<?xml version="1.0"?>
<DataBindingHints xmlns="http://www.x2software.net/xsd/databinding/DataBindingHints.xsd">
  <Properties>
    <PropertyName Schema="" XPath="//xs:element[@name='V1']">CreationDate</PropertyName>
    <PropertyName Schema="" XPath="//xs:element[@name='V2']">CreatedBy</PropertyName>
    <PropertyName Schema="" XPath="//xs:element[@name='V3']">Currency</PropertyName>
    <PropertyName Schema="" XPath="//xs:element[@name='V4']">Market</PropertyName>
  </Properties>
</DataBindingHints>
```

As you can see from the generated file, the elements will still be read as 'V1', 'V2', etc. but you can use the self-explanatory name of CreationDate to access that element in code.

```
IXMLOrder = interface(IXMLNode)

  ...

  property CreationDate: TDateTime read GetCreationDate write SetCreationDate;
  property CreatedBy: WideString read GetCreatedBy write SetCreatedBy;
  property Currency: WideString read GetCurrency write SetCurrency;
  property Market: WideString read GetMarket write SetMarket;
end;

...

function TXMLOrder.GetCreationDate: TDateTime;
begin
  Result := XMLToDateTime(ChildNodes['V1'].NodeValue, xdtDateTime);
end;
```

(continues on next page)

(continued from previous page)

```
function TXMLOrder.GetCreatedBy: WideString;
begin
  Result := ChildNodes['V2'].Text;
end;
```

2.5 Renaming enumeration values

There are two main use cases for renaming enumeration values:

1. The element is defined as an enumeration but the values are hard to read in code or even generate invalid results
2. The element is defined as a string by the supplier of the XSD but is an enumeration for all intents and purposes and you want to use it as such in code

This example XSD contains both use cases:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Rule">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Condition" type="Condition"/>
        <xs:element name="Field" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:simpleType name="Condition">
    <xs:restriction base="xs:string">
      <xs:enumeration value="="/>
      <xs:enumeration value="&lt;"/>
      <xs:enumeration value="GreaterThan"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

For the 'Condition' the three enum values will all be generated as *Condition_* as the data binding can not handle these special characters, and the result will not even compile.

The 'Field' may be a known set of values which you want to treat as an enum for easy case statements. In this case we can define enumeration hints as well. Note that if the value in an XML is not one of the defined enums, you can still read it's value by using the *FieldText* property which is generated in addition to the typed *Field* property.

```
<?xml version="1.0"?>
<DataBindingHints xmlns="http://www.x2software.net/xsd/databinding/DataBindingHints.xsd">
  <Enumerations>
    <Enumeration Schema="" XPath="//xs:simpleType[@name='Condition']">
      <Member Name="=">Condition_Equals</Member>
      <Member Name="&lt;">Condition_LessThan</Member>
    </Enumeration>
    <Enumeration Schema="" XPath="//xs:element[@name='Field']" ReplaceMembers="true">
      <Member Name="CustomerName">Field_CustomerName</Member>
      <Member Name="TotalAmount">Field_TotalAmount</Member>
```

(continues on next page)

(continued from previous page)

```
    </Enumeration>
  </Enumerations>
</DataBindingHints>
```

At the time of writing, renaming enumeration members only works for simple types, which is why it was defined separately in the example above and referenced from the element.

The ReplaceMembers attribute determines if the provided set of Members is to be considered the full new set (true) or a translation of specific members (false). In the example above, if ReplaceMembers was set to true for the Condition, the 'GreaterThan' member would be left out of the generated binding.