Liquid XML Data Binder 2018

Getting Started

Contents

CHAPTER 1
Introduction

This is a walkthrough tutorial that takes you through the process of developing an XML Data Binding Component from an XML Schema (XSD).

It is assumed that you have a working knowledge of XSD and either C#, VB .Net, C++, Java or Visual Basic 6.

The concepts covered are:

- Designing an XML Schema (XSD)
- Running the Wizard
- Editing the Schema to Object Mapping
- Generating the XML Data Binding Component
- Writing a Client
- Deploying the Component and Runtime Library

Getting Further Assistance

Further product details can be found in the documents and help files included with the Wizard installation. Latest information can be found on the Support pages of our web-site:

https://www.liquid-technologies.com/support

LTHelp.chm

This is a standard Windows Help File that provides help on using the Wizard and provides detailed code examples in C#, VB .Net, C++, Java and Visual Basic 6 for dealing with XML Schema concepts including:

Cardinality

This sample shows how to deal with mandatory, optional and collections of elements.

Derived By Extension

This sample shows how a complexType may be extended, and how the base and extended types can be manipulated in code.

Simple All

This sample shows an element containing an 'all' group of simple elements. All the elements must appear, but in any order.

Simple Choice

This sample shows an element containing a choice of simple elements. One and only one child element may be provided.

Simple Hierarchy

This sample shows a number of elements within a Hierarchy. It includes local (anonymous) and global complexType's. It also shows how collections are manipulated within sequences and choices.

Simple Sequence

This sample shows an element containing a sequence of simple elements and attributes.

Your Library Generated Documentation

The best place to find help about writing client code specific to accessing your generated library component is in the generated documentation. This can be compiled into a HTML Help

file by using the generated build.bat file (and ensuring you have Microsoft HTMLHelp 1.3 installed on your PC).

This documentation shows the complete interface to your library, as well as the standard interface provided by the Liquid XML 2018 Runtime classes specific to your chosen language.

CHAPTER 2

Designing an XML Schema (XSD)

The first thing we need to do is create an XML Schema. The wizard extracts the information from the XML Schema which then forms the basis of the rules required by the XML code generator. In simple terms, each XML Schema defined type is mapped to a class with elements and attributes mapped to properties of the class. Relationships between types are also modelled in the generated classes, taking into account cardinality (0..1, 1..1, 0..n, 1..n), as well as validation rules for data type bounds checking.

The Wizard supports XSD, XDR and DTD. The examples we use are written in XSD as it is the W3C Standard.

This is not a tutorial on XSD. It is an overview of how your XSD design affects the generated code within your XML Data Binding Component.

Sample XSD

The XSD that is provided as an example is the BookStore.xsd.

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
    <xsd:element name="bookstore" type="bookstoreType" />
    <xsd:complexType name="bookstoreType">
        <xsd:sequence minOccurs="0" maxOccurs="unbounded">
            <xsd:element name="book" type="bookType" />
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="bookType">
        <xsd:sequence>
            <xsd:element name="title" type="xsd:string" />
            <xsd:element name="author" type="authorName" />
            <xsd:element name="genre" type="xsd:string" minOccurs="0" />
        </xsd:sequence>
        <xsd:attribute name="price" type="xsd:double" use="required" />
        <xsd:attribute name="publicationdate" type="xsd:date" />
        <xsd:attribute name="ISBN" type="xsd:string" />
    </xsd:complexType>
    <xsd:complexType name="authorName">
        <xsd:sequence>
            <xsd:element name="first-name" type="xsd:string" />
            <xsd:element name="last-name" type="xsd:string" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

You can find example client code that accesses code generated from this schema in the '[Install Dir]\XmlDataBinder16\Examples\BookStoreLib' directory. If you generate code from this xsd you will see that the following classes are generated, that are specific to this schema:

AuthorName

Bookstore

BookType

BookTypeCol

Note: When generating for C# or VB .Net, BookTypeCol is not generated as Templated collections are used instead.

E.g. XmlObjectCollection<BookType>

Design Concepts

When developing the XSD we have to ensure that as well as been syntactically correct (i.e. valid against the W3C XML Schema standard) it also must make sense from a design perspective.

There are a number of key entities that we can create in our generated components

1. Classes
2. Attributes or Properties
3. Collections
4. Enumerations

Classes

Each complexType and element defined in the schema causes a class to be generated. There are also a number of optimisations in place to reduce the number of classes created.

A complexType or element is mapped to a class, its child elements and attributes become its properties in the case of VB6, C# and VB .Net, and accessors and mutators (getters and setters) in the case of java & C++.

For simple schemas, the mapping between the schema and the generated code is quite straight forward. Sometimes names need to be changed in order to avoid reserved words or duplicates.

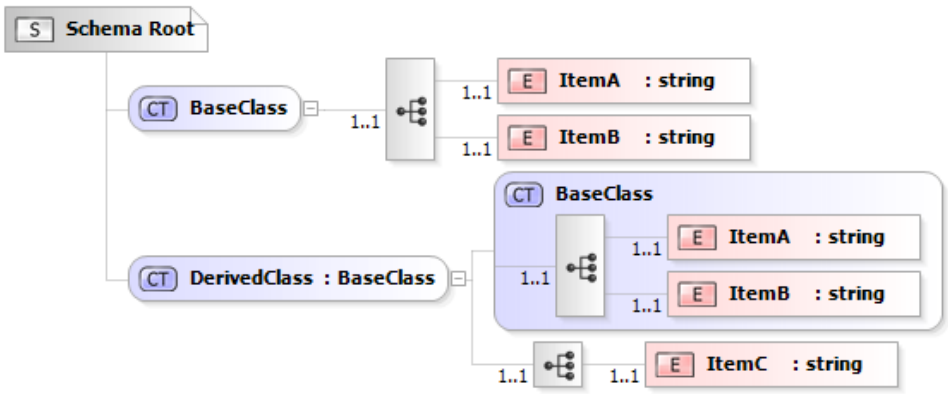
However for more complex schemas things become a little more interesting. There are 2 main schema concepts that have a bearing on the generated classes, ‘extension/restriction’ and ‘substitutionGroup’.

Elements created by extension or restriction

Elements can be created by specifying another complexType as a base. This has the effect of allowing additional child elements and attributes to be added to the one being extended (see xsd reference).

In C++, java, C# and VB .Net this is implemented using interface inheritance. An abstract base class (or interface class) is created holding all the attributes and elements of the base complexType. A concrete instance is then created for the complexType base and the new derived type.

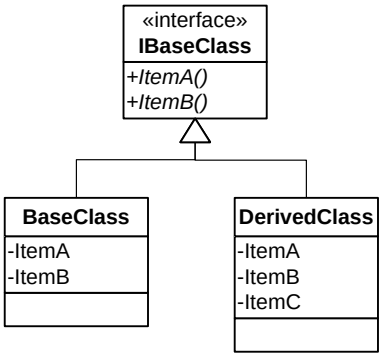
So given the schema:




```
<xs:complexType name="BaseClass">
   <xs:sequence>
      <xs:element name="ItemA" type="xs:string"/>
      <xs:element name="ItemB" type="xs:string"/>
   </xs:sequence>
</xs:complexType>
<xs:complexType name="DerivedClass">
   <xs:complexContent>
      <xs:extension base="BaseClass">
         <xs:sequence>
```

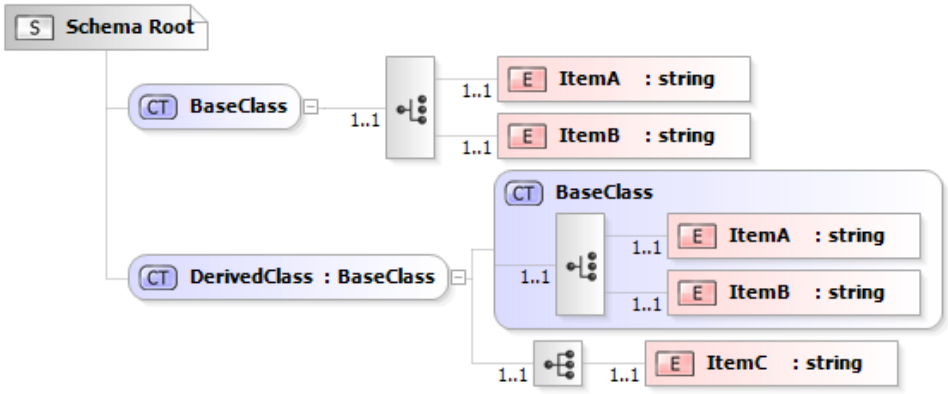
```
            <xs:element name="ItemC" type="xs:string"/>
         </xs:sequence>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>
```

You would get the following classes generated:



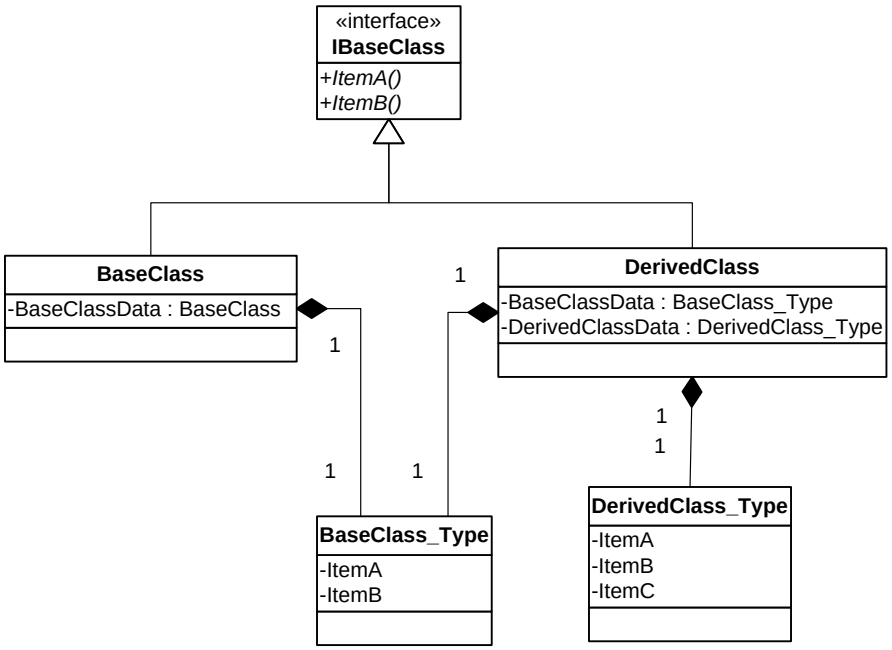
There are however a number of added complications. If the model of the base and derived types are different, e.g. in the case of ‘sequence’, ‘choice’ or ‘all’, then the generated code becomes a little more complex.

For example, if we now defined the BaseClass as having a Choice (ItemA or ItemB), and keep the DerivedClass as adding a Sequence (ItemC):



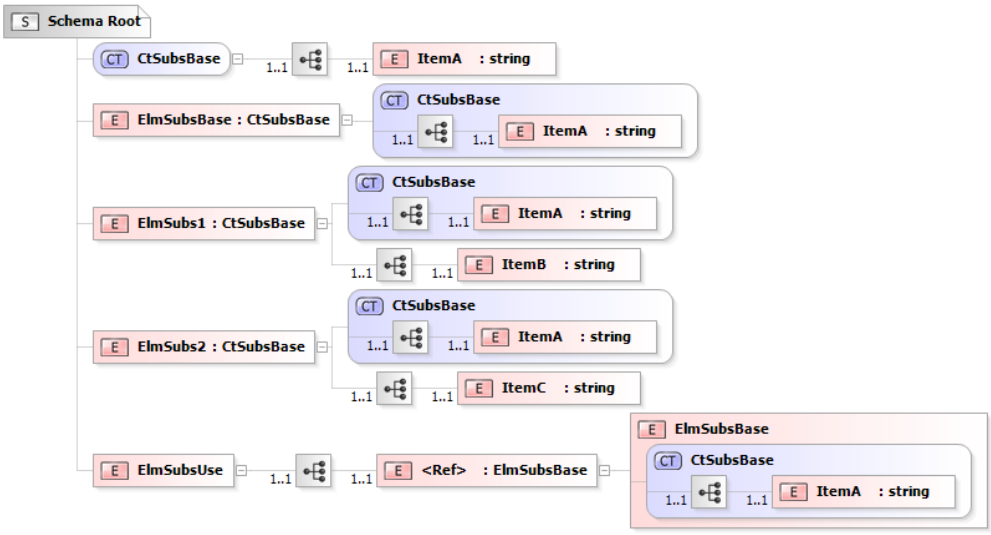

```
<xs:complexType name="BaseClass">
   <xs:choice>
      <xs:element name="ItemA" type="xs:string"/>
      <xs:element name="ItemB" type="xs:string"/>
   </xs:choice>
</xs:complexType>
<xs:complexType name="DerivedClass">
   <xs:complexContent>
      <xs:extension base="BaseClass">
         <xs:sequence>
            <xs:element name="ItemC" type="xs:string"/>
         </xs:sequence>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>
```

The generated code now includes two additional classes to allow the all XML data possibilities to be represented:



Elements defined as substitutionGroups

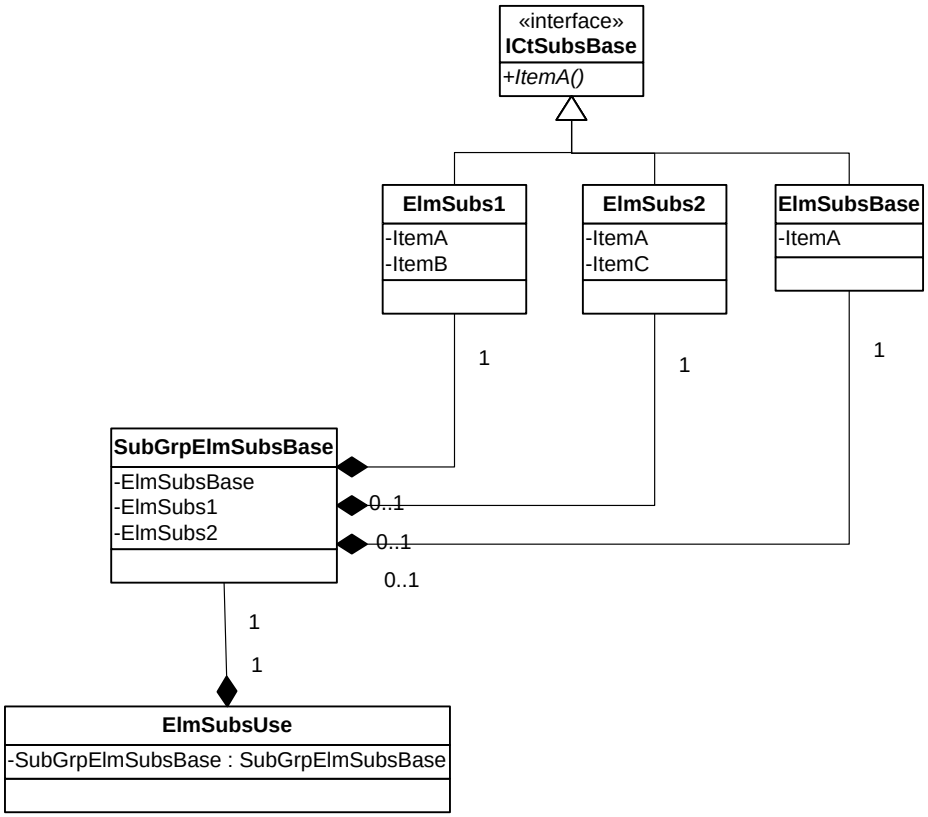
An element can define itself as a possible substitution for another element. This allows the element to be used in place of the other element. In the example below, ElmSubs1 or ElmSubs2 may be used anywhere that ElmSubsBase is specified.




```
<xs:complexType name="CtSubsBase">
   <xs:sequence>
      <xs:element name="ItemA" type="xs:string"/>
   </xs:sequence>
</xs:complexType>
<xs:element name="ElmSubsBase" type="CtSubsBase"/>
<xs:element name="ElmSubs1" substitutionGroup="ElmSubsBase">
   <xs:complexType>
      <xs:complexContent>
         <xs:extension base="CtSubsBase">
            <xs:sequence>
               <xs:element name="ItemB" type="xs:string"/>
            </xs:sequence>
         </xs:extension>
      </xs:complexContent>
   </xs:complexType>
</xs:element>
<xs:element name="ElmSubs2" substitutionGroup="ElmSubsBase">
   <xs:complexType>
      <xs:complexContent>
```

```
            <xs:extension base="CtSubsBase">
               <xs:sequence>
                  <xs:element name="ItemC" type="xs:string"/>
               </xs:sequence>
            </xs:extension>
         </xs:complexContent>
      </xs:complexType>
   </xs:element>
   <xs:element name="ElmSubsUse">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="ElmSubsBase"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
```

The code generator deals with this by creating a SubGrpElmSubsBase class wherever ElmSubsBase is specified. The SubGrpElmSubsBase class is created as a choice, which has all the possible substitutions contained within it (in this case ElmSubsBase, ElmSubs1 and ElmSubs2). The base class, ICtSubsBase, can normally be ignored.



Attributes

Each element or attribute that we define will be mapped to an attribute in the appropriate generated class.

Collections

Each association that we define will be mapped to a Collection Class in our generated component.

Enumerations

Each Enumeration that we define will be mapped to an Enum in our generated component.

Namespaces

Within an XSD schema, namespaces can be used to segment your schemas and to separate off reusable entities. The XML created by the wrapper classes qualifies the entities within the XML document. This will mean that valid XML is created, but this XML may be verbosely qualified with namespace declarations. By default only namespaces aliased in the XSD are explicitly declared in the generated XML document, the rest are aliased as they appear and given arbitrary aliases (AA, AB, AC etc). This behaviour although it produces valid XML is not always ideal.

The class XmlSerializationContext controls how namespace's are dealt with, when the ToXml methods are called. The settings can be changed in 2 ways.

- There is a global instance of the XmlSerializationContext class accessible via XmlSerializationContext::Default. This instance is used if an instance of an XmlSerializationContext class is not explicitly passed to the ToXML methods, thus changes to this are considered global.
- An instance of the XmlSerializationContext class can be created and explicitly passed to the ToXML methods. This allows for a finer level of control, if you need it to behave one way some of the time, and differently in other circumstances, then you can define two XmlSerializationContext objects, and pass them to the ToXML methods accordingly.

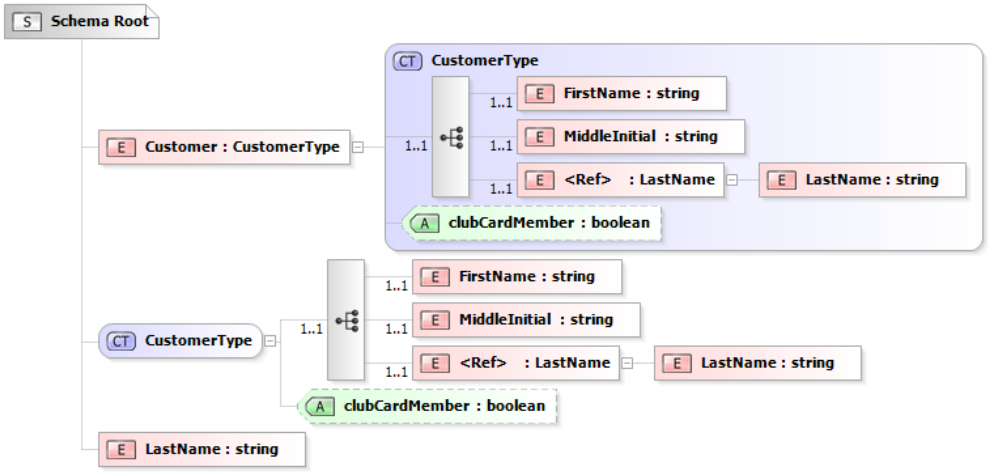
The XmlSerializationContext object allows you to 'pre-register' namespace's. This allows you to define the aliases used for them, and to define a default namespace if required.

If you want these changes to be global then you can place changes within the generated code, the changes will be persisted even if the code is re-generated.

- For C++ - Enumerations.cpp - CAppLifetime::RegisterLibrary method, inside the ##HAND_CODED_BLOCK
- For C# - Enumerations.cs - Registration.RegisterLicense method, inside the ##HAND_CODED_BLOCK
- For VB .Net - Enumerations.vb - Registration.RegisterLicense method, inside the ##HAND_CODED_BLOCK
- For Java - Registration.java - Registration.registerLicense method, inside the ##HAND_CODED_BLOCK
- For VB6 - General.bas - CF method, inside the ##HAND_CODED_BLOCK

An Example.

Assuming we were using the following XSD:




```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://sample" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns="http://sample" elementFormDefault="qualified"
attributeFormDefault="unqualified">
    <xs:element name="Customer" type="CustomerType" />
    <xs:complexType name="CustomerType">
        <xs:sequence>
            <xs:element name="FirstName" type="xs:string" />
            <xs:element name="MiddleInitial" type="xs:string" />
            <xs:element ref="LastName" />
        </xs:sequence>
        <xs:attribute name="clubCardMember" type="xs:boolean" />
    </xs:complexType>
    <xs:element name="LastName" type="xs:string" />
</xs:schema>
```

This will cause the following XML to be produced:

```
<?xml version="1.0"?>
<AA:Customer xmlns:AA="http://sample" xmlns:xs="http://www.w3.org/2001/XMLSchema-
instance" clubCardMember="false">
    <AA:FirstName>Joe</AA:FirstName>
    <AA:MiddleInitial>J</AA:MiddleInitial>
    <AA:LastName>Bloggs</AA:LastName>
</AA:Customer>
```

As you can see the document is qualified using the alias ‘AA’. By changing the entry in the NamespaceAliases collection you can control the alias used, in this case allowing ‘http://sample’ to be aliased using ‘MyAlias’ in the document element.

```
// ##HAND_CODED_BLOCK_START ID="Default Namespace Declarations"## DO NOT MODIFY …
   LtXmlLib16::CXmlSerializationContext::Default.GetNamespaceAliases().Add(
      _T("xs"),
      _T("http://www.w3.org/2001/XMLSchema-instance"));

   LtXmlLib16::CXmlSerializationContext::Default.GetNamespaceAliases().Add(
      _T("MyAlias"),
      _T("http://sample"));

// ##HAND_CODED_BLOCK_END ID="Default Namespace Declarations"## DO NOT MODIFY ANYTHING
OUTSIDE OF THESE TAGS
```

The resulting document will look like this:

```
<?xml version="1.0"?>
<MyAlias:Customer xmlns:MyAlias="http://sample"
xmlns:xs="http://www.w3.org/2001/XMLSchema-instance" clubCardMember="false">
    <MyAlias:First Name>Joe
    </MyAlias:FirstName>
```

```
    <MyAlias:MiddleInitial>J</MyAlias:MiddleInitial>
    <MyAlias:LastName>Bloggs</MyAlias:LastName>
</MyAlias:Customer>
```

The need for namespace qualification can be removed by defining a default namespace, see below.

```
// ##HAND_CODED_BLOCK_START ID="Default Namespace Declarations"## DO NOT MODIFY ...
    LtXmlLib16::CXmlSerializationContext::Default.GetNamespaceAliases().Add(
          _T("xs"),
          _T("http://www.w3.org/2001/XMLSchema-instance"));

    LtXmlLib16::CXmlSerializationContext::Default.SetDefaultNamespaceURI(
          _T("http://sample"));

// ##HAND_CODED_BLOCK_END ID="Default Namespace Declarations"## DO NOT MODIFY ANYTHING
OUTSIDE OF THESE TAGS
```

This gives the following XML

```
<?xml version="1.0"?>
<Customer xmlns="http://sample" xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
clubCardMember="false">
    <FirstName>Joe</FirstName>
    <MiddleInitial>J</MiddleInitial>
    <LastName>Bloggs</LastName>
</Customer>
```

Warning: Declaring a default namespace with some schemas, may cause the XML created to be invalid.

Common Problems

Infinite Recursion

Some Schema designs produce validation code that can never be satisfied. A good example of this is an infinitely recursive schema.

No Elements

If you do not specify any elements in your schema, no classes will be generated as the optimiser will determine that any complexTypes will never be used.

CHAPTER 3
Running the Wizard

Once we have designed our XSD we can run the wizard and generate our component library in our chosen language.

The Wizard

The Wizard can be installed on Windows Vista or later. Please ensure you have the latest service packs installed.

Once installed it can be run from:

Start->Programs->Liquid Studio 2018->Liquid XML Data Binder 2018

Select Schema File

This screen allows you to select the XSD, XDR, DTD or WSDL Schema file that describes the objects and properties used within your business process.

The first screen allows you to select your schema. This can be an XSD, XDR, DTD or WSDL document. If your schema includes other files, these must have fully qualified paths within the schema (or in the same directory).

Select Schema Type

This screen allows you to verify the 'type' of your Schema.

The XML Data Binding Wizard uses the file extension to attempt to select an initial value of XSD, XDR, DTD or WSDL.

XDR Settings

If your schema is an XDR document, you will be presented with the XDR Settings screen. This enables you to set the values to 'Allow Additional Attributes in XDR Schema' and 'Ignore Simple Elements'. Both values default to 'false'.

WSDL Settings

If your schema is a WSDL document, you have the option to 'Generate a Web Client Interface'. If selected, you will be presented with a screen asking you to select the service port from your web service to generate client code.

Select Output Language

This screen allows you to select the output language and namespace for your generated code.

C++

The C++ option generates project files for the following platforms:

- Microsoft Visual C++
- GNU g++

You have the option to change the 'Default Class Namespace'. This value represents both the name of the generated project and the namespace that all classes are generated within.

C# .Net

The C# .Net option allows you to select Development Environment from Visual Studio 2005 to Visual Studio 2017. Within the selected environment, you may then select your Target .Net Platform. This includes all .Net Frameworks from .Net 2.0 to .Net4.7, .Net Core 1.1 and 2.0, .Net Starndard 1.6 and 2.0, Portable Class Library (PCL), Silverlight 3, 4 and 5, Xamarin.Android and Xamarin.iOS.

You have the option to change the 'Default Class Namespace'. This value represents both the name of the generated project and the namespace that all classes are generated within. You

can also specify a ‘Base Package’ which is typically the package path your company uses e.g. ‘com.mycompany’.

Java

Java supports the Java 2 SDK 1.5 and above and generates an ANT build file.

You have the option to change the ‘Default Package Name. This value represents both the name of the generated project and the package that all classes are generated within. You can also specify a ‘Base Package’ which is typically the package path your company uses e.g. ‘com.mycompany’.

Visual Basic .Net

The VB .Net option allows you to select Development Environment from Visual Studio 2005 to Visual Studio 2017. Within the selected environment, you may then select your Target .Net Platform. This includes all .Net Frameworks from .Net 2.0 to .Net4.7, .Net Core 1.1 and 2.0, .Net Starndard 1.6 and 2.0, Portable Class Library (PCL).

You have the option to change the ‘Default Class Namespace’. This value represents both the name of the generated project and the namespace that all classes are generated within. You can also specify a ‘Base Package’ which is typically the package path your company uses e.g. ‘com.mycompany’.

Visual Basic 6 (ActiveX DLL)

Visual Basic 6.0 generates an ActiveX DLL for COM support.

You have the option to change the ‘Component Name’. This value represents the name of the generated project.

NOTE: Due to the COM restriction limiting each class ProgID to 39 chars, the maximum length of the ‘Component Name’ is 15 characters

Enter Output Directory

This screen allows you to select the output directory for your generated code.

Create Documentation

You also have the option to generate HTML documentation for your XML Data Binding library by ensuring the 'Create Documentation' checkbox is checked.

CHAPTER 4

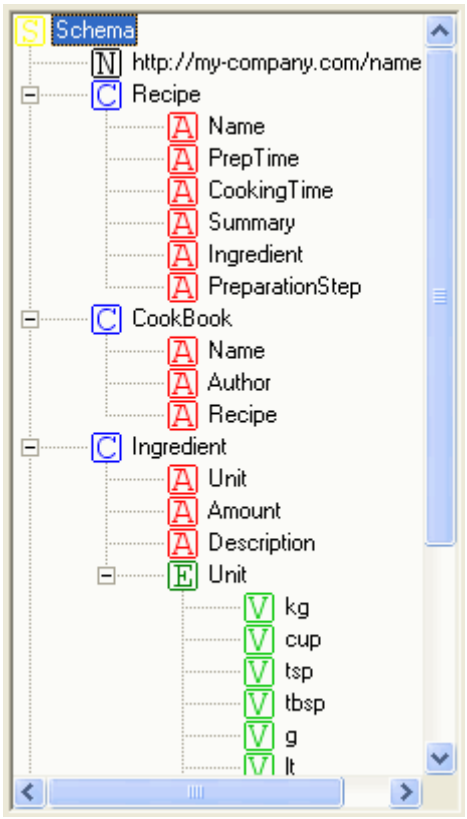
Editing the Schema to Object Mappings

Schema that you write yourself may produce code that suits your needs without making any changes. However for more complex schema or schema that is out of your control you may wish to change the names of the classes, member variables and methods that are generated as part of your XML Data Binding Component.

This is achieved by editing the Schema to Object mappings using the 'Change Schema to Object Mappings' button on the 'Select Output Options' screen.

Liquid Schema to Object Mapper

A representation of the XML Schema is displayed in a Tree.



Selecting an entry displays its associated properties.

Schema Properties

This represents the Schema.

Language (Read Only) Currently selected language.

Namespace Prefix (Read Only)

xmlns (Read Only)

Namespace Properties

This represents the Schema's Namespace.

Schema Namespace (Read Only)

Schema Alias (Read Only)

Class Namespace

Class Properties

This represents a Complex Type within the Schema.

Name (Read Only)	The name created from the schema to identify the class before illegal names and chars are removed.
Long Name (Read Only)	The full name created from the schema to identify the class before shortening, illegal names and chars are removed.
Class Name	The code legal name for the class. This is the name of the class as it will appear in the generated code.
Collection Name	The code legal name for the Collection of classes. This is the name of the class as it will appear in the generated code.
Description	The description placed into the code against the class. ASCII text.
Remarks	The remarks placed into the code against the class. ASCII text.

Attribute Properties

This represents an Element or Attribute within the Schema.

Name (Read Only)	The name of the attribute from the schema. It may have been altered from the schema to avoid duplicates.
Default Value	The default value for the attribute. Must be formatted correctly (given its type) as it would appear in a valid XML document.
IsValid Name	The name of the Is Valid method on the parent class. The IsValid method is generated to determine if a given attribute is present within the schema (See optional parameters).
SetValid Name	The name of the method used to create or remove and optional entity from the parent class. The Set valid method is generated to allow optional parameters (elements & attributes) to be added and removed from the parent class.
IsValid Member	The name of the member variable used to determine if an optional attribute or element is present.
Member Name	The member variable used to hold the data for this attribute.
Display Name	The name used to represent the selected element if the parent class is a choice.
Property Name (C#, VB .Net and VB6)	The name of the property used to expose this attribute in the parent class.
Setter Name (C++ and Java)	The name of the method used to set this attribute in the parent class.
Getter Name (C++ and Java)	The name of the method used to get the value of this attribute in the parent class.
Description	The description placed into the code against the attribute. ASCII text.

Remarks	The remarks placed into the code against the attribute. ASCII text.

Enum Properties

This represents an Enumeration within the Schema.

Name (Read Only)	The name of the enumeration from the schema. It may be different to the name in the schema (to avoid duplicates).
Enum Name	The name of the enumeration (or enumeration class in java).
Enum Collection Name	The name of the class that will hold a collection of these enumerations.

Enum Value Properties

This represents an Enumeration value within the Schema.

Value (Read Only)	The name of the enumeration value from the schema.
Mapped Value	The name enumeration as it will appear in the generated code. It may have been changed to avoid duplication, invalid names, and invalid chars.

CHAPTER 5

Generating the Object Library

Pressing the ‘Generate’ button on the ‘Code Generator’ screen will start the generation process.

Running the Generator

On pressing ‘Generate’ the Schema is validated and then the generation process begins. Each file that is been processed is displayed in the Output Window. If the ‘Show Detailed Log’ checkbox is checked, you will continue to receive a more detailed log.

The generator will inform you when it has finished. During the generation process, you can press the ‘Stop’ button at any time.

Hand Coded Blocks

Once you have generated your library, you will be able to load the generated project file into the relevant development environment. The generated files contain special markers ‘hand coded blocks’ that provide areas that will not be overwritten the next time you generate.

This gives you flexibility to add proprietary code in the knowledge that it will remain intact.

CHAPTER 6

Writing and Deploying a Client

Once you have created an XML Data Binding Component, you will need to write Client code to make use of it.

Development Language Choice

The language you choose for your client does not necessarily have to be the same as the language generated by the Wizard.

For example if you have generated a C# component, you can make use of its functionality from any .Net compliant language such as VB.Net or F#. Similarly, if you generated a Visual Basic 6 ActiveX COM component, you can write your client in Delphi, ATL or any COM compliant language.

You may also choose to generate a C++ component and wrap it in .Net or ATL.

Client Code Basics

Here are some code snippets that show how you can reference objects that are exposed from your generated component. Full client code examples that use the PriceEnquirySample.xsd are provided as part of the product installation.

The following examples assume you have generated a component ‘MyLib’ that has two classes ‘MyClass’ and ‘OtherClass’ which have a one to many relationship.

C#

```
try
{
   MyLib.MyClass myClass = new MyLib.MyClass();

   // the xml can be read into our object model from a file using FromXmlFile
   myClass.FromXmlFile("MyFile.xml");

   // properties can be read from and written to the object model
   myClass.FirstName = "Fred";
   myClass.LastName = "Smith";

   // one:many relationships can be created between classes using collections
   MyLib.OtherClass otherClass = new MyLib.OtherClass()
   otherClass.Description = "Some Information";
   myClass.OtherClass.Add(otherClass);

   // the entire object model can be converted to an xml string using ToXml()
   Console.WriteLine(myClass.ToXml());

   // or written back to a file using ToXmlFile
   myClass.ToXmlFile("MyOtherFile.xml");
}
// All exceptions are derived from LtXmlLib16.LtException
catch (LiquidTechnologies.LtXmlLib16.LtException ex)
{
   string errText = "Error - \n";
   // Note: exceptions are likely to contain inner exceptions
   // that provide further detail about the error.
   while (ex != null)
   {
      errText += ex.Message + "\n";
      ex = ex.InnerException;
   }
   Console.WriteLine(errText);
}
```

C++

```
try
{
   // classes are referenced through smart pointers, so they do not need deleting
   MyLib::CMyClassPtr spMyClass = MyLib::CMyClass::CreateInstance();

   // the xml can be read into our object model from a file using FromXmlFile
   spMyClass->FromXmlFile("MyFile.xml");

   // properties can be read from and written to the object model
   spMyClass->SetFirstName("Fred");
   spMyClass->SetLastName("Smith");

   // one:many relationships can be created between classes using collections
   MyLib::COtherClassPtr spOtherClass = spMyClass->GetOtherClass()->Add();
   spOtherClass->SetDescription("Some Information");

   // the entire object model can be converted to an xml string using ToXml()
   printf("%s", spMyClass->ToXml().c_str());

   // or written back to a file using ToXmlFile
   spMyClass->ToXmlFile("MyOtherFile.xml");
}
// All exceptions are derived from LtXmlLib16::CLtException and should be caught by
// reference.
catch (LtXmlLib16::CLtException &e)
{
   printf("Error\n%s\n", e.GetFullMessage().c_str());
}
```

Java

```
try {
   MyLib.MyClass myClass = new MyLib.MyClass();

   // the xml can be read into our object model from a file using FromXmlFile
   myClass.fromXmlFile("MyFile.xml");

   // properties can be read from and written to the object model
   myClass.setFirstName("Fred");
   myClass.setLastName("Smith");

   // one:many relationships can be created between classes using collections
   MyLib.OtherClass otherClass = myClass.getOtherClass().add();
   otherClass.setDescription("Some Information");

   // the entire object model can be converted to an xml string using ToXml()
   System.out.println(myClass.ToXml());

   // or written back to a file using ToXmlFile
   // NOTE: java components can optionally specify an end of line type (LF or CRLF)
   myClass.toXmlFile("MyOtherFile.xml", EOLType.LF);
} // All exceptions are derived from LtXmlLib16.exceptions.LtException
catch (com.liquid_technologies.LtXmlLib16.exceptions.LtException ex) {
   System.out.println(ex.getMessage());
}
```

Visual Basic 6

```
On Error GoTo Failed:

   Dim o MyClass As New MyLib.MyClass

   // the xml can be read into our object model from a file using FromXmlFile
   myClass.FromXmlFile "MyFile.xml"

   // properties can be read from and written to the object model
   myClass.FirstName = "Fred"
   myClass.LastName = "Smith"

   // one:many relationships can be created between classes using collections
   Dim otherClass As MyLib.OtherClass
   Set otherClass = myClass.OtherClass.AddNew
```

```
    otherClass.Description = "Some Information"

    // the entire object model can be converted to an xml string using ToXml()
    MsgBox myClass.ToXml()

    // or written back to a file using ToXmlFile
    myClass.ToXmlFile "MyOtherFile.xml"
    Exit Sub

Failed:
     MsgBox Err.Description
```

Runtime Libraries

In order to run your client and deploy it to your users you will need to distribute the appropriate Runtime Library.

All redistributable files can be found in the '[Install Dir]\XmlDataBinder16\Redist16' folder. The files that you must distribute are dependent on the language of your generated component.

C# and VB .Net

.Net 2.0 (C# and VB .Net)

The redistributable file **LiquidTechnologies.Runtime.Net20.dll** can be found in the 'Redist16\DotNet' sub folder. This is a .Net Component with the following Namespace:

```
LiquidTechnologies.Runtime.Net20
```

Your generated component, and possibly your client, should reference this file.

.Net 3.0 (C# and VB .Net)

The redistributable file **LiquidTechnologies.Runtime.Net30.dll** can be found in the 'Redist16\DotNet' sub folder. This is a .Net Component with the following Namespace:

```
LiquidTechnologies.Runtime.Net30
```

Your generated component, and possibly your client, should reference this file.

.Net 3.5 (C# and VB .Net)

The redistributable file **LiquidTechnologies.Runtime.Net35.dll** can be found in the 'Redist16\DotNet' sub folder. This is a .Net Component with the following Namespace:

```
LiquidTechnologies.Runtime.Net35
```

Your generated component, and possibly your client, should reference this file.

.Net 4.0 (C# and VB .Net)

The redistributable file **LiquidTechnologies.Runtime.Net40.dll** can be found in the 'Redist16\DotNet' sub folder. This is a .Net Component with the following Namespace:

```
LiquidTechnologies.Runtime.Net40
```

Your generated component, and possibly your client, should reference this file.

.Net 4.5 (C# and VB .Net)

The redistributable file **LiquidTechnologies.Runtime.Net45.dll** can be found in the 'Redist16\DotNet' sub folder. This is a .Net Component with the following Namespace:

```
LiquidTechnologies.Runtime.Net45
```

Your generated component, and possibly your client, should reference this file.

Silverlight 3 (C#)

The redistributable file **LiquidTechnologies.Runtime.Silverlight30.dll** can be found in the 'Redist16\DotNet' sub folder. This is a .Net Component with the following Namespace:

```
LiquidTechnologies.Runtime.Silverlight30
```

Your generated component, and possibly your client, should reference this file.

Silverlight 4 (C#)

The redistributable file **LiquidTechnologies.Runtime.Silverlight40.dll** can be found in the 'Redist16\DotNet' sub folder. This is a .Net Component with the following Namespace:

```
LiquidTechnologies.Runtime.Silverlight40
```

Your generated component, and possibly your client, should reference this file.

Silverlight 5 (C#)

The redistributable file **LiquidTechnologies.Runtime.Silverlight50.dll** can be found in the 'Redist16\DotNet' sub folder. This is a .Net Component with the following Namespace:

```
LiquidTechnologies.Runtime.Silverlight50
```

Your generated component, and possibly your client, should reference this file.

Xamarin.Android and Xamarin.iOS (C#)

The redistributable file **LiquidTechnologies.Runtime.PCL.dll** can be found in the 'Redist16\DotNet' sub folder. This is a .Net Component with the following Namespace:

```
LiquidTechnologies.Runtime.PCL
```

Your generated component, and possibly your client, should reference this file.

.Net Core 1.1 and .Net Standard 1.6 (C#)

The redistributable file **LiquidTechnologies.Runtime.Standard16.dll** can be found in the 'Redist16\DotNet' sub folder. This is a .Net Component with the following Namespace:

```
LiquidTechnologies.Runtime.Standard16
```

Your generated component, and possibly your client, should reference this file.

.Net Core 2.0 and .Net Standard 2.0 (C#)

The redistributable file **LiquidTechnologies.Runtime.Standard20.dll** can be found in the 'Redist16\DotNet' sub folder. This is a .Net Component with the following Namespace:

```
LiquidTechnologies.Runtime.Standard20
```

Your generated component, and possibly your client, should reference this file.

If you are using Visual Studio 2017, a Nuget package and can be installed using the Nuget Manager in Visual Studio containing the Liquid .Net Runtimes.

C++

All C++ generated code includes the files in the 'Redist16\cpp\include' folder. However the files that need to be linked against and redistributed are dependant on the platform and type of build.

Windows

The build version you require further depends on the C Runtime version that your compiler is using. The 32 bit dll files are found in the 'Redist16\cpp\win32\bin' folder and corresponding lib files in the 'Redist16\cpp\win32\lib' folder, and the 64 bit dll files are found in the 'Redist16\cpp\win64\bin' folder and corresponding lib files in the 'Redist16\cpp\win64\lib' folder.

Windows Builds

The Liquid Runtime libraries for Windows use the following naming convention:

Release libraries: LtXmlLib16_vcVV.dll

Debug Libraries: LtXmlLib16D_vcVV.dll

Unicode Release libraries: LtXmlLib16U_vcVV.dll

Unicode Debug Libraries: LtXmlLib16UD_vcVV.dll

64 bit libraries: LtXmlLib16_vcVVx64.dll

XP specific libraries: LtXmlLib16_vcVV_xp.dll

Where VV is the C++ version you are using.

E.g.

A 32 bit non-Unicode Release library using C++ 14.1 would look like:

LtXmlLib16_vc141.dll

Whereas a 64 bit Unicode Debug library using C++ 14.1 specifically for XP would look like:

LtXmlLib16UD_vc141x64_xp.dll

These are all C++ dynamic link libraries and have the following namespace:

```
LtXmlLib16
```

Your generated component, and possibly your client, should reference this file.

Linux

The build version you require further depends on the gcc compiler version. The.so libraries are found in the 'Redist16\cpp\linux\bin' folder and are specific to the g++ version you are using.

Linux Builds

The Liquid Runtime libraries for Linux use the following naming convention:

Release libraries: libXml16_gccVV.so

Unicode Release libraries: libXml16U_vcVV.so

64 bit libraries: libXml16_vcVV_64.so

Where VV is the g++ version you are using.

E.g.

A 32 bit non-Unicode Release library using g++ 4.3.3 would look like:

libXml16_gcc433.so

Whereas a 64 bit Unicode Release library using g++ 7.2.0 would look like:

libXml16U_gcc720_64.so

These are all C++ dynamic link libraries and have the following namespace:

```
LtXmlLib16
```

Your generated component, and possibly your client, should reference this file.

Java

The redistributable file **LtXmlLib16.jar** can be found in the 'Redist16\Java' sub directory. This is a Java jar file containing Java class files with the following Namespaces:

```
com.liquid_technologies.LtXmlLib16
com.liquid_technologies.LtXmlLib16.exceptions
```

Your generated component, and possibly your client, should reference these files.

Visual Basic 6

The redistributable files **LtXmlComLib16.dll and LtXmlComHelp16.dll** can be found in the 'Redist16\COM' sub directory. These are both COM Components.

Your generated component, and possibly your client, should reference **LtXmlComLib16.dll**.

Multilanguage, Unicode and UTF-8 Encoding Considerations

In order to explain how the Liquid XML Data Binging Libraries handle Multilanguage issues we will first explain the different technologies involved.

Unicode

On the windows platform Unicode encodes each character using 2 bytes. This gives a possible 0xffff (65535) characters. The Unicode standard dictates the code used to represent each character.

Typically, when a Unicode document is written to file, the file is prefixed with the 2 bytes 0xFF 0xFE. When the file is read these bytes indicate that the file is Unicode and are ignored.

Windows 32 bit Operating Systems (NT, 2000, XP and Windows 7) all use Unicode internally to represent characters.

CodePages

On older systems, each character is encoded using a single byte, this provides 0xff (256) possible characters. This system proved limited as there where not enough encodings for all the characters that needed to be represented. The solution was to use a code page. The code page specifies which characters the 256 encodings map to. An application will set its codepage, thus determining which 256 characters it has available to use.

Problems arise with this system when converting data between different code pages. It is normal for many of the characters from the source code page to be missing from the destination code page (and visa versa). This causes data or the interpretation of the data to be corrupted. To avoid this typically a system will try to use the same code page across all the applications.

MultiByte

There are a number of MultiByte formats, but we are going to look at UTF-8 (Unicode Transformation Format). MultiByte means that variable a number of bytes are used to encode a single character.

If you develop a new application that uses Unicode internally, everything is fine until you need to pass some information via an older system (e.g. Email). The old system does not understand Unicode, and when you attempt to pass your Unicode strings to it, it comes across a null in the string (0x00 0x41 is the A character) and stops reading, believing it has hit the end of the string.

MultiByte encoding promises a solution. Before passing your string to the older system, you encode your Unicode string as UTF-8. The legacy system can quite happily use the encoded string, even if it can't represent it all correctly on the screen (extended characters will not appear correctly), but all regular chars (0x00-0x7f) are fine. On the other side the data can be extracted and decoded, back into Unicode.

The advantage of UTF-8 over other encodings like BASE64, UUE, and HEX, is that the message is still (mostly) viewable, with only the extended characters corrupted.

XML

Xml documents can be encoded using a number of different encodings. The type of encoding is indicated using the encoding tag in the document header (i.e. <?xml version="1.0" encoding="UTF-8"?>).

Writing an XML document to file

When an XML document is persisted as a file, it is safer to consider it in terms as of a stream of bytes as opposed to a stream of characters. When an XML document is serialized to a file, an encoding is applied to it. The resulting file will then be correctly encoded given the encoding applied.

If a Unicode encoding is applied, the resulting file is prefixed with the Unicode header 0xFF 0xFE, and will be encoded with 2 bytes per character.

If a UTF-8 encoding is applied the resulting file will contain a variable number of bytes per character. If this file is then viewed using a tool incapable of decoding UTF-8, then you may see it contains a number of strange characters. If the file is viewed using an UTF-8 compliant application (e.g. Internet Explorer or Notepad on Win2000) then the XML Document will appear with the correct characters. NOTE: If characters are corrupted or misrepresented, it should be noted that some fonts do not contain the full Unicode set.

Turning an XML Document into a String

When an XML document is created from a generated class using `ToXml` (`ToXml` returns a string). The string returned is encoded as Unicode (except Non-Unicode C++ builds), however the XML Document header does not show any encoding (<?xml version="1.0"?>).

Unicode is the internal character representation for VB6, .Net & Java, as such if it is written to file or passed to another application, it should be passed as Unicode. If it has to be converted to a 1 byte per character representation prior to this, then data will likely be corrupted if complex characters have been used within the document.

If you need to persist an XML document to a file use `ToXmlFile`, if you need pass an XML document to another (non-Unicode) application, then you should use `ToXmlStream`.

Passing an XML Document to an ASCII or ANSI application

It is common to want to pass the XML Document you have created to a non-Unicode application. If you need to do this then you may look first at `ToXml`, this will provide you with a Unicode string, however converting this to an ASCII or ANSI string may cause the corruption of complex characters (you loose information going from 2 bytes to 1 byte per character). You could take the string returned from `ToXml`, and apply your own UTF-8 encoding, however the encoding attribute in the header (<?xml version="1.0" encoding="UTF-8"?>) would not be present, and the XML parser decoding the document may misinterpret it.

The better solution is to use the `ToXmlStream` method. This allows you to specify an encoding, and returns a stream of bytes (array of bytes in VB). This byte stream is a representation of the XML Document in the given encoding, containing the correct encoding attribute in the header (<?xml version="1.0" encoding="UTF-8"?>).

CHAPTER 7
Schema Versioning

There are 2 main considerations when versioning schemas,

- Backwards compatibility – Allowing code written to work against a new version of the schema to continue to work against older versions.
- Future compatibility – Allowing code written to work against an old version to continue working if messages constructed against a new version of the schema are passed to it.

There are broadly speaking 2 types of changes,

- Minor – ones that preserve compatibility with the previous versions. (e.g., new optional elements, attributes, extensions to an enumerated list, etc.). These can allow Backwards & Future compatibility to be preserved
- Major – ones that break compatibility with existing versions (these break Backwards & Future compatibility).

Minor Changes

It is only possible to provide any real form of compatibility with minor changes, so we will consider this first.

Minor changes should be optional and should appear after all exiting elements/attributes.

When designing a schema it is important to decide from what perspective you want versioning to be performed.

Case 1

You are implementing a service. The xml messages you are building describe requests/response messages to/from this service. The service will always be implemented using Liquid XML Library generated from the latest version of the messaging schema, however when you upgrade the clients may still be using older versions.

In this case you need backward compatibility. If only ‘minor’ changes have occurred between versions then if a message is received constructed according to an old version of the schema, then there will just be optional attributes/elements missing from it. This combined with the SchemaVersion implemented in the root element allow this missing data to be defaulted appropriately.

Case 2

You are implementing a client. The xml messages you are building describe requests/responses to the service. You want to build the application, but if the service is upgraded you don’t want to have to roll out the clients again.

If only *minor* changes have taken place to the schema, then the messages being sent from the client to the service will still be compatible, but the responses received may now contain new (unexpected attributes/elements/enumerations). There are a number of ways for the client to deal with these.

The schema can be altered (either formally or on a working copy if you don’t own the schema) in order to facilitate version proofing. If an <any> element is appended to each element. These <any>’s will form catch all’s for any unknown elements. This can be time consuming on large schemas, and can cause the schema to become con-deterministic.

The XmlSerializationContext can be setup to ignore unknown elements and attributes. This is a simple solution, and is easy to implement especially if you don’t have control of the schema.

Major changes

When making *major* changes to the schema there is no way to preserve compatibility between versions, so a new version needs to be produced i.e. 1.0 to 2.0.

If you are building a client then this is not so much of an issue, you build your client to work against a specific major version of the schema, your client can cope with minor changes to the standard (see previous section), but there is no clear way to cope with major changes without re-working code or when using Liquid XML re-generating the library.

If you are building a service then you may have to support a number of versions until your clients upgrade (possibly indefinitely). At this point the strategy you have taken for versioning becomes more critical, both for simplifying your own life and your clients upgrade path.

There are a number of approaches to producing major versions.

Create a new schema

This requires a new namespace and filename/url for the schema, in order to make it distinct from the previous one. This does of course mean that if you are implementing this server side you now have 2 distinct libraries providing similar functionality, and the code behind them needs to be replicated. This approach is better suited when the schema is changing significantly, and the differences between the schemas outweigh any re-use you may get from code produced against the previous version.

Add to the existing one

The simplest way to create a new version is to add new functionality to an existing schema, but do it in such a way that it will support old and new messages. SubstitionGroups and extensions are a useful tool in accomplishing this, let's look at an example.

The following is the first version of a schema, the CustomerDetails element can contain a Person element, which describes a person.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified">
    <xs:element name="Person">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="Name" type="xs:string" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="CustomerDetails">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="Person" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

And a sample file…

```
<?xml version="1.0" encoding="UTF-8"?>
<CustomerDetails xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Person.xsd">
    <Person>
        <Name>Joe Bloggs</Name>
    </Person>
</CustomerDetails>
```

Later the Person definition is found to be a little lacking, and it is decided that the forename & surname need breaking out. If we were to just remove the 'Name' element and add the Surname and Forename elements then this would break compatibility with the previous version.

An alternative approach is to create a substitution group or choice allowing legacy documents to be readable.

This 'version 2' of the schema shows how to use a choice to provide compatibility with the existing schema, while allowing a new Person2 element to be used interchangeably with it. The choice solution is simple to implement if Person appears in a small number of places, but the substitution group approach is better if Person appears in a large number of locations.

The Choice approach also has the draw back that each instance creates a new class, which needs to be dealt with as a separate entity in the code.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified">
    <xs:element name="Person">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="Name" type="xs:string" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="CustomerDetails">
        <xs:complexType>
            <xs:choice>
                <xs:element ref="Person" />
                <xs:element name="Person2" type="Person2Type" />
            </xs:choice>
        </xs:complexType>
    </xs:element>
    <xs:complexType name="Person2Type">
        <xs:sequence>
            <xs:element name="Forename" type="xs:string" />
            <xs:element name="Surname" type="xs:string" />
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

'Version 2' of the schema uses substitution groups to accomplish the same thing. This approach is more effort to setup, but is easier to change each instance of Person in use (simply swap it for a reference to the PersonGroup element).

The code generated for the Person substation group is also consistent, so a single function can be used to manipulate Person/Person2 elements throughout the code.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified">
    <xs:element name="CustomerDetails">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="PersonGroup" />
            </xs:sequence>
```

```
        </xs:complexType>
    </xs:element>
    <xs:complexType name="PersonBase" abstract="true" />
    <xs:complexType name="PersonType">
        <xs:complexContent>
            <xs:extension base="PersonBase">
                <xs:sequence>
                    <xs:element name="Name" type="xs:string" />
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
    <xs:complexType name="Person2Type">
        <xs:complexContent>
            <xs:extension base="PersonBase">
                <xs:sequence>
                    <xs:element name="Forename" type="xs:string" />
                    <xs:element name="Surname" type="xs:string" />
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
    <xs:element name="PersonGroup" type="PersonBase" abstract="true" />
    <xs:element name="Person" type="PersonType" substitutionGroup="PersonGroup" />
    <xs:element name="Person2" type="Person2Type" substitutionGroup="PersonGroup" />
</xs:schema>
```

This new versions of the schema will allow the previous XML sample to be read as well as the new version.

```
<?xml version="1.0" encoding="UTF-8"?>
<CustomerDetails xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Person2.xsd">
    <Person2>
        <Forename>Joe</Forename>
        <Surname>Bloggs</Surname>
    </Person2>
</CustomerDetails>
```

Note
Even if you achieve your goal of making your schema backwardly compatible, then you must remember that if clients come in with requests marked as version 1, they will still expect the response that conforms to Version 1.

Best practices

- Include the version in the schema itself i.e.

```
<xs:schema xmlns="http://www.exampleSchema" targetNamespace="http://www.exampleSchema"
xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified" version="1.3">
    <xs:element name="Example">
        <xs:complexType>
            ….
            <xs:attribute name="schemaVersion" type="xs:string" use="required" />
        </xs:complexType>
    </xs:element>
```

- Use a convention for schema versioning to indicate whether the schema changed significantly (case 1) or was only extended (case 2). For example, for case 1 a version could increment by one (e.g., v1.0 to v2.0) whereas for case 2, a version could increment by less than one (e.g., v1.2 to v1.3).

- Be consistent, create root level complexTypes for each element, use naming conventions for complextypes (i.e. postfix Type)

- Avoid using the same name for complex types and element definitions.
- Avoid complex structures substitution groups, extensions etc unless you have a clear reason to use them.