Common Pitfalls in System.XML 4.0

By Kurt Evenepoel, April 2010
Technical Lead .NET
Wolters-Kluwer Belgium

Table of contents

Introduction 3
Basic knowledge 3
Pitfalls 4
Default settings for reading and writing examined 4
XmlWriterSettings 4
XmlReaderSettings 5
How to handle white-space for mixed content 6
About significant and insignificant white-space, and pretty-printing 6
Inconsistencies between XDocument, XmlDocument and XmlReaders/Writers regarding white-space 8
Conclusion 11
How to deal with quirky namespaces 11
Handling DTD's 12
Conclusion 14
Base URI's and Streams, consequences for schemas 14
What went obsolete in .NET 4.0? 19
Changes in System.XML and related namespaces 20
What changed in the System.XML namespace? 20
XmlConvert 20
XmlReader 20
XmlReaderSettings 21
XmlResolver 21
XmlTextReader 22
XmlUrlResolver 22
XmlValidatingReader 22
XmlWriterSettings 23
DtdProcessing and NamespaceHandling 23
What changed in the System.Xml.Serialization namespace? 23
XmlSerializer, XmlSerializerFactory 23
What changed in the System.Xml.Xsl namespace? 23
XslCompiledTransform 23
What changed in the System.Xml.Linq namespace 24
SaveOptions 24
XDocument, XElement and XStreamingElement 24
XNode 26
Conclusion 27

Introduction

This small session presents my ***own research*** behind some of the pitfalls found in System.Xml and child namespaces. We'll be discussing :

- non-obvious differences between ***XDocument*** and ***XmlReaders/Writers***
- ***namespace handling*** with duplicate namespace declarations
- how to properly treat ***mixed-content*** XML files
- difficulties with Streams and SchemaSet

Finally we'll take a look at what's been made obsolete and what's brand new in the different namespaces in System.Xml for .NET 4.0.

You will need:

- Visual Studio 2010 (most will work with VS2008 SP1 too)
- A strong base of C# and XML, working knowledge of XPath and XSLT to understand every example. If you also know DTD's and XML Schema's all the better!

This session is not intended to teach you about how to process basic XML files in C#, you should know this already, sorry!

If you're in a hurry, skip forward to the conclusions of each topic, everything in between is background information or proof.

Basic knowledge

There are four major ways of working with native XML:

- Serialization of .NET classes (not discussed)
- Using XmlReaders and XmlWriters
- Using XmlDocument
- Using XDocument

You should know how to use all of these.
As a quick refresher, here's an example of how to create an XDocument:

```
XDocument document = new XDocument("root",
    new XElement("first-child",
        new XAttribute("an-attribute", "with a value"),
        "some text inside"));
```

As you'll see, XDocument is the class to use (with caution) for handling most situations.

Pitfalls

Default settings for reading and writing examined

Default settings should ideally be the same for every overload. The .NET library behaves predictably in this regard for XML reading and writing with the XmlReader.Create and XmlWriter.Create static methods. That is, if you take into account some considerations.

See included project: XmlReaderWriterDefaults

XmlWriterSettings

The factory method ***XmlWriter.Create*** has a pretty consistent behaviour across constructor calls if you know what to watch out for.
CloseOutput and Encoding are the big differences between the different constructors.
Constructors based on in-memory string structures like StringBuilders by default use what Microsoft calls '***Unicode***' (this does not apply to Streams, whose representation in-memory isn't 'text' per se). The ***others***, based on filestreams that don't use StringBuilders, by default use ***UTF-8*** (which obviously is Unicode too, but another Unicode format than 'Unicode' or UTF-16). ***CloseOutput is true for overloads that use a filename.*** That means you are responsible for closing the stream yourself, for example by using the 'using' keyword, unless you hand it a filename, in that case it is closed automatically when the reader or writer is disposed.
There were ***no significant changes between .NET 2.0 and 4.0 except for "NamespaceHandling"***, and code should have stayed compatible.

The follow are the defaults ***shared for every constructor*** call tested:

CheckCharacters	TRUE
ConformanceLevel	Document
Indent	FALSE
IndentChars	(space)
NamespaceHandling (added in .NET 4.0)	Default
NewLineChars	(newline)
NewLineHandling	Replace
NewLineOnAttributes	FALSE
OmitXmlDeclaration	FALSE

Differences between every constructor call are:

constructor XmlWriter 2.0	CloseOutput	Encoding

XmlWriter.Create(dummyStream)	FALSE	UTF8Encoding
XmlWriter.Create("c:\in.xml")	TRUE	UTF8Encoding
XmlWriter.Create(dummyBuilder)	FALSE	UnicodeEncoding
XmlWriter.Create(new StringWriter(dummyBuilder)	FALSE	UnicodeEncoding
XmlWriter.Create(Create("c:\in2.xml"))	TRUE	UTF8Encoding
XmlWriter.Create(dummyStream, defaultSettings)	FALSE	UTF8Encoding
XmlWriter.Create(@"c:\in.xml", defaultSettings)	TRUE	UTF8Encoding
XmlWriter.Create(dummyBuilder, defaultSettings)	FALSE	UnicodeEncoding
XmlWriter.Create(new StringWriter(dummyBuilder), defaultSettings)	FALSE	UnicodeEncoding
XmlWriter.Create(XmlWriter.Create(@"c:\in2.xml", defaultSettings))	TRUE	UTF8Encoding
XmlWriter 4.0		
XmlWriter.Create(dummyStream)	FALSE	UTF8Encoding
XmlWriter.Create("c:\in.xml")	TRUE	UTF8Encoding
XmlWriter.Create(dummyBuilder)	FALSE	UnicodeEncoding
XmlWriter.Create(new StringWriter(dummyBuilder)	FALSE	UnicodeEncoding
XmlWriter.Create(dummyStream, defaultSettings)	FALSE	UTF8Encoding
XmlWriter.Create(@"c:\in.xml", defaultSettings)	TRUE	UTF8Encoding
XmlWriter.Create(dummyBuilder, defaultSettings)	FALSE	UnicodeEncoding
XmlWriter.Create(new StringWriter(dummyBuilder), defaultSettings)	FALSE	UnicodeEncoding
XmlWriter.Create(XmlWriter.Create(@"c:\in2.xml", defaultSettings))	TRUE	UTF8Encoding

XmlReaderSettings

The factory method ***XmlReader.Create*** has a consistent behaviour across all constructor calls tested, both for .NET 2.0 as for .NET 4.0. ***ProhibitDtd is now obsolete and was replaced by DtdProcessing.*** This does mean that all old code will have warnings, but those are easy to remedy. The Encoding and CloseInput settings behave similar to the writer settings: streams are closed automatically if you hand the factory method a filename. Encoding is 'Unicode' (UTF-16) if you're using a StringBuilder as a base to write to it.

CheckCharacters	TRUE
CloseInput	FALSE

ConformanceLevel	Document
DtdProcessing (added in .NET 4.0)	Prohibit (compatible with ProhibitDtd=true)
IgnoreComments	FALSE
IgnoreProcessingInstructions	FALSE
IgnoreWhitespace	FALSE
LineNumberOffset	0
LinePositionOffset	0
MaxCharactersFromEntities	0
MaxCharactersInDocument	0
NameTable	(empty)
ProhibitDtd	TRUE
Schemas	System.Xml.Schema.XmlSchemaSet
ValidationFlags	ProcessIdentityConstraints, AllowXmlAttributes
ValidationType	None

How to handle white-space for mixed content

See included projects: XsltWhiteSpace, XdocumentWhitespace

About significant and insignificant white-space, and pretty-printing

Whitespaces are tabs, newlines or spaces (but not 'non-breaking spaces' – in HTML that is the character entity *** *** - it's a special space that prevents going to a new line).
Pretty-printing an XML document means to indent it so its more easily human-readable.

```
<div>
	<p>
		<em>C# :</em>
		<span class="description">My fave</span>
	</p>
	<br/>
	<p>
		<em>VB.NET :</em>
		<span class="description">Jean's fave</span>
	</p>
</div>
```

This XML is pretty-printed.
The original could have been like this:

```
<div>
	<p><em>C# :</em> <span class="description">My fave</span></p><br />
	<p><em>VB.NET :</em> <span class="description">Jean's fave</span></p>
</div>
```

If there is no DTD (doctype declaration) specified in the XML, whitespace is not ignorable[1]. A DTD can specify which whitespaces can be ignored, and which cannot. An XML parser has no idea whether whitespace is meant to be important (significant), and should keep all whitespaces it's not instructed to discard by the DTD. The XML specification does not force this (XML processors can choose if they want to treat spaces as significant or insignificant), but it is the only 'safe' default, and companies like IBM and Oracle also figured this out. The default behaviour should be to keep whitespaces (but you'll see later this is not the implementation with XDocument nor XmlDocument). Notice in the above document that tabs are inserted. ***Tabs are valid XML whitespace***. The in-memory DOM doesn't change until the document is parsed again from the output that includes the new tabs – for example after performing an XSL tranforsmation on it.

Similarily, an ***XSLT processor*** has no clue what to do with whitespaces, and it's default behaviour is to ***preserve spaces***. (see: http://www.w3schools.com/XSL/el_preserve-space.asp [2]).
To make spaces insignificant, you need to use the ***<xsl:strip-space /> element*** at the top of your stylesheet. To verify we'll run the following stylesheet over both versions:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
>
    <xsl:output method="text" />
</xsl:stylesheet>
```

This is an 'empty' stylesheet. Providing an <xsl:template> for an element overrides the default handling, but we're not going to do that. The default handling for any element is to strip the element itself, strip the attributes, and keep all text, and this is exactly what we want to do.

- Non-pretty printed version

```
C# : My fave
VB.NET : Jean's fave
```

- Pretty-printed version

```
C# :
My fave


VB.NET :
Jean's fave
```

As you can see, ***pretty-printing actually changes the content*** of the document somewhat! There are newlines and tabs added that were not there in the original marked up docoment. In most cases this is harmless, especially with serialized .NET or data objects, but with markup languages like XHTML and possibly with other mixed-content formats, that's an inconvenience. To actually get what you needed, the XSLT would turn more complex and sometimes use variables and the normalize-space

function. For a good example, take a look on MSDN (http://msdn.microsoft.com/en-us/library/ms256063.aspx)

Inconsistencies between XDocument, XmlDocument and XmlReaders/Writers regarding white-space

In this light, a nasty difference between XmlReaders/Writers and XDocument is the way whitespace is treated. (see http://msdn.microsoft.com/en-us/library/bb387014.aspx) The defaults between XDocument and XmlReader are different:

XDocument	XmlReader
LoadOptions.PreserveWhitespace	XmlReaderSettings.IgnoreWhitespace = false (***default***)
(not specified: does not preserve whitespace, ***default***)	XmlReaderSettings.IgnoreWhitespace = true

This means that using an XmlReader means you will be using 'standard' XML handling with the default settings. Using XDocument means that mixed content will come out with quirks in the spacing for mixed content, when using default settings...quite unexpected?

Additionally, ***XmlReaderSettings and LoadOptions can conflict***. By handing it a 'LoadOptions' argument when loading the XDocument, the LoadOptions are used as you'd expect ***except when using an XmlReader/Writer, then the options of the reader/writer are used***.

XDocument.Load(string)	Ignored
XDocument.Load(string,LoadOptions.PreserveWhitespace)	Preserved
XDocument.Load(XmlReader, LoadOptions.PreserveWhitespace), XmlReaderSettings.IgnoreWhitespace=false	***Preserved***
XDocument.Load(XmlReader, LoadOptions.None), XmlReaderSettings.IgnoreWhitespace=true	***Ignored***
XDocument.Load(XmlReader, LoadOptions.PreserveWhitespace, LoadOptions.None), XmlReaderSettings.IgnoreWhitespace=false	***Preserved***
XDocument.Load(XmlReader), XmlReaderSettings.IgnoreWhitespace=true	***Ignored***
XmlReader, XmlReaderSettings.IgnoreWhitespace=false	Preserved
XmlReader, XmlReaderSettings.IgnoreWhitespace=true	Ignored

This difficulty exists since XDocument exists, and tests point out that it still exists in .NET 4.0- the API has not changed in that regard.

Why is this important? Look at the results. We had a text with 2 elements: one and one <i>. In the original document, there is a space between the two. This is what XML calls 'mixed content'. Notice how the space is stripped in some of the overloads that ignore whitespace. The whitespace is significant, and an XSLT processor usually

keeps this space. The result is that ***there is a difference in the 'Value' property of XDocument*** (which is essentially the same as an xslt without any templates defined), depending on your settings:

- whitespace not preserved

```
Paragraph1Paragraph2
```

- whitespace preserved

```
Paragraph1 Paragraph2
```

Where you had marked up text before, that correctly had spaces in the right places, everything sticks together now.

Again, this is not very important if you are serializing classes, but it is quite important if you are doing transformations on text, where any ***mixed content text will have spaces missing in the rendering if you used XDocument without LoadOptions***.

A similar story can be told for writing out the XDocument:
XDocument.Save has a ***SaveOptions*** argument wich can be set to ***disable formatting***.
This has the same effect as putting the ***XmlWriterSettings "Indent"*** property to "false".

By default indenting on save is enabled, but that's not always what you want and can cause your XSLT's to need a lot of 'normalize-space()' calls for 'mixed content'.

XDocument	XmlWriter
SaveOptions.DisableFormatting	XmlWriterSettings.Indent = false (***default***)
(not specified: uses 'pretty print' XML, ***default***)	XmlWriterSettings.Indent = true

XmlWriterSettings and SaveOptions can't conflict however: there is ***no overload on XDocument.Save that allows for giving both an XmlWriter and SaveOptions***, making it easier for you:

XDocument.Save(TextWriter, SaveOptions.DisableFormatting)	Not indented
XDocument.Save(TextWriter)	Indented
XDocument.Save(XmlWriter), XmlWriterSettings.Indent=true	Indented
XDocument.Save(XmlWriter), XmlWriterSettings.Indent=false	Not indented

For ToString(SaveOptions) the same principles are at work: default ToString() will indent your XML, if you don't want this you need to explicitly tell it not to. ToString() does not include the XML declaration.
By default, XDocument applies pretty-print, and XmlWriter does not. By default XDocument strips the spaces again on load, but may strip more than was added by saving. XmlReader does not strip any spaces by default.

That leaves the XmlDocument class. How would this class behave? Prepare to be confused.. Because this one doesn't listen to your XmlReaderSettings.

For example:

```
using (XmlReader rdr = XmlReader.Create("source.xml", new XmlReaderSettings
{
    IgnoreWhitespace = false
}))
{
    XmlDocument doc = new XmlDocument();
    doc.Load(rdr);
    return doc;
}
```

This code fragment strips whitespace, even though you explicitly told it to keep them! The source document with the following contents (shortened to fit on one line)

```
..<div><em>Paragraph1</em> <span>Paragraph2</span></div>..
```

actually gets read as the following:

```
..<div><em>Paragraph1</em><span>Paragraph2</span></div>..
```

So ***even with XmlReaderSettings set to keep whitespaces, XmlDocument strips them***. It's the reading of the document that is affected. These different ways were tested and all yielded the same result:

- XmlDocument.Load with XmlReaderSettings IgnoreWhitespace=true
- XmlDocument.ParseXml
- XmlDocument.Load with XmlReaderSettings IgnoreWhitespace=false
- XmlDocument.Load without XmlReaderSettings

All of these yielded a result that stripped spaces regardless of XmlReaderSettings. Why is that? Because, unlike with XDocument where the XmlReader/Writer settings take precedence, when it comes to white-space, XmlDocument uses the ***"PreserveWhitespace" property*** before loading. This property overrides the settings with which the document is read, and it's ***by default, set to… False***. The default is the same between XmlDocument and XDocument, but the way to override it isn't.

In short, to properly use the XmlDocument class with white-space, you have to use code similar to this:

```
using (XmlReader rdr = XmlReader.Create("source.xml"))
{
    XmlDocument doc = new XmlDocument { PreserveWhitespace = true }
    doc.Load(rdr);
    return doc;
}
```

This option is a recipe for bugs, there *will* be times you forgot to preserve white space because its not listed in the constructor and code needs to be delivered. Yesterday.

Another possible solution is to put the ***xml:space="preserve"*** attribute in your source:

```
<div xml:space="preserve"><em>Paragraph1</em> <span>Paragraph2</span></div>
```

This way the XmlDocument *does* keep the white-spaces. But this is very annoying. You can read more on http://www.ibm.com/developerworks/xml/library/x-tipwhitesp.html. This option requires you to change your XML files, which is also unpleasant and you may not even be in control of the source format.

Conclusion

So what is ***the safest way to parse XML that contains mixed content***?

- while loading, explicitly keep whitespaces
  - XmlReader + XmlReaderSettings.IgnoreWhitespace = false (default)
  - XDocument.Load(LoadOptions.PreserveWhitespace)
- When saving intermediary results, don't indent
  - XmlWriter + XmlWriterSettings.Indent = false (default)
  - XDocument.Save(SaveOptions.DisableFormatting)
- Alternatively, only use ***overloads for XDocument that use XmlReaders/Writers***, in this case the XmlReader/Writer settings are used. This is my personal preferred method- always use my own readers and writers.
- ***When using XmlDocument, XmlReader/WriterSettings are ignored- you need to set the PreserveWhitespace property to true.***

This shows that for mixed content you cannot use the easiest overloads of the static methods in the XDocument class. This has remained exactly the same in .NET 3.5 and .NET 4.0, but causes confusion. In theory, the XDocument interpretation is too liberal in assuming whitespaces can be safely ignored to safely work with marked up text documents, because it requires a lot of care to check the call every time. Practically, most errors that spring from this non-standard behaviour are usually non-blocking, unless you're in the publishing or markup industry where the XML is on a different level of difficulty to parse, not just used as a human-readable, interchangeable data storage format. ***XDocument's defaults are geared towards data, not marked up text. The same goes for XmlDocument, but the ways to override the defaulst are different.***

How to deal with quirky namespaces

See included project: Namespaces

Namespaces can be handy, but they can also cause you countless headaches.
In the following example XML document, there is a ***duplicate namespace declaration*** on the local-disk element. I've also included **the same namespace uri twice but with a different prefix**, on the same element.

```
<laptop xmlns:work="http://hard.work.com"
xmlns:work2="http://too.hard.work.com" xmlns:work3="http://hard.work.com">
  <work2:drive>
    <local-disk xmlns:work="http://hard.work.com">C:\</local-disk>
    <work:network-drive>Z:\</work:network-drive>
  </work2:drive>
</laptop>
```

Loading this file with the new option to remove duplicate namespaces yields the following result:

```
<laptop xmlns:work="http://hard.work.com"
xmlns:work2="http://very.hard.work.com" xmlns:work3="http://hard.work.com">
  <work2:drive>
```

```
    <local-disk>C:\</local-disk>
    <work3:network-drive>Z:\</work3:network-drive>
  </work2:drive>
</laptop>
```

Notice that references to the 'work' namespace have been replaced by the 'work3' namespace, and that the duplicate namespace declaration on the local-disk element was removed. Notice also that, since the work and work3 namespaces have the same URI they're logically equivalent. So .NET's decision to put everything on the same namespace makes sense. It would be logical then maybe to remove the 'work' namespace automatically too, but .NET leaves it declared.

This table explains how loading an XDocument affects the quirky namespaces:

Duplicate URI in the namespaces of an ancestor, but different alias	***Elements take the last defined namespace alias, even if they were originally written with another alias***
Duplicate URI in the namespace of an element, with the same alias	Unable to load document
Namespace and alias is also defined on an ancestor, default	The duplicate namespace declaration stays as read
Namespace and alias is also defined on an ancestor, and NamespaceHandling.OmitDuplicates is used	The duplicate namespace declaration is removed from the child

Handling DTD's

See included project: UsingDTDs

Using the ***DtdProcessing*** enumeration, wich can be set with XmlReaderSettings for an XmlReader, you can allow the use of DTD's. By default, DTD's are not allowed because its an URI, provided by the document itself, that the processor is instructed to visit. This could be used for distributed attacks or with other security risks – for example someone put a DTD in 20 000 documents you're parsing, every 0.05 seconds your server will attempt to connect to an uri of the attacker's choice.
However, a document is possibly not parseable (this means ***NO DOM POSSIBLE***) ***without its DTD if it has one*** because the document may contain entities that are described in the DTD. .NET 4.0 now includes the option to load XML documents that contain DTD's without actually getting the DTD and visiting the URI mentioned, using XML Readers. This was already possible using XDocuments in .NET 3.5.

Prohibit	No DTD's are allowed, an exception occurs if you try to load a document that has one (***default for XmlReaderSettings***).
Parse	DTD's are allowed, and if found, they are downloaded with the XmlResolver in the reader and parsed as part of the document. An

	error in the DTD, inability to find the DTD or a document not adhering to the DTD will cause an exception to be thrown if validation is enabled too.
Ignore	DTD's are allowed, but ignored. The DTD is not loaded. If the XML document contains entities that were defined in the DTD (DTD's do more than only validation, they can define character entities for example), the object can't be successfully loaded and an exception is thrown (***default for XDocument, both in .NET 3.5 as .NET 4.0)***

One thing to note is that XDocument.Load, when using the overload with an XmlReader, uses the DTD settings of the XmlReader.

Method	Result on DTD
XDocument.Load(string)	Ignored
XmlReader, no XmlReaderSettings	Prohibited
XmlReader XmlReaderSettings.DtdProcessing=DtdProcessing.Parse	Parsed, not validated
XmlReader XmlReaderSettings.DtdProcessing=DtdProcessing.Parse XmlReaderSettings.ValidationType=ValidationType.DTD	Parsed and validated

So to allow XML documents to be ***validated*** according to their DTD, you need to set the following settings:

```
using (XmlReader reader = XmlReader.Create("people.xml", new XmlReaderSettings
{
    DtdProcessing = DtdProcessing.Parse,
    ValidationType = ValidationType.DTD
}))
{
    XDocument.Load(reader);
}
```

There is still no reusablility mechanism for DTD's like there is for schema's however, so every time a document is parsed the DTD is parsed and retrieved. And you can't validate an XML document easily with a DTD your program owns, you will need to ***re-parse the document after adding it*** (careful with indenting!). This is in contrast to XML schema's- XML schema's can be reused for parsing a set of documents.

Traditionally there was no caching mechanism present out of the box, and many people wrote some kind of cache for their XmlResolvers. ***DTD's are resolved with XmlResolvers*** too, and .NET 4.0 allows you to set a caching policy on resolved streams now. So you could ***resolve DTD's with your XmlUrlResolver's caching policy if the DTD is located on a remote PC*** (ie if you have a high retrieval cost).
An example of how to do this:

```
using (XmlReader reader =
XmlReader.Create("http://www.microsoft.com/en/us/default.aspx", new
XmlReaderSettings
{
```

```
    DtdProcessing = DtdProcessing.Parse,
    ValidationType = ValidationType.DTD,
    XmlResolver = new XmlUrlResolver
    {
        CachePolicy = new
RequestCachePolicy(RequestCacheLevel.CacheIfAvailable),
    }
}))
{
    XDocument.Load(reader);
}
```

If you run this, you'll see exactly why DTD's are generally resolved locally using a resolver: the webpage that is supposed to host the DTD serves a '503 Page Unavailable'. That means your HTML pages cannot be properly validated unless you have your own copy of the DTD's, because no document is complete without its DTD if it mentions it, and if there's entities inside you can't even load it without it (for example). ***Imagine every browser or HTML processor in the world retrieving the DTD every time a HTML page is loaded***- millions at the same time- what would that cost, and is there even infrastructure capable of handling that?

Conclusion

Avoid adding DTD's to your document whenever possible. Keep entities out of your documents (easiest way is to use UTF-8) and if you need to validate you can use DTD or schema at your convenience. But your program should decide what DTD or schema to validate against. ***You don't want to validate according to the document's rules, but to the rules your program is expecting***: your program needs to dictate what it expects, it couldn't care less if the document thinks it's valid. Validation serves a purpose, and the purpose is not 'making sure the document is valid according to its own standards', wich is what including a DTD inside of a document actually means.

An example of a worst case scenario:

A webservice processes files using a 'customers.dtd' file. It expects the files to be valid XML files adhering to the customer DTD. A new programmer joined your team, sees the folder and decides to put files adhering to the 'sales.dtd' document type inside the same folder. They get processed. The webservice checks if the document is valid according to the DTD mentioned in the document, in this case 'sales.dtd'. It's a valid document says that DTD, so processing starts. But the service now crashes every 10minutes, trying to get the oldest file from the folder, because the content is not what it expected. Had the program validated according to the 'customers' dtd and not the DTD mentioned inside of the document, things wouldve gone differently, and it would've been easy to log a message and move the file.

Base URI's and Streams, consequences for schemas

See included project: StreamsSchemaSet, EmbeddedResourceResolver

Streams don't have the information that tells the reader what filename or uri it's handling. You just have a stream of data; where it came from, you have no way of knowing. It could be one you got from a HTTP request, a memorystream, or one on your local disk. ***A stream doesn't know 'where' it is***. But the 'where' is important in some scenario's. For example, the ***SchemaSet class determines duplicate Schemas by their filename or URI***. If you're using streams, the ***schema can't tell the schemaset what its URI is.***

Suppose we compile a schema set. We want to validate different types of documents with it, and parts of the schema's are re-used between formats.
Reading the XmlSchema's first through a stream and then adding them to the schemaset does not work properly:

```
XmlSchemaSet set = new XmlSchemaSet();
XmlSchema schema = new XmlSchema();
using (FileStream fs = new FileStream("sale.xsd", FileMode.Open))
{
    // reading the schema with default settings resolves the included schemas
    // properly, and does not trigger duplicates within the same root schema
    schema = XmlSchema.Read(fs, (o, e) => Console.WriteLine(e.Message));
}
set.Add(schema);
using (FileStream fs = new FileStream("client.xsd", FileMode.Open))
{
    // however, adding an already included child schema twice gives a conflict
    schema = XmlSchema.Read(fs, (o, e) => Console.WriteLine(e.Message));
}
set.Add(schema);
set.Compile();
```

What happens?

- The root schema, sale.xsd is loaded. The Schema does not have a base uri, because its loaded via a stream.
- The included schema's inside are resolved with an XmlUrlResolver per default, so they DO have a filename known to the Schema
- A file we also want to use as root schema but that is already included via 'sale.xsd' cannot be added: the stream in the example does not know the location, and so SchemaSet cannot determine it's the same schema file as the one already loaded. The result is an exception.

Instead, if we hand the SchemaSet a filename, included schemas are resolved as before and all the schema's know their location. The result is that the second root schema isn't loaded twice, and there is no exception because of the duplicate declaration.

```
XmlSchemaSet set = new XmlSchemaSet();
XmlSchema schema = new XmlSchema();
// handing the SchemaSet filenames solves the problem
set.Add(null, "sale.xsd");
set.Add(null, "client.xsd");
set.Compile();
```

This poses a problem if you wish to get your schema's for example from embedded resources, so normal users of your program will have difficulties changing the schema your program uses to validate input with. Embedded resources only know streams, not filenames, so you have to work around it. You need to schema to know a unique URI for the stream, and the best way to do this is to ***use your own XmlResolver that resolves filenames or URI's to the embedded resources into streams***, like this (you could it for DTD's too):

```
/// <summary>
/// Decorator class used to resolve xsd's from the embedded resources
/// Limited capabilities: use a type in the same folder (namespace) as the
resource
/// to locate, will ignore all folders etc., will just look for
/// the filename in the folder of the type T
/// </summary>
/// <typeparam name="T">Type to locate resources</typeparam>
public class EmbeddedResourceUrlResolver<T> : XmlResolver
{
    private readonly XmlResolver _resolver;
    private readonly string[] _schemes;

    /// <summary>
    /// Decorating constructor
    /// </summary>
    /// <param name="resolver"></param>
    /// <param name="schemes">array of Uri.UriScheme* constant entries</param>
    public EmbeddedResourceUrlResolver(XmlResolver resolver, params string[]
schemes)
    {
        if (resolver == null) throw new ArgumentNullException("resolver");
        _resolver = resolver;
        _schemes = schemes;
    }

    /// <summary>
    /// Sets the credentials to use for resolving
    /// </summary>
    public override System.Net.ICredentials Credentials
    {
        set { _resolver.Credentials = value; }
    }

    /// <summary>
    /// Gets the <see cref="Stream" /> referenced by the uri
    /// </summary>
    /// <param name="absoluteUri"></param>
    /// <param name="role"></param>
    /// <param name="ofObjectToReturn"></param>
    /// <returns>a <see cref="Stream"/></returns>
    public override object GetEntity(Uri absoluteUri, string role, Type
ofObjectToReturn)
    {
        if (_schemes.Contains(absoluteUri.Scheme))
        {
            string filename = Path.GetFileName(
                absoluteUri.ToString());
            Type locatorType = typeof(T);
            Stream stream = locatorType
```

```
                .Assembly
                .GetManifestResourceStream(locatorType, filename);
            if (stream == null)
            {
                try
                {
                    stream = (Stream)_resolver.GetEntity(absoluteUri, role,
ofObjectToReturn);
                }
                catch (MissingManifestResourceException missingException)
                {
                    throw new MissingManifestResourceException(
                        string.Format(
                            "Embedded resource {0} could not be resolved using
type {1}.  Full request was: {2}.",
                            filename, typeof (T), absoluteUri),
missingException);
                }
                catch (IOException exception)
                {
                    throw new MissingManifestResourceException(
                        string.Format(
                            "Embedded resource {0} could not be resolved using
type {1}.  Full request was: {2}.",
                            filename, typeof (T), absoluteUri), exception);
                }
                if (stream == null)
                {
                    throw new MissingManifestResourceException(
                    string.Format("Embedded resource {0} could not be resolved
using type {1}.  Full request was: {2}.",
                    filename, typeof(T), absoluteUri));
                }
            }
            return stream;

        }
        return _resolver.GetEntity(absoluteUri, role, ofObjectToReturn);

    }
}
```

You can then hand the EmbeddedResourceResolver a filename and type (which is used to get the namespace), and embedded resources will be used (note that you could rewrite it, to pass on requests it can't find, onto the decorated resolver). Disabling the resolver again would load them from file in the folder next to the application executeable.

Here's some example code how you can use this class:

```
XmlSchemaSet set = new XmlSchemaSet();
XmlSchema schema = new XmlSchema();
set.XmlResolver = new EmbeddedResourceUrlResolver<SchemaLocation>(new
XmlUrlResolver(), Uri.UriSchemeFile, Uri.UriSchemeHttp);
set.Add(null, "sale.xsd");
set.Add(null, "client.xsd");
set.Compile();
```

These were my most common pitfalls. Now let's look at what has changed in .NET 4.0 so you can take on migrating from .NET 3.5 to.NET 4.0 XML with confidence.

What went obsolete in .NET 4.0?

As mentioned before: no more ProbihitDtd with XmlReaders, and no more Evidence to use in Xml Serialization, and no more XmlValidatingReader.
The following classes/methods/overloads have been made obsolete that I know of (in bold):

```
    public class XmlConvert {
        public static String ToString(DateTime value);
    }

    public sealed class XmlReaderSettings {
        public Boolean ProhibitDtd { get; set; }
    }

    public class XmlTextReader : XmlReader, IXmlLineInfo, IXmlNamespaceResolver
{
        public Boolean ProhibitDtd { get; set; }
    }

    public class XmlValidatingReader

    public class XmlSerializer {
        public XmlSerializer(Type type, XmlAttributeOverrides overrides, Type[]
extraTypes, XmlRootAttribute root, String defaultNamespace, String location,
Evidence evidence);

        public static XmlSerializer[] FromMappings(XmlMapping[] mappings,
Evidence evidence);
    }

    public class XmlSerializerFactory {
        public XmlSerializer CreateSerializer(Type type, XmlAttributeOverrides
overrides, Type[] extraTypes, XmlRootAttribute root, String defaultNamespace,
String location, Evidence evidence);
    }
```

Changes in System.XML and related namespaces

What changed in the System.XML namespace?

XmlConvert

ToString(datetime) has been made obsolete. ***A number of tests for characters and XML strings have been added to the XmlConvert class***:

```
    public class XmlConvert {
        [ObsoleteAttribute("Use XmlConvert.ToString() that takes in
XmlDateTimeSerializationMode")]
        public static String ToString(DateTime value);

        public static Boolean IsNCNameChar(Char ch);
        public static Boolean IsPublicIdChar(Char ch);
        public static Boolean IsStartNCNameChar(Char ch);
        public static Boolean IsWhitespaceChar(Char ch);
        public static Boolean IsXmlChar(Char ch);
        public static Boolean IsXmlSurrogatePair(Char lowChar, Char highChar);
        public static String VerifyPublicId(String publicId);
        public static String VerifyWhitespace(String content);
        public static String VerifyXmlChars(String content);
    }
```

IsNCNameChar: Checks whether the passed-in character is a valid non-colon character type.
IsPublicIdChar: Returns the passed-in character instance if the character in the argument is a valid public id character, otherwise Nothing.
IsStartNCNameChar: Checks if the passed-in character is a valid Start Name Character type.
IsWhitespaceChar: Checks if the passed-in character is a valid XML whitespace character.
IsXmlChar: Checks if the passed-in character is a valid XML character.
IsXmlSurrogatePair: Checks if the passed-in surrogate pair of characters is a valid XML character.
VerifyPublicId: Returns the passed in string instance if all the characters in the string argument are valid public id characters.
VerifyTOKEN: Verifies that the string is a valid token according to the W3C XML Schema Part2: Datatypes recommendation.
VerifyWhitespace: Returns the passed-in string instance if all the characters in the string argument are valid whitespace characters.
VerifyXmlChars: Returns the passed-in string if all the characters and surrogate pair characters in the string argument are valid XML characters, otherwise Nothing.

XmlReader

HasValue went from abstract to virtual, so has received a default implementation

```
    public abstract class XmlReader : IDisposable {
        public (abstract) virtual Boolean HasValue { get; }
    }
```

XmlReader.Create now also have overloads to programmatically set the base uri for streams or readers (anything that doesn't have a base uri itself, like filenames or uri's do).

XmlReaderSettings

ProbititDtd has been replaced by DtdProcessing

This new setting also allows parsing documents that do have a DTD, but to ignore it completely.

XmlResolver

See included project: NonStreamXmlResolver

This abstract class now has a SupportsType function

```
    public abstract class XmlResolver {
        public virtual Boolean SupportsType(Uri absoluteUri, Type type);
    }
```

This new feature allows something else to be returned by an XmlResolver, like an XDocument. The following is an example of a class that does just that:

```
public class XDocumentUrlResolver: XmlResolver
    {
        XmlResolver _resolver;
        public XDocumentUrlResolver(XmlResolver wrappedResolver)
        {
            _resolver = wrappedResolver;
        }

        public override System.Net.ICredentials Credentials
        {
            set { _resolver.Credentials = value; }
        }

        public override bool SupportsType(Uri absoluteUri, Type type)
        {
            if (type == typeof(XDocument)) return true;
            if (_resolver != null) return _resolver.SupportsType(absoluteUri,
type);
            return false;
        }
        public override object GetEntity(Uri absoluteUri, string role, Type
ofObjectToReturn)
        {
            if (_resolver != null && ofObjectToReturn == typeof(XDocument))
            {
                XDocument doc =
XDocument.Load((Stream)_resolver.GetEntity(absoluteUri, role, typeof(Stream)),
LoadOptions.PreserveWhitespace);
                return doc;
            }
            if (_resolver != null) return _resolver.GetEntity(absoluteUri,
role, ofObjectToReturn);
            throw new NotSupportedException("Can't resolve without an
underlying resolver");
        }
    }
```

Ofcourse, the use of this new ability is limited: ***XmlReaders won't suddenly return XDocuments***, it just means you can build your own frameworks with similar resolver mechanics and have more re-use out of resolver classes.

XmlTextReader

ProhibitDtd was replaced by DtdProcessing to make it consistent with XmlReaderSettings.

```
    public class XmlTextReader : XmlReader, IXmlLineInfo, IXmlNamespaceResolver
{
        [ObsoleteAttribute("Use DtdProcessing property instead.")]
        public Boolean ProhibitDtd { get; set; }

        public DtdProcessing DtdProcessing { get; set; }
    }
```

XmlUrlResolver

See included project: CachePolicy

XmlUrlResolver now has ***write-only*** properties for ***cache policy and proxy***

```
    public class XmlUrlResolver : XmlResolver {
        public RequestCachePolicy CachePolicy { set; }
        public IWebProxy Proxy { set; }
    }
```

Note that this is for XmlUrlResolver, and not XmlResolver. One is a concrete implementation, the other the abstract base class for it. Inheriting from XmlResolver will not get you a CachePolicy or a proxy.
For more information on XmlUrlResolver see:
http://msdn.microsoft.com/enus/library/system.xml.xmlurlresolver.aspx.
You can set the proxy and cache policy as shown below:

```
WebProxy proxy = new WebProxy("http://localhost:8080");
RequestCachePolicy policy = new
RequestCachePolicy(RequestCacheLevel.BypassCache);
XmlUrlResolver resolver = new XmlUrlResolver();
resolver.Proxy = proxy;
resolver.CachePolicy = policy;
using (XmlReader reader = XmlReader.Create("people.xml", new XmlReaderSettings
{
    XmlResolver = resolver
}))
{
    XDocument.Load(reader);
}
```

XmlValidatingReader

This class was ***rendered obsolete***, making it consistent with the other XmlReaders. Use XmlReader.Create now like you would for every other XmlReader class (no more need to know the concrete implementation).

```
    [ObsoleteAttribute("Use XmlReader created by XmlReader.Create() method
using appropriate XmlReaderSettings instead.
http://go.microsoft.com/fwlink/?linkid=14202")]
```

```
    public class XmlValidatingReader : XmlReader, IXmlLineInfo,
IXmlNamespaceResolver {
        public override XmlReaderSettings Settings { get; }
    }
```

XmlWriterSettings

Now supports setting namespace handling mode; this was dicussed before.

```
    public sealed class XmlWriterSettings {
        public NamespaceHandling NamespaceHandling { get; set; }
    }
```

DtdProcessing and NamespaceHandling

Were discussed above

What changed in the System.Xml.Serialization namespace?

XmlSerializer, XmlSerializerFactory

All constructors and ways to create serializers taking ***Evidence*** as a parameter have been made obsolete, and replaced by a new one without this parameter.

```
    public class XmlSerializer {
        public XmlSerializer(Type type, XmlAttributeOverrides overrides, Type[]
extraTypes, XmlRootAttribute root, String defaultNamespace, String location,
Evidence evidence);

        public static XmlSerializer[] FromMappings(XmlMapping[] mappings,
Evidence evidence);

        public XmlSerializer(Type type, XmlAttributeOverrides overrides, Type[]
extraTypes, XmlRootAttribute root, String defaultNamespace, String location);
    }

    public class XmlSerializerFactory {
        public XmlSerializer CreateSerializer(Type type, XmlAttributeOverrides
overrides, Type[] extraTypes, XmlRootAttribute root, String defaultNamespace,
String location, Evidence evidence);

        public XmlSerializer CreateSerializer(Type type, XmlAttributeOverrides
overrides, Type[] extraTypes, XmlRootAttribute root, String defaultNamespace,
String location);
    }
```

What changed in the System.Xml.Xsl namespace?

XslCompiledTransform

A new overload for ***Transform*** was added:

```
    public sealed class XslCompiledTransform {
        public void Transform(IXPathNavigable input, XsltArgumentList
arguments, XmlWriter results, XmlResolver documentResolver);
    }
```

What changed in the System.Xml.Linq namespace

SaveOptions

SaveOptions now allows to specify to ***remove or keep duplicate namespaces.***
SaveOptions can be used as bitwise flags and use the |-operator

```
    [FlagsAttribute]
    public enum SaveOptions {
        OmitDuplicateNamespaces
    }
```

This is rougly the same as the NamespaceHandling enum in System.XML, but for System.XML.Linq and for writing documents only.

XDocument, XElement and XStreamingElement

See included project: XElementVsXStreamingElement

Overloads have been added for ***loading from and saving to streams***.

```
    public class XDocument : XContainer {
        public static XDocument Load(Stream stream, LoadOptions options);
        public static XDocument Load(Stream stream);
        public void Save(Stream stream);
        public void Save(Stream stream, SaveOptions options);
    }

    public class XElement : XContainer, IXmlSerializable {
        public static XElement Load(Stream stream, LoadOptions options);
        public static XElement Load(Stream stream);
        public void Save(Stream stream);
        public void Save(Stream stream, SaveOptions options);
    }

    public class XStreamingElement {
        public void Save(Stream stream);
        public void Save(Stream stream, SaveOptions options);
    }
```

These overloads use ***UTF-8*** readers and writers.

We'll show an example using streams and an XStreamingElement. XStreamingElements are like normal Xelements, but with one distinction: the contents are lazy-evaluated. That means that the LINQ query or IEnumerable inside is only calculated and processed when the value of the element itself is requested (note that this already happens when you add it to an XElement or XDocument!). This is especially handy for ***keeping your memory footprint low***, or preparing a large XML file and then turning out not to need it, or only need it in part. More information can be found at http://msdn.microsoft.com/en-us/library/system.xml.linq.xstreamingelement.aspx, but right now let's see it in practise.
We'll use a small helper function to write out when the LINQ query is enumerated:

```
private static XElement CreateLineElement(string line)
{
    Debug.WriteLine(line);
    return new XElement("line", line);
```

```
}
```

The following code creates an XElement based on LINQ.

```
string path = "line_input.txt";
Debug.WriteLine("start");
// the constructor of XElement enumerates
// the IEnumerable
XElement elem = new XElement("root",
        from line in ReadNextLine(path)
        select CreateLineElement(line));

Debug.WriteLine("after creation of XElement");
XDocument doc = new XDocument(elem);
Debug.WriteLine("after adding XElement to XDocument");
```

The output shows that XElement enumerates the LINQ query in its constructor:

```
start
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Etiam lorem velit, elementum a pellentesque nec, pharetra in ipsum.
Ut libero lorem, ultricies in auctor elementum, vestibulum sed lacus.
Mauris consectetur quam sit amet libero pretium ut dapibus libero ornare.
after creation of XElement
after adding XElement to XDocument
```

Next' we'll try the same, but with XStreamingElement:

```
string path = "line_input.txt";
Debug.WriteLine("start");
// the constructor of XStreamingElement does
// not enumerate the IEnumerable yet
XStreamingElement elem = new XStreamingElement("root",
        from line in ReadNextLine(path)
        select CreateLineElement(line));

Debug.WriteLine("after creation of XStreamingElement");

// XStreamingElement is enumerated when it is added
// to a non-streaming element or document
XDocument doc = new XDocument(elem);
Debug.WriteLine("after adding XStreamingElement to XDocument");
```

This yields the following output:

```
start
after creation of XStreamingElement
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Etiam lorem velit, elementum a pellentesque nec, pharetra in ipsum.
Ut libero lorem, ultricies in auctor elementum, vestibulum sed lacus.
Mauris consectetur quam sit amet libero pretium ut dapibus libero ornare.
after adding XStreamingElement to XDocument
```

This shows that the XStreamingElement's constructor does not enumerate the IEnumerable, but once it's added to an XDocument (or XElement) it will be enumerated.

Saving the XDocument then yields an UTF-8 XML document

```
MemoryStream memStream = new MemoryStream();
doc.Save(memStream, SaveOptions.DisableFormatting);
```

Conclusion:

- An XStreamingElement only generates its contents when the contents are requested (eg, if you need database access, keep the database open while its contents are not requested..)
- An XElement generates its contents at the constructor
- Adding an XStreamingElement to an XDocument or XElement generates its contents, so placing an XDocument full of XStreamingElements in case they won't be needed is pointless, they will get calculated at the documents constructor anyway.
- Streams overloads use UTF-8 encoding

XNode

A new overload to create a reader from an XNode, so it will also work with any derived class (eg XElement).

```
public abstract class XNode : XObject, IXmlLineInfo {
    public XmlReader CreateReader(ReaderOptions readerOptions);
}
```

Conclusion

Your .NET 3.5 XML code should still be compatible with the new .NET 4.0. There are new ways for handling DTD's, and it's good DTD's finally got some extra lovin' (the obsolete property will break builds if you treat warnings as errors however). Duplicate namespaces can now be handled gracefully. More XML readers are deprecated and now require the use of XmlReader.Create, but this was a general guideline since .NET 2.0 already. Some pitfalls still remain, making working with XML sometimes harder and more error-prone than it should be. The difference between XDocument with or without XmlReaders/Writers comes to mind, the same goes for XmlDocument. Documentation for, for example, XStreamingElement is still pretty basic.
I hope you enjoyed the read, and if you have questions or comments bring them on!
Oh, and thanks for reading all my ramblings (you made it to the end!)

---

[1] A few examples:
http://www.oracle.com/technology/pub/articles/wang-whitespace.html
http://www.ibm.com/developerworks/xml/library/x-diff/index.html

[2] Altova XML Spy does not adhere to this, but .NET 3.5 and .NET 4.0 default settings for XslCompiledTransform do – XslCompiledTransform is the class that's used to do XSL Transformations in .NET