XML Serialization in .NET

Venkat Subramaniam
venkats@durasoftcorp.com
http://www.durasoftcorp.com

Abstract

XML Serialization in .NET provides ease of development, convenience and efficiency. This article discusses the benefits of XML serialization and ways to configure the generated XML document. It goes further to discuss some of the issues and deficiencies of this process as well. This article assumes that the reader is familiar with XML format, XML Schema and C#.

Processing XML in .NET

The .NET framework has a number of classes to process XML documents. To start with, the features of the MSXML parser, which was a COM component, has now been moved into the .NET framework with more efficiency. The complete DOM API is implemented in the System.Xml namespace. XmlDocument is the class that represents a DOM document node and various classes like XmlElement, XmlAttribute, etc., represent the different types of nodes in the DOM API. While SAX is, so to say, a pull technology, a similar but more efficient push technology is introduced in .NET through the XmlReader class. The XmlReader allows you to process an XML document by instructing the parser to read and navigate serially through an XML document. The XmlReader provides a fast, read-only, forward-only access to an XML document. While these classes and APIs are significant, our focus in this article is on XML Serialization, and we will not discuss these classes further in this article.

A problem that will benefit

We got our first exposure to XML Serialization when we were developing an ASP.NET application. We wanted to gather some information from the user and keep it in XML format, so we could easily apply a style sheet to it and display the contents any time. The application did not warrant the use of any DBMS. The first approach we took was to start writing the XML document from the user specified information using the standard file I/O classes. Further, in order to fetch the information again for later modifications, we had to use an XML parser. Of course, using the parser to process the contents of the XML document is better than reading the contents by ourselves. However, the fact that we had to write each and every tag out was quite bothersome. We could have used the XMLWriter class provided in the System.Xml namespace to do that. But, wouldn’t it be nice if we can simply take the data from an object and write out an XML document and also perform the reverse operation of taking an XML document and converting it back into an object? Pretty soon we found out that this is exactly what XMLSerialization does. The process of transforming the contents of an object into XML format is called serialization, and the reverse process of transforming an XML document into a .NET object is called deserialization.

An example

Let us take a C# class shown below (it could be VB.NET or any other .NET language for that matter):

```
public class Car
{
     private int m_yearOfMake;

     public int yearOfMake
     {
          get { return m_yearOfMake; }
          set { m_yearOfMake = value; }
     }

     public override string ToString()
     {
          return "Car year: " + m_yearOfMake;
     }
}
```

Assume that we want to convert the information in an object of the above Car class into XML representation. Given below is a sample code that will let us either convert a Car object into an XML representation, or to create a Car object given a valid XML document.

```
class User
{
     [STAThread]
     static void Main(string[] args)
     {
          if (args.Length != 2)
          {
               Console.WriteLine("Usage XMLSerialization
                                   [r|s] filename");
               Console.WriteLine("r to read, s to save");
          }
          else
          {
               string fileName = args[1];
               if (args[0] == "r")
               {
                    System.Xml.Serialization.XmlSerializer
                         serializer =
                              new
                    System.Xml.Serialization.XmlSerializer(
                                   typeof(Car));
                    System.IO.FileStream stream =
                         new System.IO.FileStream(fileName,
                              System.IO.FileMode.Open);

                    Car obj = serializer.Deserialize(
                                   stream) as Car;

                    Console.WriteLine(obj);
               }
               else
               {
                    Car obj = new Car();
                    obj.yearOfMake = 2002;

                    System.Xml.Serialization.XmlSerializer
                         serializer =
                              new
                    System.Xml.Serialization.XmlSerializer(
                              typeof(Car));

                    System.IO.FileStream stream =
                         new System.IO.FileStream(fileName,
                              System.IO.FileMode.Create);

                    serializer.Serialize(stream, obj);
               }
          }
     }
}
```

Note that we first create an object of XmlSerializer. The XMLSerializer takes an argument which is the Type reflection meta object of the Car class. We then call either the Serialize method or the Deserialize method on it. The Serialize method takes a FileStream and an object of Car as arguments, while the Deserialize method takes the FileStream and returns an object of Car.

Running the program as

```
XMLSerialization s car.xml
```

creates an XML document car.xml as shown below:

```
<?xml version="1.0"?>
<Car xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <yearOfMake>2002</yearOfMake>
</Car>
```

Notice that Car is the root element name and the yearOfMake field of the Car became a child element of the root element. *By default, each public field and public property of an object is transformed into an XML element.* What if we want the yearOfMake to appear as an attribute and not as a child element? This is very easy to achieve. Let’s modify the Car class as follows:

```
[System.Xml.Serialization.XmlRoot("Automobile")]
public class Car
{
     private int m_yearOfMake;

     [System.Xml.Serialization.XmlAttribute("Year")]
     public int yearOfMake
     {
          get { return m_yearOfMake; }
          set { m_yearOfMake = value; }
     }

     public override string ToString()
     {
          return "Car year: " + m_yearOfMake;
     }
}
```

The XmlRoot attribute indicates that the root element’s name should be Automobile instead of Car. The XmlAttribute attribute indicates that the public property yearOfMake should appear as an attribute, with name Year, instead of appearing as a child element like it did in the previous case.

No change is required to the User class. Simply running the program again produces the following car.xml document:

```
<?xml version="1.0"?>
<Automobile xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

     Year="2002" />
```

Note that the yearOfMake now appears as an attribute and the root element’s name is Automobile - thanks to the attributes that we set on the class Car and its yearOfMake property.

How about aggregation?

What if the Car has an aggregation relationship to an Engine? Here is the related Engine class and the Car class modified to do just that:

```
public class Engine
{
     private int m_power;

     [System.Xml.Serialization.XmlAttribute]
     public int Power
     {
          get { return m_power; }
          set { m_power = value; }
     }

     public override string ToString()
     {
          return "power " + m_power;
     }
}

[System.Xml.Serialization.XmlRoot("Automobile")]
public class Car
{
     private int m_yearOfMake;
     private Engine m_engine;

     [System.Xml.Serialization.XmlAttribute("Year")]
     public int yearOfMake
     {
          get { return m_yearOfMake; }
          set { m_yearOfMake = value; }
     }

     public Engine TheEngine
     {
          get { return m_engine; }
          set { m_engine = value; }
     }

     public Car()
     {
          m_yearOfMake = 2002;
          m_engine = new Engine();
          m_engine.Power = 150;
     }

     public override string ToString()
     {
          return "Car year: " + m_yearOfMake +
               " with Engine " + m_engine;
     }
}
```

No change is again required to the User class. Simply running the program again creates the following car.xml document:

```
<?xml version="1.0"?>
<Automobile xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          Year="2002">
  <TheEngine Power="150" />
</Automobile>
```

You may modify the car.xml to change the power of the Engine to 200 and the year of make to 2003 and run the program as follows:

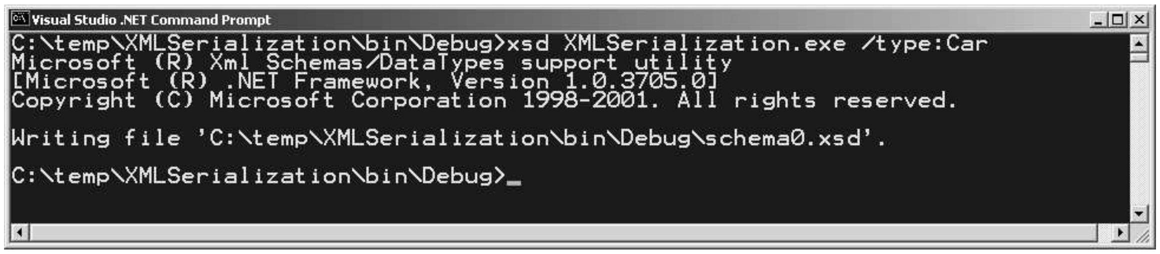
```
XMLSerialization r car.xml
```

The output of the program will be:

```
Car year: 2003 with Engine power 200
```

XML Schema generation

A tool, xsd.exe, is provided with .NET framework to generate an XML schema from a given .NET class. This tool may also be used to generate a .NET class give an XML schema. Let’s try it out on our Car class by typing the following from a Visual Studio .NET command prompt:



```
C:\temp\XMLSerialization\bin\Debug>xsd XMLSerialization.exe /type:Car
Microsoft (R) Xml Schemas/DataTypes support utility
[Microsoft (R) .NET Framework, Version 1.0.3705.0]
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.

Writing file 'C:\temp\XMLSerialization\bin\Debug\schema0.xsd'.

C:\temp\XMLSerialization\bin\Debug>_
```

The xsd.exe was asked to generate an XML schema for the Car class. The generated schema is shown below:

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
     <xs:element name="Automobile" nillable="true"
          type="Car" />
     <xs:complexType name="Car">
          <xs:sequence>
               <xs:element minOccurs="0" maxOccurs="1"
                    name="TheEngine" type="Engine" />
          </xs:sequence>
          <xs:attribute name="Year" type="xs:int" />
```

```
	</xs:complexType>
	<xs:complexType name="Engine">
		<xs:attribute name="Power" type="xs:int" />
	</xs:complexType>
</xs:schema>
```

You may observe from the schema that the format of the generated XML document matches with the structure specified by this XML Schema.

The xsd.exe also has options to generate a .NET class given an XML schema. This comes in handy if you need to receive an XML document from another application and process it in your application. Instead of parsing the XML document using one of the APIs like DOM, you can simply deserialize the XML document into a .NET object. This saves quite a bit of effort in receiving and processing XML documents in your application.

What about collections?

Let's extend the above example to a collection of Cars. We will also add a namespace to the generated XML document.

In the Car class, we add a constructor as shown below:

```
	public Car(int year)
	{
		m_yearOfMake = year;
		m_engine = new Engine();
		m_engine.Power = 150;
	}
```

We then write a Shop class as shown below:

```
[System.Xml.Serialization.XmlRoot("AutoShop",
	Namespace="http://www.auto.com")]
public class Shop
{
	private Car[] m_cars = new Car[3];

	[System.Xml.Serialization.XmlArray("Automobiles"),
	System.Xml.Serialization.XmlArrayItem(typeof(Car))]
	public Car[] cars
	{
		get { return m_cars; }
		set { m_cars = value; }
	}
}
```

Notice how we have set the namespace for the xml document in the XmlRoot attribute. Also, we have indicated that we are dealing with an array and the array items are of type Car using the XmlArray and XmlArrayItem attributes.

The User.cs is modified to create and serialize a Shop object as follows:

```
          Shop aShop = new Shop();
          aShop.cars[0] = new Car(2000);
          aShop.cars[1] = new Car(2001);
          aShop.cars[2] = new Car(2002);

          System.Xml.Serialization.XmlSerializer serializer
               = new
                  System.Xml.Serialization.XmlSerializer(
                         typeof(Shop));

          System.IO.FileStream stream =
               new System.IO.FileStream(fileName,
               System.IO.FileMode.Create);

          serializer.Serialize(stream, aShop);
```

Running the XMLSerialization produces the following car.xml document now:

```
<?xml version="1.0"?>
<AutoShop xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns="http://www.auto.com">
  <Automobiles>
    <Car Year="2000">
      <TheEngine Power="150" />
    </Car>
    <Car Year="2001">
      <TheEngine Power="150" />
    </Car>
    <Car Year="2002">
      <TheEngine Power="150" />
    </Car>
  </Automobiles>
</AutoShop>
```

What’s the catch?

As we can see, the XML Serialization mechanism in .NET is pretty powerful. The amount of effort required in transforming between a .NET object and its corresponding XML representation is minimal. The classes related to serialization and processing have been implemented as part of the .NET class framework with at utmost efficiency. There is, however, one thing undesirable. You may have already observed it in the above example. Let’s go back to the Car class once again. The Car has a reference to Engine. If we serialize the Car, the Engine is serialized along with it. However, a reference of type Engine, may refer to an object of the Engine class, or any class that is derived from the Engine class. Let TurboEngine be a class that inherits from the Engine class.

```
public class TurboEngine : Engine
{
     public TurboEngine() { Power = 300; }

     public override string ToString()
     {
          return "Turbo Engine " + base.ToString();
     }
}
```

We now modify the User.cs to use a TurboEngine for one of the Cars:

```
          Shop aShop = new Shop();
          aShop.cars[0] = new Car(2000);
          aShop.cars[1] = new Car(2001);
          aShop.cars[2] = new Car(2002);
          aShop.cars[2].TheEngine = new TurboEngine();

          System.Xml.Serialization.XmlSerializer serializer
               = new
                  System.Xml.Serialization.XmlSerializer(
                         typeof(Shop));

          System.IO.FileStream stream =
               new System.IO.FileStream(fileName,
               System.IO.FileMode.Create);

          serializer.Serialize(stream, aShop);
```

Let’s run the program again to generate the car.xml. This time, we get an exception: *“Unhandled Exception: System.InvalidOperationException: There was an error gener ting the XML document. ---> System.InvalidOperationException: The type XMLSeria ization.TurboEngine was not expected. Use the XmlInclude or SoapInclude attribute to specify types that are not known statically.…”*

The serialization process does not deal with inheritance hierarchy in a smooth way. For this to work, you will have to indicate that the engine reference may refer to an object of Engine or TurboEngine as follows:

```
public class Car
{
     private int m_yearOfMake;
     private Engine m_engine;

     [System.Xml.Serialization.XmlElement(typeof(Engine)),
     System.Xml.Serialization.XmlElement(
               typeof(TurboEngine))]
     public Engine TheEngine
     {
          get { return m_engine; }
          set { m_engine = value; }
     }
…
```

Running the program again produces the following car.xml document:

```
<?xml version="1.0"?>
<AutoShop xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://www.auto.com">
  <Automobiles>
    <Car Year="2000">
      <Engine Power="150" />
    </Car>
    <Car Year="2001">
      <Engine Power="150" />
    </Car>
    <Car Year="2002">
      <TurboEngine Power="300" />
    </Car>
  </Automobiles>
</AutoShop>
```

While the above fix works, we have completely violated the Open-Closed Principle. The code is not extensible to adding new types of Engine. If we decide to add another class which inherits from Engine or TurboEngine, we will have to modify the Car class. This, to say the least is undesirable. This seems to be the only significant limitation and hope the future revisions of the XML serialization mechanism will address this.

Conclusion

In this article, we have presented the details of the support provided for XML Serialization in .NET. XML Serialization allows us to transform between a .NET object and an XML representation. This makes it easier to exchange XML documents between applications. After all, in an object-oriented application we deal with objects and it makes a lot of sense to be able to generate an object from an XML representation and vice versa. This is simply achieved in .NET using the xsd.exe tool. Only public fields and public properties are transformed into XML representation. But, the representation can be controlled and one can decide on the names of elements, attributes and also whether an entity should be represented as a child element or as an attribute. Finally, one major limitation seems to be the non-extensible nature of the mechanism. It does not support inherited types automatically and requires type declaration for each type of inherited class.

References

1. MSDN http://msdn.microsoft.com