Read XML Files using LINQ to XML and Extension Methods

In previous blog posts I have discussed how to use XML files to store data in your applications. I showed you how to read those XML files from your project and get XML from a WCF service. One of the problems with reading XML files is when elements or attributes are missing. If you try to read that missing data, then a null value is returned. This can cause a problem if you are trying to load that data into an object and a null is read. This blog post will show you how to create extension methods to detect null values and return valid values to load into your object.

The XML Data

An XML data file called Product.xml is located in the \Xml folder of the Silverlight sample project for this blog post. This XML file contains several rows of product data that will be used in each of the samples for this post. Each row has 4 attributes; namely ProductId, ProductName, IntroductionDate and Price.

```
<Products>
  <Product ProductId="1"
           ProductName="Haystack Code Generator for .NET"
           IntroductionDate="07/01/2010"  Price="799" />
  <Product ProductId="2"
           ProductName="ASP.Net Jumpstart Samples"
           IntroductionDate="05/24/2005"  Price="0" />
  ...
  ...
</Products>
```

The Product Class

Just as you create an Entity class to map each column in a table to a property in a class, you should do the same for an XML file too. In this case you will create a Product class with properties for each of the attributes in each element of product data. The following code listing shows the Product class.

```
public class Product : CommonBase
{
  public const string XmlFile = @"Xml/Product.xml";

  private string _ProductName;
  private int _ProductId;
  private DateTime _IntroductionDate;
  private decimal _Price;

  public string ProductName
  {
    get { return _ProductName; }
    set {
      if (_ProductName != value) {
        _ProductName = value;
        RaisePropertyChanged("ProductName");
      }
    }
  }

  public int ProductId
  {
    get { return _ProductId; }
    set {
      if (_ProductId != value) {
        _ProductId = value;
        RaisePropertyChanged("ProductId");
      }
    }
  }

  public DateTime IntroductionDate
  {
    get { return _IntroductionDate; }
    set {
      if (_IntroductionDate != value) {
        _IntroductionDate = value;
        RaisePropertyChanged("IntroductionDate");
      }
    }
  }

  public decimal Price
  {
    get { return _Price; }
    set {
      if (_Price != value) {
        _Price = value;
        RaisePropertyChanged("Price");
      }
    }
  }
}
```

NOTE: The CommonBase class that the Product class inherits from simply implements the INotifyPropertyChanged event in order to inform your XAML UI of

any property changes. You can see this class in the sample you download for this blog post.

Reading Data

When using LINQ to XML you call the Load method of the XElement class to load the XML file. Once the XML file has been loaded, you write a LINQ query to iterate over the “Product” Descendants in the XML file. The “select” portion of the LINQ query creates a new Product object for each row in the XML file. You retrieve each attribute by passing each attribute name to the Attribute() method and retrieving the data from the “Value” property. The Value property will return a null if there is no data, or will return the string value of the attribute. The Convert class is used to convert the value retrieved into the appropriate data type required by the Product class.

```
private void LoadProducts()
{
  XElement xElem = null;

  try
  {
    xElem = XElement.Load(Product.XmlFile);

    // The following will NOT work if you have missing attributes
    var products =
        from elem in xElem.Descendants("Product")
        orderby elem.Attribute("ProductName").Value
        select new Product
        {
          ProductId = Convert.ToInt32(
            elem.Attribute("ProductId").Value),
          ProductName = Convert.ToString(
            elem.Attribute("ProductName").Value),
          IntroductionDate = Convert.ToDateTime(
            elem.Attribute("IntroductionDate").Value),
          Price = Convert.ToDecimal(elem.Attribute("Price").Value)
        };

    lstData.DataContext = products;
  }
  catch (Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
}
```

This is where the problem comes in. If you have any missing attributes in any of the rows in the XML file, or if the data in the ProductId or IntroductionDate is not of the

appropriate type, then this code will fail! The reason? There is no built-in check to ensure that the correct type of data is contained in the XML file. This is where extension methods can come in real handy.

Using Extension Methods

Instead of using the Convert class to perform type conversions as you just saw, create a set of extension methods attached to the XAttribute class. These extension methods will perform null-checking and ensure that a valid value is passed back instead of an exception being thrown if there is invalid data in your XML file.

```
private void LoadProducts()
{
  var xElem = XElement.Load(Product.XmlFile);

  var products =
      from elem in xElem.Descendants("Product")
      orderby elem.Attribute("ProductName").Value
      select new Product
      {
        ProductId = elem.Attribute("ProductId").GetAsInteger(),
        ProductName = elem.Attribute("ProductName").GetAsString(),
        IntroductionDate =
           elem.Attribute("IntroductionDate").GetAsDateTime(),
        Price = elem.Attribute("Price").GetAsDecimal()
      };

  lstData.DataContext = products;
}
```

Extension Methods

To create an extension method you will create a class with any name you like. The code below shows a class named XmlExtensionMethods. This listing just shows a couple of the available methods such as GetAsString and GetAsInteger. These methods are just like any other method you would write except when you pass in the parameter you prefix the type with the keyword “this”. This lets the compiler know that it should add this method to the class specified in the parameter.

```
public static class XmlExtensionMethods
{
  public static string GetAsString(this XAttribute attr)
  {
    string ret = string.Empty;

    if (attr != null && !string.IsNullOrEmpty(attr.Value))
    {
      ret = attr.Value;
    }

    return ret;
  }

  public static int GetAsInteger(this XAttribute attr)
  {
    int ret = 0;
    int value = 0;

    if (attr != null && !string.IsNullOrEmpty(attr.Value))
    {
      if(int.TryParse(attr.Value, out value))
        ret = value;
    }

    return ret;
  }

  ...
  ...
}
```

Each of the methods in the XmlExtensionMethods class should inspect the XAttribute to ensure it is not null and that the value in the attribute is not null. If the value is null, then a default value will be returned such as an empty string or a 0 for a numeric value.

Summary

Extension methods are a great way to simplify your code and provide protection to ensure problems do not occur when reading data. You will probably want to create more extension methods to handle XElement objects as well for when you use element-based XML. Feel free to extend these extension methods to accept a parameter which would be the default value if a null value is detected, or any other parameters you wish.

NOTE: You can download the complete sample code at my website. http://www.pdsa.com/downloads. Choose “Tips & Tricks”, then "Read XML Files using LINQ to XML and Extension Methods" from the drop-down.

Good Luck with your Coding,

Paul D. Sheriff