# Extending Linq to XML

Linq to XML is a great and neat API. However, having working with it for quite a long time, I have several concerns to it. Let’s consider them and then take a look at what we can do to make this API even better.

As an example XML we’ll parse this excerpt from Amazon search response:

<Item>

<ASIN>059035342X</ASIN>

<SmallImage>

<URL>http://ecx.images-amazon.com/images/I/51MU5VilKpL.\_SL75\_.jpg</URL>

<Height Units="pixels">75</Height>

<Width Units="pixels">51</Width>

</SmallImage>

<ItemAttributes>

<Author>J.K. Rowling</Author>

<EAN>9780590353427</EAN>

<ISBN>0439708184</ISBN>

<ListPrice>

<Amount>1099</Amount>

<CurrencyCode>USD</CurrencyCode>

<FormattedPrice>$10.99</FormattedPrice>

</ListPrice>

<PublicationDate>1999-10-01</PublicationDate>

<Title>Harry Potter and the Sorcerer's Stone (Book 1)</Title>

</ItemAttributes>

</Item>

If I have XElement variable with that XML, I would like to get its properties in the following way:

var title = item.Element("ItemAttributes").Element("Title").Value;

var imageUrl = new Uri(item.Element("SmallImage").Element("URL").Value);

var imageHeight = (int) item.Element("SmallImage").Element("Height");

var publicationDate = (DateTime) item.Element("ItemAttributes").Element("PublicationDate");

The problem here is that almost all nodes in this response are optional. I can assume that some of them, such as Author and Title, will always be specified for any book. Most of the others are really optional. Some books might not have images, others are out of stock and don’t have price, yet another available only for pre-order and don’t have publication date yet. Another problem is that conversion to value types might not go smooth and you’ll get FormatException.

So real-life defensive code will look like this:

var smallImageElement = item.Element("SmallImage");

if (smallImageElement != null)

{

var urlElement = smallImageElement.Element("URL");

if (urlElement != null && Uri.IsWellFormedUriString(urlElement.Value, UriKind.Absolute))

{

var imageUrl = new Uri(urlElement.Value);

}

}

It’s not nice at all. We can hide it into helper method or property. It is likely that this XML will be wrapped with a DTO type which will be responsible for proper XML parsing and it will expose properties for every value. Anyway, I would prefer to not write a lot of such ugly code.

I would like to write code that:

1. Can easily and safely be chained.
2. Mark elements and attributes as mandatory or optional. If mandatory element is missing, throw meaningful exception, but do not throw exceptions for missing optional nodes.
3. Get value of specified type. If element is optional, try get value and use some default if value is missing.

Below is re-written code that uses several extension methods:

var title = item.MandatoryElement("ItemAttributes").MandatoryElement("Title").Value;

var imageUrl = item.ElementOrEmpty("SmallImage").ElementOrEmpty("URL").Value(value => Uri.IsWellFormedUriString(value, UriKind.Absolute) ? new Uri(value) : null);

var imageHeight = item.ElementOrEmpty("SmallImage").ElementOrEmpty("Height").Value<int>(0);

var publicationDate = item.MandatoryElement("ItemAttributes").ElementOrEmpty("PublicationDate").Value<DateTime>();

This code doesn’t require any checks for null or catching of FormatExceptions.

However, if mandatory element is missing, the MandatoryElement method will throw XmlException with message “The element 'Item' doesn't contain mandatory child element 'ItemAttributes'.”. That is exactly what we want – detect that element we’re expecting to be there is missing. On the other hand, ElementOrEmpty method will newer throw exceptions. If element is missing (e.g. Element method returns null), it will create and return empty element with specified name so chaining can be continued (the NullObject pattern).

The Value methods also might throw XmlException like this: “The element 'Amount' has value 'unknown' which cannot be converted to the value of type 'int'.” which has much more context information (the original FormatException is included as inner exception). Error handling is made in a way that all exceptions have XmlException type and include necessary context information for easy location and reproduction of the occurred problem.

Let’s take a look at some extension methods implementation.

public static XElement MandatoryElement(this XElement element, XName name)

{

XElement childElement = element.Element(name);

if (childElement == null)

{

throw new XmlException(string.Format("The element '{0}' doesn't contain mandatory child element '{1}'.", element.Name, name));

}

return childElement;

}

The MandatoryElement just checks result of Element method and throws exception if null was returned. Nothing fancy, it just gets the null value handling out of your parsing logic.

The ElementOrEmpty is similarly simple:

public static XElement ElementOrEmpty(this XElement element, XName name)

{

if (element != null)

{

XElement childElement = element.Element(name);

return childElement ?? new XElement(name);

}

return new XElement(name);

}

The Value<T> method with custom convert function:

public static T Value<T>(this XElement element, Func<string, T> convert)

{

if (element == null)

{

return default(T);

}

return convert(element.Value);

}

In this case, the error handling of incorrect values is the responsibility of the convert method as demonstrated above.

The last method is the Value<T> method that parses a lot of pre-defined value types and enum values:

public static T Value<T>(this XElement element, T defaultValue = default(T)) where T : struct, IConvertible

{

if (element == null || string.IsNullOrEmpty(element.Value))

{

return defaultValue;

}

string value = element.Value;

try

{

Type typeOfT = typeof(T);

if (typeOfT.IsEnum)

{

return (T)Enum.Parse(typeOfT, value, ignoreCase: true);

}

return (T)Convert.ChangeType(value, typeof(T));

}

catch (Exception ex)

{

throw new XmlException(string.Format("The element '{0}' has value '{1}' which cannot be converted to the value of type '{2}'.", element.Name, element.Value, typeof(T).Name), ex);

}

}

This method allows parsing of int, uint, byte, sbyte, short, ushort, char, long, ulong, float, double, decimal, bool or DateTime values. Moreover, it can handle custom enumerations, including flag enumerations. Suppose, we have Color enum:

[Flags]

private enum Colors

{

None = 0,

Red = 1,

Green = 2,

Blue = 4

}

Having that, you can use Value<T> method to deserialize enum values:

var enumElement = new XElement("color", Colors.Red | Colors.Green); // Value = "Red, Green"

Colors colors = enumElement.Value<Colors>();

The project has several more extension methods which you may find useful.

XAttribute MandatoryAttribute(this XElement element, XName name);

XAttribute AttributeOrEmpty(this XElement element, XName name, string defaultValue = null);

T Value<T>(this XAttribute attribute, T defaultValue = default(T));

T Value<T>(this XAttribute attribute, Func<string, T> convert);

void AssertName(XElement element, XName expectedName);

XElement NewOptionalXElement(XName name, object value, object defaultValue = null);

XAttribute NewOptionalXAttribute(XName name, object value, object defaultValue = null);

As you see, with a little bit of adjusting of LINQ to XML API it becomes even better. Feel free to use these extensions in your projects. Happy coding!