XLinq
.NET Language Integrated Query for XML Data

Punit Ganshani

About Author:
Punit Ganshani is a software professional working for a IT MNC, and holds good command over .NET and Sharepoint. He has authored on several topics on C, VB6.0 and .NET. He is available on www.ganshani.com

XML seems to have changed the way development happens. It has got adoption for formatting of data in Office files, in configuration files and in database. For a developer, XML is, yet, tough to be worked upon. There are various reasons to it: not very strong APIs, and DOM specific objects such as XQuery, and XPath which have a steep learning curve. To lessen the time required to master XML and to capitalize the benefits of .NET fScottework, Microsoft introduced a new feature in their .NET FScottework 3.5 in November, 2007: XLinq

What you take away from this whitepaper is a basic understanding of XLinq, and how it can be used to perform complex tasks. Wherever possible, we would compare XLinq with W3C DOM.

XLinq: Its Structure

XLinq stands for Language Integrated Query for XML, a mechanism to take advantage of Standard Query Operators (and not DOM specific Object Models) and yet leverage the power of XQuery, XPath into .NET. XLinq introduces some new classes in its model and the hierarchy appears as shown in **Figure 1: XLinq Classes**

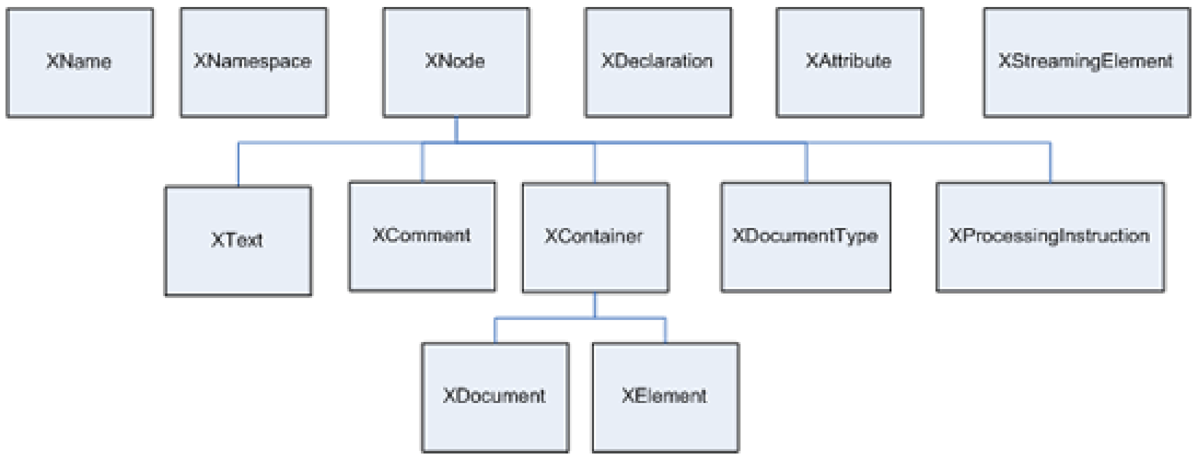


Figure 1: XLinq Classes

XLinq: Creating a XML file

Let us create a XML file to have following output:

```
 <Records>
     <Employee>
        <name>John</name>
        <phone type="mobile">9881070204</phone>
        <phone type="home">27471111</phone>
  </Employee>
</Records>
```

Below is the snippet to generate this file using traditional XmlDocument:

```
        private static void CreateXML()
        {
            XmlDocument doc      = new XmlDocument();
            XmlElement records   = doc.CreateElement("Records");
            XmlElement employee = doc.CreateElement("Employee");

            XmlElement name = doc.CreateElement("name");
            name.InnerText = "Punit";
            XmlElement phone1 = doc.CreateElement("phone");
            phone1.SetAttribute("type", "mobile");
            phone1.InnerText = "9881070204";

            XmlElement phone2 = doc.CreateElement("phone");
            phone2.SetAttribute("type", "home");
            phone2.InnerText = "27471111";

            employee.AppendChild(name);
            employee.AppendChild(phone1);
            employee.AppendChild(phone2);
            records.AppendChild(employee);
            doc.AppendChild(records);

            doc.Save(@"C:\\Punit Ganshani\\Employee.xml");
        }
```

With XLinq, these 17 lines of code can be reduced to:

```
            XElement employee =
                new XElement("Records",
                    new XElement("Employee",
                        new XElement("name", "Punit"),
                        new XElement("phone", "9881070204",
                            new XAttribute("type", "mobile")),
                        new XElement("phone", "27471111",
                            new XAttribute("type", "home"))
                ));
            employee.Save(@"C:\\Punit Ganshani\\EmployeeXLinq.xml");
```

By indenting (and squinting a bit) the code to construct the XML tree shows the structure of the underlying XML.

Let us analyze this:

```
XmlDocument doc = new XmlDocument();
XmlElement name = doc.CreateElement("name");
```

Becomes:

```
XElement name = new XElement("name");
```

XLinq avoids unnecessary layer of creating of XmlDocument object; hence allowing easier usage of nodes across documents.

```
public XElement(XName name, paScotts object[] contents)
```

The contents are very flexible, supporting any type of object that is legitimate child of XElement. These objects can be type-casted to string, bool, bool?, int, int?, uint, uint?, long, long?, ulong, ulong?, float, float?, double, double?, decimal, decimal?, DateTime, DateTime?, TimeSpan, TimeSpan?, and GUID, GUID?

```
string name = (string) employee.Element("name");
```

PaScotteters can also contain objects of XLinq classes:

1. XElement (to have an inner/child node)
2. XAttribute (to have a attribute)
3. XProcessingInstruction or XComment (to add comments)
4. XText (which acts like CData)

XLinq also allows us to retrieve data from functions or variable names like:

```
XElement employee =
     new XElement("Records",
         new XElement("Employee",
             new XElement("name", sName),
             new XElement("phone", getPhone(sName),
                 new XAttribute("type", "mobile")),
             new XElement("phone", getHomePhone(sName),
                 new XAttribute("type", "home"))
     ));
```

Here functions **getPhone**() and **getHomePhone**() may return any type of data and XLinq will automatically store it in XElement.

XLinq: Creating XML from a class object

In most of the business scenarios, we encapsulate data of entities into class objects. Say for example, we have a class Department to store Department Name and Department ID.

```
/*
    Class for the XML data coming from physical file
    Using the Automatic feature
*/
public class Departments
{
    public int DeptID { get; set; }
    public string DeptName { get; set; }
}

var departments = new[] {
            new Departments { DeptID=1, DeptName="Research" },
            new Departments { DeptID=2, DeptName="Development" },
            new Departments { DeptID=3, DeptName="Testing" }
};

XElement depts =
      new XElement("departments",
          from d in departments
          select new XElement("department",
                 new XElement("ID", d.DeptID),
                 new XElement("Name", d.DeptName))
                 );
```

This query returns the following XML in variable `depts`:

```
<departments>
  <department>
    <ID>1</ID>
    <Name>Research</Name>
  </department>
  <department>
    <ID>2</ID>
    <Name>Development</Name>
  </department>
  <department>
    <ID>3</ID>
    <Name>Testing</Name>
  </department>
</departments>
```

Implementing this tasks using W3C DOM model, requires lot of efforts in terms of coding & time! We will consider the "select", "from" clauses in the later part of this whitepaper. Now let us see how to load XML data in XElement object.

XLinq: Loading XML data in XElement

XLinq allows us to load XML into XElement from 3 sources: string, TextReader, or XMLReader. To load XML from a string, we can use the **Parse** Method. And to load from a URI, or using TextReader/XmlReader, we can use **Load** Method.

Method 1: From a string

```
string XML = @"<Records>
                    <Employee>
                        <name>John</name>
                        <phone type="mobile">9881070204</phone>
                        <phone type="home">27471111</phone>
                    </Employee>
                  </Records>";
XElement records = XElement.Parse(XML);
```

Method 2: From URI, TextReader or XmlReader

```
XElement recordsFromFile =
            XElement.Load(@"C:\Punit Ganshani\Employee.xml");
```

After retrieving the data from either an XML source or a text file, the next task is to traverse XML & modify it.

XLinq: Traversing XML & Modifying data

Continuing with the same example,

```
XElement recordsFromFile =
            XElement.Load(@"C:\Punit Ganshani\Employee.xml");

foreach (XNode c in recordsFromFile.Nodes()) {
        Console.WriteLine(c);
}
```

Output:

```
<Employee>
  <name>Punit</name>
  <phone type="mobile">9881070204</phone>
  <phone type="home">27471111</phone>
</Employee>
```

This will get us all the nodes in the XML file. To get inner/child nodes,

```
foreach (XNode c in recordsFromFile.Elements("Employee").Nodes())  {
        Console.WriteLine(c);
}
```

Output:

```
<name>John</name>
<phone type="mobile">9881070204</phone>
<phone type="home">27471111</phone>
```

Addition of data, deletion or updating data can be classified under 'modification of data.' Now, let us add another record into this XML file.

```
XElement newRecord = new XElement("Employee",
                        new XElement("name", "Scott"),
                        new XElement("phone", "123",
                            new XAttribute("type", "mobile")),
                        new XElement("phone", "12345",
                            new XAttribute("type", "home")));

recordsFromFile.Add(newRecord);
recordsFromFile.Save(@"C:\\Punit Ganshani\\EmployeeNew.xml");
```

Let us now replace content of XML node using XLinq. Say, we want to change the phone number to 454645667

```
records.Element("Employee").Element("phone").ReplaceNodes("454645667");
```

To remove a node/element/attribute from the XML file:

```
records.Element("Employee").Element("name").Remove();
records.Elements("phone").First().Attribute("type").Remove();
```

XLinq: Querying XML

In this section, we will consider the "select" and "from" clauses that will help us to retrieve only selected few records from the XML file. I've inserted some more records in `EmployeeXLinq.xml` to have more results from the queries in this section.

Let us first retrieve all the records from the XML file without the column City. So in the select clause, we will only mention Columns Name and Phone.

```
records = XElement.Load(@"C:\Punit Ganshani\EmployeeXLinq.xml");

XElement newRecords = new XElement("Records",
         from c in records.Elements("Employee")
         select new object[] {
             new XElement("Employee",
                   new XElement("name", (string)c.Element("name")),
                    c.Elements("phone")
                )
        }
);
```

Output:

```
<Records>
  <Employee>
    <name>John</name>
    <phone type="mobile">9881070204</phone>
    <phone type="home">27471111</phone>
  </Employee>
  <Employee>
    <name>Pete</name>
    <phone type="mobile">9960930661</phone>
    <phone type="home">12436156</phone>
  </Employee>
  <Employee>
    <name>Scott</name>
    <phone type="mobile">9981120232</phone>
    <phone type="home">99836156</phone>
  </Employee>
  <Employee>
    <name>Waker</name>
    <phone type="mobile">9964827451</phone>
    <phone type="home">34563211</phone>
  </Employee>
</Records>
```

There may be some instances, where employees may not have contact number. In other words, there would be a node Phone, but will not have any data (or will have null) in it. To handle such situations, the field/element gets appended by “Any() ? condition: else” just as in the next example.

```
c.Elements("phone").Any() ? ... : null
```

acts as a ternary operator in C#. If the condition `c.Elements("phone").Any()` is false, it assigns `null` to it, else it returns the phone number from the XML.

```
XElement newRecords = new XElement("Records",
         from c in records.Elements("Employee")
         select new object[] {
             new XElement("Employee",
                   new XElement("name", (string)c.Element("name")),
                   c.Elements("phone").Any() ?
                  new XElement("phoneNumbers", c.Elements("phone")) :
                  null
               )
        }
);
```

Now let us apply a filter over the records, or let us design a query on XML files.

```
XElement filterRecords = new XElement("Records",
           from c in records.Elements("Employee")
           where ((string)c.Element("name")).ToString().StartsWith("S")
           orderby (string) c.Element("name")
            select new object[] {
                new XElement("Employee",
                        new XElement("name", (string)c.Element("name")),
                        c.Elements("phone")
                 )
            }
);
```

```
Output:
```

```
<Records>
  <Employee>
    <name>Scott</name>
    <phone type="mobile">9981120232</phone>
    <phone type="home">99836156</phone>
  </Employee>
  <Employee>
    <name>Pete</name>
    <phone type="mobile">9960930661</phone>
    <phone type="home">12436156</phone>
  </Employee>
</Records>
```

Note that the other two records have not been displayed here. Let us understand the query:

```
from c in records.Elements("Employee")
```

Here `records.Elements("Employee")` is the source of data and 'c' is the temporary variable that will store the data.

```
where ((string)c.Element("name")).ToString().StartsWith("S")
```

Here `c.Element("name")` is the column on which the filter is applied. The data of the column is first converted to string (which is must) before the filter condition is coded

```
orderby (string) c.Element("name")
```

OrderBy clause defines the order of display. By default, it is ascending. To have descending order, `descending` needs to be appended as a suffix

Xlinq - DLinq: Collaborating data with SQL

To understand this section requires basic understanding of DLinq (which is assumed). We have created a DataContext (equivalent of database in DLinq) using the LINQ to SQL Wizard in Visual Studio 2008 IDE in our application: dbNorthWindDataContext.

```
dbNorthWindDataContext db = new dbNorthWindDataContext();
```

Now, let us retrieve data from SQL database and display it in form of XML.

```
XElement londonCustomers =
         new XElement("Customers",
                    from c in db.Customers
                    where c.City == "London"
                    select new XElement("Customer",
                        new XAttribute("CustomerID", c.CustomerID),
                        new XElement("Name", c.ContactName),
                        new XElement("Phone", c.Phone)
                    )
                    );
```

Here, the basic query remains the same; however, the datasource changes from a traditional XML file to a SQL database. Further going into the depth,

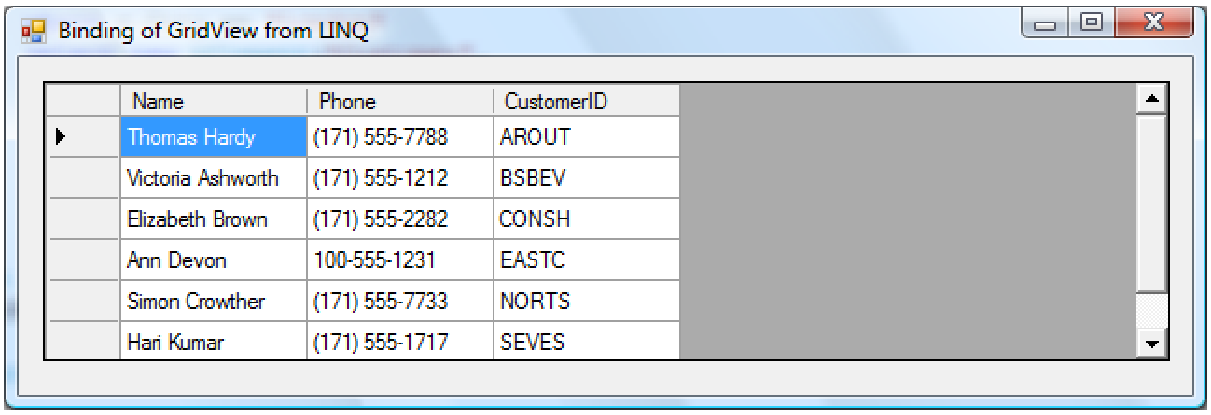
```
XElement londonCustomers =
         new XElement("Customers",
                  from c in db.Customers
                  where c.City == "London"
```

db.Customers relates to the Table Customers in the NorthWind Database. A copy/replica of the table is created in the temporary variable 'c'

```
  select new XElement("Customer",
                      new XAttribute("CustomerID", c.CustomerID),
                      new XElement("Name", c.ContactName),
                      new XElement("Phone", c.Phone)
                  )
```

The select clause creates a new copy of the XML file, which gets populated by the records that match the filter condition `c.City == "London"` The result of the query londonCustomers can be loaded into DataSet and can be displayed as shown in the Figure **2: Binding of GridView from LINQ.**

```
DataSet ds = new DataSet();
ds.ReadXml(londonCustomers.CreateReader());
```



	Name	Phone	CustomerID
▶	Thomas Hardy	(171) 555-7788	AROUT
	Victoria Ashworth	(171) 555-1212	BSBEV
	Elizabeth Brown	(171) 555-2282	CONSH
	Ann Devon	100-555-1231	EASTC
	Simon Crowther	(171) 555-7733	NORTS
	Hari Kumar	(171) 555-1717	SEVES

Figure 2: Binding of GridView from LINQ.

After understanding the development aspects of XLinq, let us see the benefits that XLinq offers us

Benefits of XLinq

1. Ease in the manipulation of existing XML documents.
2. Lesser time to code resulting in faster & improved productivity of developers.
3. Unified querying of Objects, Database and XML eases the interaction between XML and databases.

4. Ease of switching from Disconnected/Offline mode to Online mode in Smart Client applications.

   Most Smart Client applications use XML as datasource in Offline mode & use SQL database in the Online mode. Since only the datasource name changes and other code remains AS-IS, it becomes easier to handle data.

5. Since XElement is lighter than XmlDocument, XLinq proves to be faster than W3C DOM APIs.

Summary

XLinq is a powerful technology that can help us integrate XML with SQL, ease the use XML in Smart Client applications, and reduce the development efforts, yet provide high performance.