Chapter No.6
"LINQ to XSD"
In this package, you will find:
A Biography of the author of the book
A preview chapter from the book, Chapter NO. 6 "LINQ to XSD"
A synopsis of the book’s content
Information on where to buy this book

About the Author

N Satheesh Kumar has a Bachelor's Degree in Computer Science Engineering and has around eleven years of experience in software development lifecycle, project and program management. He started his career developing software applications using Borland software products in a company based in India, and then moved to the United Arab Emirates where he continued developing custom application software using Borland Delphi, and customizing Great Plain Dynamics (now known as Microsoft Dynamics) for an automobile company. He moved back to India and spent three years in design and developing application software using Microsoft products for a top multi national company, and then spent a couple of years in project management and program management activities. Now he works as a Technical Architect for a top retail company based in the United States. He works with the latest Microsoft technologies and has published many articles on LINQ and other features of .NET.

LINQ Quickly

Language Integrated Query (LINQ) is a new feature in Visual Studio 2008 that extends its query capabilities using C# and Visual Basic. Visual Studio 2008 comes with LINQ provider assemblies that enable the use of LINQ with data sources, such as in-memory collections, SQL relational databases, ADO.NET Datasets, XML documents, etc. In Visual Studio 2008, Visual C# and Visual Basic are the languages that implement the LINQ language extensions. LINQ language extensions use the new standard query operators, API, which is the query language for any collection that implements IEnumerable<T>.

This book introduces the reader to the basic concepts of LINQ, and takes them through using LINQ with an example-driven approach.
What This Book Covers

Chapter 1 looks at the overall features of LINQ, and gives an overview of different operators provided by LINQ to operate on objects, XML, relational databases, etc.

Chapter 2 examines LINQ to Objects, which means that you can use LINQ to query objects in a collection. Using this feature, you can access in-memory data structures using LINQ. You can directly query collections, and filter out required values without using powerful filtering, ordering, and grouping capabilities.

Chapter 3 looks at LINQ to XML. It is a new in-memory XML programming API to work against XML data. There are different ways of creating XML trees in .NET. LINQ to XML is the new method of creating and manipulating XML data through .NET. The properties and methods of LINQ help in navigating and manipulating XML elements and attributes.

Chapter 4, which covers LINQ to SQL, takes care of translating LINQ expressions into equivalent T-SQL, passing it onto the database for execution, and then returning the results back to the calling application. It reduces programming time and comes with two different design-time tools, which are used for converting relational database objects into object definitions. It also has the ability to create databases, and database objects.

Chapter 5 examines LINQ to DataSets. An ADO.NET DataSet provides a disconnected data source environment for applications. It can be used with multiple data sources. A DataSet has the flexibility of handling data locally in cache memory where the application resides. The application can continue working with a DataSet when it is disconnected from the source and is not dependent on the availability of the data source. The DataSet maintains information about the changes made to data so that updates can be tracked and sent back to the database as soon as the data source is available or reconnected.

Chapter 6 covers LINQ to XSD. It enhances XML programming by adding the feature of typed views on un-typed XML trees. LINQ to XSD gives a better programming environment by providing the object models generated from XML schemas. This is called typed XML programming.

Chapter 7 looks at standard query operators provided by LINQ, and how you can use some of them against different data sources.

Appendix A takes you through building a simple ASP.NET application using LINQ.

Appendix B tells you how to access Outlook objects using LINQ.
LINQ to XSD enhances XML programming by adding the feature of typed views on un-typed XML trees. A similar type of feature is available for DataSets in ADO.NET programming where we have typed DataSets. LINQ to XSD gives a better programming environment by providing the object models generated from XML schemas. This is called **typed XML programming**.

LINQ to XSD is an incubation project on typed XML programming. This product is not released yet. All examples and information in this chapter are based on this incubation project and are tested with Visual Studio 2008 Beta 1.

This LINQ to XSD project should reference System.Xml.XLinq and Microsoft.Xml.Schema.Linq libraries. Following is an example of accessing un-typed XML elements using LINQ query.

```csharp
from c in LoadIcecreams.Elements("Icecream")
select new XElement("Icecream",
    c.Element("Price"),
    c.Element("Name"));
```

An equivalent LINQ query for the above un-typed XML as a typed XML would be as follows:

```csharp
from Icecream in chapter6.Icecream
select new { Icecream.Price, Icecream.Name};
```

In this chapter we will see how to create typed XML, the features supported by typed XML, and how it helps in development.

The XML element that has been assigned a data type in an XML schema is called typed XML. This data type is used by the XML parser to validate the XML element value against the data type. The data type definition resides either in the same XML file, or in a separate schema file.
Let us consider the following XML in all our examples. It contains a namespace called http://www.Sample.com/Items. The XML has details of three different ice-creams. The root element of the XML is Chapter6. The first line in the XML shows details like version, and encoding for the XML.

```xml
<?xml version="1.0" encoding="utf-8"?>
<Chapter6 xmlns="http://www.Sample.com/Items">
  <Icecream>
    <!--Chocolate Fudge Icecream-->
    <Name>Chocolate Fudge Icecream</Name>
    <Type>Chocolate</Type>
    <Ingredients>
      cream, milk, sugar, corn syrup, cellulose gum...
    </Ingredients>
    <Cholestrol>50mg</Cholestrol>
    <TotalCarbohydrates>35g</TotalCarbohydrates>
    <Price>10.5</Price>
    <Protein>
      <VitaminA>3g</VitaminA>
      <Calcium>1g</Calcium>
      <Iron>1g</Iron>
    </Protein>
    <TotalFat>
      <SaturatedFat>9g</SaturatedFat>
      <TransFat>11g</TransFat>
    </TotalFat>
  </Icecream>
  <Icecream>
    <!--Cherry Vanilla Icecream-->
    <Name>Vanilla Icecream</Name>
    <Type>Vanilla</Type>
    <Ingredients>
      vanilla extract, guar gum, cream, nonfat milk, sugar, locust bean gum, carrageenan, annatto color...
    </Ingredients>
    <Cholestrol>65mg</Cholestrol>
    <TotalCarbohydrates>26g</TotalCarbohydrates>
    <Price>9.5</Price>
    <Protein>
      <VitaminA>1g</VitaminA>
      <Calcium>2g</Calcium>
      <Iron>1g</Iron>
    </Protein>
    <TotalFat>
      <SaturatedFat>7g</SaturatedFat>
      <TransFat>9g</TransFat>
    </TotalFat>
  </Icecream>
</Chapter6>
```
Chapter 6

</Icecream>
</Icecream>

<!-- Chocolate Icecream-->
>Name>Banana Split Chocolate Icecream</Name>
>Type>Chocolate</Type>
<Ingredients>Banana, guar gum, cream, nonfat milk, sugar, almonds, raisins, honey, locust bean gum, chocolate, annatto color...</Ingredients>

<Cholesterol>58mg</Cholesterol>
>TotalCarbohydrates>24g</TotalCarbohydrates>
PRICE>11</Price>
-Protein>
>VitaminA>2g</VitaminA>
>Calcium>1g</Calcium>
<Iron>1g</Iron>
</Protein>
>TotalFat>
>SaturatedFat>7g</SaturatedFat>
<TransFat>6g</TransFat>
</TotalFat>
</Icecream>
</Chapter6>

Un-typed XML

Following is a sample query for accessing data from an un-typed XML, shown pereviously:

XNamespace ns = "http://www.sample.com/Items";
return
(
from icecreams in Items.Elements(ns + "Icecreams")
from item in icecreams.Elements(ns + "Icecream")
select item.Element(ns + "Price"),
item.Element(ns + "Name")
);

The query uses a namespace, ns. This namespace is used to uniquely identify the XML elements. It is prefixed with all the elements used in the query. Each element of the XML is accessed using Element object. The select statement in the query above uses Element object to access the value of Price and Name of each Icecream in the XML document.
Typed XML

The following code is the XML Schema (XSD) contract for the sample XML that we have in the previous section. This schema defines a namespace for the XML, a type for the XML element and its maximum and minimum occurrences. It also describes the name and type for each element in the XML.

```xml
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema id="IcecreamsSchema"
    targetNamespace="http://www.Sample.com/Items"
    elementFormDefault="qualified"
    xmlns="http://www.Sample.com/Items"
    xmlns:mstns="http://www.Sample.com/Items"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Chapter6">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Icecream"
          minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="Icecream">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Name" type="xs:string"/>
        <xs:element name="Type" type="xs:string"/>
        <xs:element name="Ingredients" type="xs:string"/>
        <xs:element name="Cholestrol" type="xs:string"/>
        <xs:element name="TotalCarbohydrates" type="xs:string"/>
        <xs:element name="Price" type="xs:double"/>
        <xs:element ref="Protein"
          minOccurs="0" maxOccurs="1"/>
        <xs:element ref="TotalFat"
          minOccurs="0" maxOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="Protein">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="VitaminA" type="xs:string"/>
        <xs:element name="Calcium" type="xs:string"/>
        <xs:element name="Iron" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="TotalFat">
    <xs:complexType>
      <xs:sequence>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```
LINQ to XSD automatically creates classes from the XML schema used for the XML. These classes provide typed views of XML elements. Instances of LINQ to XSD types are referred to as XML Objects. Instances of LINQ to XSD classes are wrappers-around instances of the LINQ to XML class, XElement. All LINQ to XSD classes have a common base class, XTypedElement. This is contained in the Microsoft.Xml.Schema.Linq library.

```csharp
class XTypedElement
{
    private XElement xElement;
    /*
    Remaining class definition
    */
}
```

The element declaration is mapped to a subclass of XTypedElement, shown as follows:

```csharp
public class Icecream : XTypedElement
{
    //
}
```

These classes contain a default constructor, and properties for the elements and attributes. It also provides methods like Load, Save, Clone, etc. When XML is typed, the XML tree is readily loaded into the instance of the generated classes. Here, the type casting of the elements is not required for querying the XML elements.

**Creating Typed XML using Visual Studio**

Visual Studio gives IDE support for the LINQ to XSD feature. It automates the mapping of schemas to classes. The following example is based on LINQ to XSD, Preview Alpha 2.0 with Visual Studio Beta 1.
LINQ to XSD

Using the **New Project** option, create a LINQ to XSD Console Application.

![New Project dialog]

Using the **Add New Item** dialog, select the option to create an XML file under the project. If you have an XML file already, add it to the project using **Add Existing Item** option or copy-paste the contents of the available XML into the new XML file that is added to the project.

![Add New Item - Chapter 6 dialog]

Similarly add the XML schema file to the project. After adding the schema file, copy the schema content given in the typed XML section into the file.

Now we have an XML file and an XML schema file, but we need a tool to generate an object model for the schema. Open the Properties window of the XML schema file and select LinqToXsdSchema under Build Action. This is to instruct the tool to consider the schema file in its build process.
LINQ to XSD

The object model for the XML schema will get generated only after the build process. This build process also enables the Intellisense for the generated classes, and also displays the information in the **Object Browser** window. To view the object browser, select the menu option **View | Object Browser** from the Visual Studio IDE. This will display the hierarchy of all the objects present in the current project.

![Object Browser](image)

Now we can code with objects using Intellisense, as shown in the following screenshot:
We can also get a list of objects with the use of IntelliSense, as shown in the following screenshot:

```csharp
using System;
using System.Linq;
using System.Collections.Generic;
using System.Text;
using System.Xml;
using System.Xml.Linq;
using www.Sample.com;

namespace Chapter6Sample
{
    class Program
    {
        static void Main(string[] args)
        {
            var chapter6 = Chapter6.Load("Icecreams.xml");
            
            // Functional construction
            var icecreams = XElement.Parse("<Icecreams>
                <Icecream>
                    <Name>Vanilla</Name>
                    <Topping>Chocolate</Topping>
                </Icecream>
                <Icecream>
                    <Name>Strawberry</Name>
                    <Topping>Berry</Topping>
                </Icecream>
            </Icecreams>".ToString());
        }
    }
}
```

**Object Construction**

LINQ to XML provides a powerful feature called *functional construction*, which is the ability to create an XML tree in a single statement. All attributes and elements are listed as arguments of XElement constructor. XElement constructor takes various types of arguments for content. The argument can be an XElement, XAttribute, or an array of objects, so that we can pass any number of objects to the XElement. The functional construction feature is mainly used for navigating and modifying the elements and attributes of an XML tree. It actually transforms one form of data into another, instead of manipulating it.
LINQ to XSD

The following code shows how to build an un-typed XML tree using functional construction. Here, we use XElement to build a new Icecream element and add it to the existing Icecream element. For adding each element, we have to use XElement with the element name and value as parameters.

```csharp
Icecream.Add
{
    new XElement("Icecream",
        new XElement("Name", "Rum Raisin Ice Cream"),
        new XElement("Ingredients", "Rum, guar gum, nonfat milk, cream, almonds,
            sugar, raisins, honey, chocolate, annatto color...")
    )
};

Now we will see how to add a new element to the typed XML, without using XElement. We can directly use the objects to add the elements.

```csharp
var newObj = new Icecream
{
    Name = "Rum Raisin Ice Cream",
    Ingredients = "Rum, guar gum, nonfat milk, cream, almonds, sugar,
        raisins, honey, chocolate, annatto color...",
    Cholesterol = "49mg",
    TotalCarbohydrates = "28g",
    Protein = new Protein {VitaminA = "2g", Iron = "4g",
        Calcium = "1g"},
    Price = 10.25,
    TotalFat = new TotalFat {SaturatedFat = "5g", TransFat = "3g"}
};
chapter6.Icecream.Add(newObj);
```
Load Method
This is similar to the Load method that we saw for LINQ to XML, but the difference is that the Load method here is the typed version of the LINQ to XML’s Load method. Below is a sample which loads the xml file.

```csharp
var chapter6 = Chapter6.Load("Icecreams.xml");
```

Here chapters is a specific type which is already defined. The un-typed loading in LINQ to XML can be made typed by casting the un-typed loading with the type that is required.

```csharp
namespace Chapter6Sample
{
    class Program
    {
        static void Main(string[] args)
        {
            var chapter6 = Chapter6.Load("Icecreams.xml");
            var chapter6 = (Chapter6) XElement.Load("Icecreams.xml");
            var chapter6 = new Chapter6();
        }
    }
}
```

In this example, you can see the casting of Chapter6 done to the XElement, used for loading the XML document into chapterSix, which is a variable equivalent to the typed XML, chapters.

Parse Method
This method is the typed version of the Parse method used in LINQ to XML. Parsing is used to convert a string containing XML into XElement and cast that instance to a type required. Following is an example which parses a string containing XML, and types it to Chapter6.

```csharp
Sample.com/Items'>
<Icecream>
  <!--Chocolate Fudge Icecream-->
  <Name>Chocolate Fudge Icecream</Name>
  <Type>Chocolate</Type>
  <Ingredients>cream, milk, sugar, corn syrup, cellulose gum...</Ingredients>
  <Cholesterol>50mg</Cholesterol>
  <TotalCarbohydrates>35g</TotalCarbohydrates>
  <Price>10.5</Price>
  <Protein>
    <VitaminA>3g</VitaminA>
</Icecream>
</Chapter6>");
```

LINQ to XSD

"   <Calcium>1g</Calcium> " +
"   <Iron>1g</Iron> " +
" </Protein> " +
" <TotalFat> " +
"   <SaturatedFat>9g</SaturatedFat> " +
"   <TransFat>11g</TransFat> " +
" </TotalFat> " +
" </Icecream></Chapter6> "
};

Save Method

This method is the typed version of the Save method used by LINQ to XML. This method outputs the XML tree as a file, a TextWriter, or an XmlWriter.

// Save as xml file
chapter6.Save(@"c:\LINQtoXSDSave.xml");
// or output as TextWriter
chapter6.Save(TextWriter testTextWriter);
// or output as XmlWriter
chapter6.Save(XmlWriter testXmlWriter);

The above code saves the XML tree in chapter6 object to a file named LINQtoXSDSave.xml.

Clone Method

The XTypedElement base class used for all the generated classes defines a Clone method. The result of the clone operation is weakly typed, as the clone is applied to the underlying un-typed XML tree. So, to get a specific type, a casting must be used while cloning.

// Load xml
var chapter6 = Chapter6.Load("Icecreams.xml");
// Create a Clone of chapter6 xml
var chapter6Clone = (Chapter6)chapter6.Clone();
var Qry1 = from Icecream in chapter6Clone.Icecream
select new { Icecream.Price, Icecream.Name };
Console.WriteLine(" ");
Console.WriteLine("Clone Sample ");
foreach (var itm in Qry1)
    Console.WriteLine("Price of {0} is : {1}", itm.Name , itm.Price);
In the above example, we are loading the XML into `chapter6` variable, and then creating a clone for it. We are type casting the new clone of the type `Chapter6`, and then assigning the resultant clone to `chapter6Clone`. Even though we have not assigned any XML to `chapter6Clone`, the query produces the same result as that of the same query applied on `chapter6` XML. Internally, the XML is the same for both objects, as `chapter6Clone` is just a clone of `chapter6`.

**Default Values**

Default is the value returned for the elements in the XML, in case the value of the XML element is empty in the XML tree. The same applies to the attributes also, but in case of attributes, they may not be present in the XML tree. The default value is specified in the XML fragment.

```xml
<xs:element name="Protein">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="VitaminA" type="xs:string" default="0g"/>
      <xs:element name="Calcium" type="xs:string" default="0g"/>
      <xs:element name="Iron" type="xs:string" default="0g"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

In the above example, the elements VitaminA, Calcium, and Iron are the three elements that have a default value of 0g. So if the XML tree does not have any value specified for these elements, the resulting value for these elements would be 0g.

**Customization of XML Objects**

The various types of customizations used in LINQ are explained in the following subsections.

**Mapping Time Customization**

There is a configuration file that controls the LINQ to XSD mapping details. XML namespaces can be mapped to CLR namespaces. For example, the default mapping for `http://www.Sample.com/Items` would be `www.Sample.com.Items`. The following example maps `http://www.Sample.com/Items` to `LinqToXsdExample.Schema.Items`:

```xml
<Configuration xmlns="http://www.microsoft.com/xml/schema/linq">
  <Namespaces>
    <Namespace>
      Schema="http://www.Sample.com/Items"
    </Namespace>
  </Namespaces>
</Configuration>
```
LINQ to XSD

```xml
<Configuration

<Namespaces

  Clr="LinqToXsdExample.Schema.Items"/>
</Namespaces>
</Configuration>
```

This is also used for systematic conversion of nested, anonymous complex types into complex type definitions.

A configuration file is:

- An XML file with a designated namespace.
- Used by the command line processor of LINQ to XSD.
- Used in Visual Studio for LINQ to XSD project. The build action can be specified as LinqToXsdConfiguration.

We can map the Schema without a target namespace to a CLR namespace.

**Compile Time Customization**

LINQ to XSD generates classes, and provides the object model which can be customized using .NET. It is not so easy to customize the generated code as it requires a lot of understanding. Even if we customize the generated code, the customization will be lost if the code gets regenerated. The best option is to use sub-classing or extension of partial classes by which we can add methods to the generated class by LINQ to XSD.

Following is the object model for our chapter6 XML where `chapter6`, `Icecream`, `Protein`, and `TotalFat` are all generated as classes.
Now we can create a partial class for the corresponding classes, and add methods to override the base class functionality.

The LINQ to XSD Visual Studio projects use the obj\Debug\LinqToXsdSource.cs file to hold the generated classes.

**Post Compile Customization**

For customizing the classes at compile time, we can use partial classes. If the object models are available in compiled format, and we do not have the source for the generated classes, we can use the extension methods to add the behaviour to the compiled objects. The LINQ to XML annotation facility can be used for this.

**Using LINQ to XSD at Command Line**

There is a command line tool called `LinqToXsd.exe` which is an XSD to class mapper. This tool provides two options to convert XSD:

- Generating a `.dll` from XSD.
- Generating a `.cs` file from XSD, which is a default.

For example, following is the command to generate a DLL from an XSD from the location where the LINQ to XSD is installed:

```
LinqToXsd.exe Items.xsd /lib: Items.dll
```

**Summary**

In this chapter, we have seen the different features that are going to come up with LINQ to XSD. We have also seen examples for some of the features supported by LINQ to XSD. This makes the programmer's job easier by turning the un-typed XML to typed XML. LINQ to XSD generates the classes for XML elements, and also provides an object model, which the programmer can directly access, just as he or she would do with .NET objects.
Where to buy this book
You can buy LINQ Quickly from the Packt Publishing website:
Free shipping to the US, UK, Europe, Australia, New Zealand and India.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.