|  |
| --- |
|  |
| LINQ to XML |
| <Linq love=”true” /> |
|  |
|  |
|  |

|  |
| --- |
|  |

# 

LINQ to XML

# Objectives

After completing this lab you should understand how to do the following:

* Create an XML file using the System.Xml.Linq API.
* Add content to an XElement using functional construction.
* Query an XML file using LINQ to XML

# Overview

In this lab, we will be working with the file system and ASP.NET. We’ll build an XML file with the contents of a directory and its subdirectories, and bind the XML to a TreeView control. We’ll also query the XML file and bind our query results to a GridView control.

# Part I – Creating XML

1. In Visual Studio, use the File -> Open Web Site command and open the LINQ\_XML\before directory. The before directory is the directory we’ll be working inside, the after directory contains a completed version of the lab.
2. Press Ctrl + F5 to run the web site and ensure there are no errors. You should see the page render a TreeView control with some data. There is also a GridView control on this page, but it does not have any information to display (yet).
3. Close the application and open Default.aspx in design view.

Notice that in addition to the TreeView and GridView controls, we have XmlDataSource and ObjectDataSource controls. These controls will feed their respective UI controls with data. Our first job will be to refresh the file consumed by the XmlDataSource control.

1. Right-click the XmlDataSource and select the Properties command. Notice the DataFile property is set to ~/App\_Data/FileData.xml.
2. Open FileData.xml from the App\_Data directory. We are going to recreate this file during the Load event of the page using your current file system.
3. From the App\_Code directory, open the FileReportDataSource.cs file.
4. Add a public method to the class with the name *RefreshFileData*.
5. Add a local string variable to the method by the name of *path*. Pick a directory on your local hard drive and assign its full path name to the *path* variable.

Since we will be creating an XML file by recursively traversing the filesystem from this point, try to pick a directory that will not strain your system (“C:\” would be a bad choice). You might start by using the directory where this lab resides. The code should look similar to the following.

public class FileReportDataSource

{

...

public void RefreshFileData()

{

string path =

@"C:\labs\LINQ\_Xml";

}

...

}

1. Inside the RefreshFileData method, construct a new XElement object with an XName of Files (remember this API will happily convert the string literal “Files” to an XName for you). Name the object variable itself as *root*.

XElement root = new XElement("Files");

1. Invoke the Add method of the root object, and inside the method call construct a new XElement with an XName of “Directory”.
2. The new “Directory” XElement must include a new XAttribute with an XName of “Name”, and the value of this attribute should equal the value of the *path* variable.

Be careful with your commas and parenthesis, you must pass the XAttribute as a content parameter to the Directory element, **not** the Add method.

root.Add(new XElement("Directory",

new XAttribute("Name", path)));

1. Finally, invoke the Save method of the *root* object, passing *\_fileName* as the file name to use.

The \_fileName field was already declared inside the class and initialized inside the constructor for you. Your code so far should look like the following.

public void RefreshFileData()

{

string path = @"… your choice …";

XElement root = new XElement("Files");

root.Add(new XElement("Directory",

new XAttribute("Name", path)));

root.Save(\_fileName);

}

1. Go to the Page\_Load method of Default.aspx.cs and add an if statement to check for *!IsPostBack*. We only want to build this file on the initial page request. Inside the if statement, create a new instance of the FileReportDataSource class and invoke the RefreshFileData method.

protected void Page\_Load(object sender, EventArgs e)

{

if (!IsPostBack)

{

var dataSource = new FileReportDataSource();

dataSource.RefreshFileData();

}

}

1. Press F5 to run the project. The page should display without an error, but display only the single directory you’ve selected.

If you have any data binding errors at this point, you’ve probably constructed the XML incorrectly. The XML should look like the following.

<?xml version="1.0" encoding="utf-8"?>

<Files>

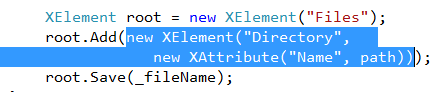
<Directory Name="…your path…" />

</Files>

1. Close the application and return to the RefreshFileData method.

We need to add some logic to recurse through directories and add each directory’s content to the XML. We’ll need a recursive method, and some additional logic. Visual Studio refactoring support will help us create a new method.

1. Highlight all of the code *inside* the Add method, as shown below.



1. With the above selection still highlighted, go to the Refactor menu in Visual Studio and click the Extract Method command.
2. In the Extract Method dialog, enter a method name of *CreateXmlForDirectory*. Press OK.
3. Verify the code inside RefreshFileData should now look like the following:

XElement root = new XElement("Files");

root.Add(CreateXmlForDirectory(path));

root.Save(\_fileName);

1. Go to the CreateXmlForDirectory method. Since we need additional logic, remove the return keyword and assign the newly constructed Directory element to a local variable named *root*. Add a new return statement at the end of the method to return the *root* reference.

private static XElement CreateXmlForDirectory(string path)

{

**XElement root =** new XElement("Directory",

new XAttribute("Name",

**directoryInfo.Name**));

**return root;**

}

We are passing an absolute path to CreateXmlForDirectory. You might have noticed that when you ran the page earlier that the TreeView displayed this absolute path. What we actually want is for the Name attribute of each Directory element to hold just the directory’s name, not it’s absolute path. We will address this feature in the following steps.

1. Construct a new DirectoryInfo object at the top of the CreateXmlForDirectory method using the incoming *path* parameter. Assign the DirectoryInfo object to a local variable named *directoryInfo*.
2. Use the directoryInfo’s Name property when creating the XAttribute for the XML. The updated code should look like the following.

private static XElement CreateXmlForDirectory(string path)

{

**var directoryInfo = new DirectoryInfo(path);**

XElement root **=** new XElement("Directory",

new XAttribute("Name",

**directoryInfo.Name**));

return root;

}

1. After constructing the root element, save the result of the directoryInfo’s GetDirectories method to a local variable named *directories*.
2. Next, save the result of the directoryInfo’s GetFiles method to a local variable named *files*.

The last two steps are highlighted in the code below.

var directoryInfo = new DirectoryInfo(path);

XElement root = new XElement("Directory",

new XAttribute("Name",

directoryInfo.Name));

**var directories = directoryInfo.GetDirectories();**

**var files = directoryInfo.GetFiles();**

return root;

1. Declare a new local variable with the var keyword named *directoryXml*.
2. Assign the variable a query that will construct an XElement for each entry in the *directories* variable.

You can do this using recursion by passing the FullName property of each DirectoryInfo object to CreateXmlForDirectory. .

…

var directories = directoryInfo.GetDirectories();

var files = directoryInfo.GetFiles();

**var directoryXml =**

**from d in directories**

**select CreateXmlForDirectory(d.FullName);**

…

1. Next, create a new local variable with the var keyword named *fileXml*. Assign the variable a query that will construct a new XElement for each entry in the *files* sequence. The element should have the XName of File, and include attributes named Name, Size, and LastAccess. You can find values for each attribute in the FileInfo objects of the files sequence.

…

var directoryXml =

from d in directories

select CreateXmlForDirectory(d.FullName);

**var fileXml =**

**from f in files**

**select new**

**XElement("File",**

**new XAttribute("Name", f.Name),**

**new XAttribute("Size", f.Length),**

**new XAttribute("LastAccess", f.LastAccessTime));**

…

1. Using the *root* object’s Add method, insert the directoryXml and fileXml as content into *root*. The completed method looks like the following.

private static XElement CreateXmlForDirectory(string path)

{

var directoryInfo = new DirectoryInfo(path);

XElement root = new XElement("Directory",

new XAttribute("Name",

directoryInfo.Name));

var directories = directoryInfo.GetDirectories();

var files = directoryInfo.GetFiles();

var directoryXml =

from d in directories

select CreateXmlForDirectory(d.FullName);

var fileXml =

from f in files

select new

XElement("File",

new XAttribute("Name", f.Name),

new XAttribute("Size", f.Length),

new XAttribute("LastAccess",

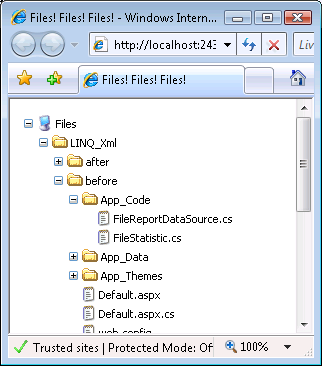
f.LastAccessTime));

**root.Add(directoryXml, fileXml);**

return root;

}

1. Press F5 to run your project with the debugger. You should see the TreeView now with directories, subdirectories, and files.
2. Make sure you also inspect the FileData.xml file in App\_Data to see the raw XML you’ve created.



# Part II – Querying XML

In this section of the lab, we want to query FileData.xml to find the 10 largest files from any directory.

Open FileStatistic.cs from the App\_Code directory.

This is the class we will instantiate during our queries. We will bind collections of FileStatisitc objects to the GridView control.

1. Open the FileReportDataSource.cs file in App\_Code.

This is the class the ObjectDataSource control will use to fetch data for the

GridView. .

1. Go to the GetLargestFiles method, and delete the existing return statement.
2. We’ll first need to load FileData.xml into memory. Use the static Load method on the XDocument class to bring the file into memory (use \_*fileName* for the file name parameter). Save the return value in a local variable named *doc*.

public IEnumerable<FileStatistic> GetLargestFiles()

{

XDocument doc = XDocument.Load(\_fileName);

}

1. Use the var keyword to declare a local variable named *query*, and start a LINQ query with the keyword *from*.

In deciding how we want to get to the <File> elements in our document, we have a couple options. We could use the Elements method of the document to retrieve <File> elements, but the Elements method only looks at the immediate children of a given node. If we use the Descendants method, however, we can pull all the <File> elements out of the document regardless how deep they are nested.

1. Add the following cod to the method:

var query =

from e in doc.Descendants("File")

1. We want the largest files to appear first in the grid, so we need an orderby clause that looks at the Size attribute of each file element.

We need to be careful not to perform a string comparison, so we’ll parse the Size attribute’s Value property into a long for ordering.

var query =

from e in doc.Descendants("File")

**orderby long.Parse(e.Attribute("Size").Value) descending**

1. Project a new FileStatistic for each element. For the Directory property, we’ll need to fetch the Name attribute of the File element’s **Parent** property.

var query =

from e in doc.Descendants("File")

orderby long.Parse(e.Attribute("Size").Value) descending

**select new FileStatistic**

**{**

**Name = (string)e.Attribute("Name"),**

**Size = long.Parse(e.Attribute("Size").Value),**

**LastAccess = DateTime.Parse(**

**(string)e.Attribute("LastAccess")),**

**Directory = e.Parent.Attribute("Name").Value**

**};**

Notice there are a couple options for extracting a string value from an Attribute. We can use the Value property, or we can use an explicit cast to string. We could have also used casting to retrieve the Size attribute by casting to double (or double? if there could be a null value).

1. Return the top 10 files using the Take operator on the query and a return statement.

**return** query.Take(10);

1. Press F5 to run the project and verify your results.

Notice in the output that the Directory column show a directory name, not a full path. This could be confusing if we had directories in different folders with the same name. Let’s try to extract more path information using data from our XML file.

1. Create a new method in the FileReportDataSource class named *CombineParentDirectoryNames*. The method should return a string, and take a parameter named *fileElement* of type XElement.

string CombineParentDirectoryNames(XElement fileElement)

{

}

1. Now we’ll write a query that will traverse up the XML tree to find all the directory names above the current *fileElement*. This can be done using the Ancestors method of an XElement.

string CombineParentDirectoryNames(XElement fileElement)

{

var names =

from e in fileElement.Ancestors("Directory")

select e.Attribute("Name").Value;

}

One catch to this approach is that the ancestor elements are returned in their order going up the tree of XML. We want to formulate a path starting with the top-most directory name.

1. Use the Reverse operator to reverse the sequence of names. After we’ve reversed the sequence we can combine the sequence of strings into a more complete file path. See if you can do this with the LINQ Aggregate operator.

Hint: you might try a StringBuilder from the System.Text namespace.

string CombineParentDirectoryNames(XElement fileElement)

{

var names =

from e in fileElement.Ancestors("Directory")

select e.Attribute("Name").Value;

**return**

**names.Reverse()**

**.Aggregate(**

**new StringBuilder(),**

**(builder, name) =>**

**builder.Append(**

**String.Format("{0}{1}",**

**Path.DirectorySeparatorChar,**

**name)),**

**builder => builder.ToString());**

}

1. Back in the GetLargestFilesMethod, assign the return value of CombineParentDirectoryNames to the Directory property during projection.

var query =

from e in doc.Descendants("File")

orderby long.Parse(

e.Attribute("Size").Value) descending

select new FileStatistic

{

Name = (string)e.Attribute("Name"),

Size = long.Parse(e.Attribute("Size").Value),

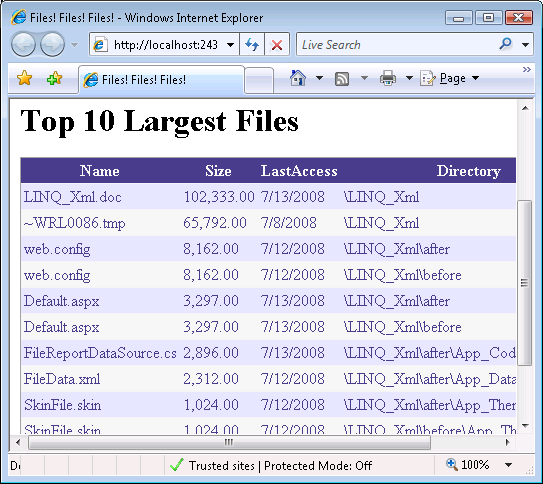
LastAccess = DateTime.Parse(

(string)e.Attribute("LastAccess")),

**Directory = CombineParentDirectoryNames(e)**

};

1. Press F5 to run your project and verify the results. We should now have more complete path information in the table showing the largest files.



# Conclusion

Congratulations! You’ve created and queried XML using LINQ to XML and the System.Xml.Linq API.