EnerNex

LINQ to Green Button

Authored by

Robert Henry, Software Engineer

January 23, 2014

EnerNex LLC
620 Mabry Hood Road, Suite 300
Knoxville, TN 37932
Telephone: 865-218-4600 || Facsimile: 865-218-8999
Website: www.enernex.com

This technical article presents a way to use software to help reduce energy costs, improve electric power system reliability, and reduce CO_2 emissions to the environment. Many electric power consumers are motivated to do their part to reduce energy use. If consumers could know their electric power consumption in real-time, it would help them identify unnecessary consumption, reduce peak power consumption to help the utility handle a power crisis, help to reduce their CO_2 footprint, and of course, save money. An increasing number of utilities are installing Smart Meters, which automatically collect data at frequent intervals and forward the data to the utility's business office in order to calculate the user's bill.

This technology provides several advantages for the utility, but it also offers direct advantages to the consumer if the utility can provide the data to the consumer in a form that they can use to understand their usage patterns. The objective of the Green Button Initiative is to provide a universal data format for developers to build useful applications for monitoring energy consumption. The advantages for application developers of a common data format should be obvious; a single application can be re-used by customers of many different utilities.

This project demonstrates how to read the data in a Green Button file using LINQ. It is not a complete application. Development of practical, useful Green Button applications is left to the reader.

1.1 Green Button

While smart meters help the utility reduce costs and improve reliability, there's also a payoff for the consumer by providing more feedback about one's energy use. Instead of receiving an electric bill at the end of the month containing a single meter reading, a smart meter may provide readings every 15 minutes. With that much data, consumers can monitor their energy use and quickly evaluate changes in their behavior, if they can get their hands on that data in an understandable form.

Green Button Initiative (http://www.greenbuttondata.org/) is an industry effort in collaboration with Commerce Department's National Institute of Standards and Technology, to develop a standard data format that utilities can use to provide data to consumers dubbed "Green Button." The availability of a common data format simplifies the developer's job of developing energy applications for the consumer.

An increasing number of utilities are providing their customers their energy data using the Green Button format. As the Green Button format becomes more widely deployed, developers can create applications to display and analyze energy data that will work with a wide audience of users.

EnerNex LLC
620 Mabry Hood Road, Suite 300 || Knoxville, TN 37932
Telephone: 865-218-4600 || Facsimile: 865-218-8999 || Website: www.enernex.com

1.1.1 Data structure of Green Button

Green Button is implemented by an XML document containing energy consumption data. Customers can download their usage data from their participating utility's web site and study the data using a suitable display application. The file uses the Atom Syndication Format (http://tools.ietf.org/html/rfc4287) to package the data. The Green Button data, or "Energy Standard Provider Interface" (ESPI), constitutes the content of the Atom feed. Some utilities may compress the XML document to reduce transmission costs. We'll briefly explore dealing with compressed Green Button packages as well.

The Green Button XML document can be accessed using many XML processing techniques, of which the LINQ technique described here is one example.

1.1.2 Sample Data

Sample data sets can be downloaded for testing from the Green Button Data site: http://www.greenbuttondata.org/greendevelop.aspx. This site has other information about Green Button. A collection of sample files is also included in the download package for this article.

1.2 LINQ

LINQ (Language Integrated Query) is a .NET framework component that provides in-language querying for .NET languages. In addition to querying SQL servers, LINQ also works with data structures such as arrays, collections, and XML documents. Among its advantages, LINQ uses similar query syntax for each of the different types of data sources and allows most errors to be checked at compile time rather than runtime. If you're using an IDE, it can display errors as you type.

This project uses an XML document so we'll be using LINQ to XML for this project. While there are other ways of accessing data in XML files, LINQ offers an alternative way of selecting and aggregating data that can simplify developing an energy application.

1.2.1 LINQPad

While LINQ applications can be written using Microsoft Visual Studio (http://www.visualstudio.com) or other .NET / ASP.NET development tools, LINQPad provides a lightweight tool to develop and test LINQ queries. LINQPad is available as a free download from www.linqpad.net. Advanced features, such as autocomplete, are available in LINQPad with paid upgrades. This project will be demonstrated using LINQPad. Once a query works in LINQPad, incorporating the query code into a Visual Studio project is usually a fairly simple operation.

EnerNex LLC
620 Mabry Hood Road, Suite 300 || Knoxville, TN 37932
Telephone: 865-218-4600 || Facsimile: 865-218-8999 || Website: www.enernex.com

1.3 Extracting data using LINQPad

To get started, open LINQPad. LINQPad opens with a query already set up so that all you have to do is type a query. By default, LINQPad creates new queries using the "Expression" setting, allowing developers to evaluate LINQ queries directly as a simple expression. However, we'll be writing a program with multiple statements, so in the "language" dropdown, select "C# Statement(s)." This example code is written in C#. LINQ will work with VB, and the queries are substantially the same, but a translation of the code from C# to VB is beyond the scope of this article.

A LINQPad file of this project is included in the download package. The LINQPad file, which has a file extension of .linq, can be opened and executed using LINQPad. However, if you'd prefer to build the code piece by piece, the following explains the code.

1.3.1 The Namespaces

Green Button uses two namespaces in the XML document, the namespace for *Atom* and the namespace for *Energy Services Provider Interface* (ESPI). First we'll add these namespaces to the LINQ application using the XNamespace class. They'll be needed to navigate the Green Button XML.

```
XNamespace a = @"http://www.w3.org/2005/Atom", e = @"http://naesb.org/espi";
```

1.3.2 XDocument Class

Next, we'll load the Green Button XML document using the LINQ XDocument class. This can be done simply with XDocument's Load() method. In this example we'll be loading the document from a file, but the Load() method has several overloads that might be appropriate for other projects.

Document.Load() requires a full path to the input file and that path might change, depending on where the sample files were downloaded and opened. To avoid hard-coding the path to the sample folder, which might break if the project folder were moved, we'll obtain the path to the sample file in the sample folder with the expression, "Path.GetDirectoryName (Util.CurrentQueryPath)." With the path to the project folder, just prepend the path to the sample file. This way, LINQPad will be able to find the sample file no matter where you extract the download package. Util.CurrentQueryPath will only work in LINQPad. In a deployed .NET application, you'd need to use different techniques to load an XML document.

```
var xml = XDocument.Load(Path.GetDirectoryName (Util.CurrentQueryPath)
      + @"\Green Button Samples\A1 Bakersfield pge_electric_interval_data_2011-04-04_to_2012-05-04 zipcode cost.xml");
```

1.3.3 Time

Each interval record in the file is labeled with a timestamp using UNIX time, which began on January 1, 1970, represented by the number of seconds after that date. To convert UNIX time to a .NET DateTime, we'll set up a DateTime object to help with the conversion.

EnerNex LLC
620 Mabry Hood Road, Suite 300 || Knoxville, TN 37932
Telephone: 865-218-4600 || Facsimile: 865-218-8999 || Website: www.enernex.com

```
DateTime uRef = new DateTime(1970,1,1,0,0,0);
```

To convert, we add the number of seconds from the XML file to this reference using the DateTime class AddSeconds method, which returns a .NET DateTime object.

The sample file used here covers over a year of data, but for this demonstration I want to extract a small interval to demonstrate LINQ's ability to manipulate the data. We'll add variables bracketing the desired dates. In an actual application, we'd enter these variables as parameters containing user-supplied values, but for this demonstration they're hard-coded.

```
DateTime startTime = new DateTime(2011,6,1,1,0,0);
DateTime endTime = new DateTime(2011,6,2,0,0,0);
```

1.3.4 XPath

Since this is XML, we still need to use XPath to traverse the document structure. For this demonstration, I'm simply going to extract the power consumption data. However, there are other important data items in the document that would also be needed for a complete application. They can be retrieved in much the same way with additional LINQ queries.

The root element of the Atom Syndication Format document is named "feed" and can be retrieved using the XDocument Descendants() method. Since there are namespaces, we'll have to include the Atom namespace in our call to Descendants. The namespace is prepended using the "+" operator, but it's important to note that this is not a simple C# string concatenation and there's no need to add any additional separator characters. The "+" operator has been overridden in the XDocument class to properly combine the namespace and the element name.

Atom can be traversed as far as the "content" element using the Elements() method. At the content element, we continue traversing into the ESPI part of the document through "IntervalBlock" and "IntervalReading" where the desired data is found. Note that here the ESPI namespace "e" defined previously is being used for the "IntervalBlock" and "IntervalReading" nodes. The completed navigation path to the desired data is:

```
…
xml.Descendants(a + "feed").Elements(a + "entry").Elements(a + "content")
.Elements(e + "IntervalBlock").Elements(e + "IntervalReading")
…
```

Next…

1.3.5 Filtering our sample

If I were to read all the data points reported in this sample file, there would be 37916 samples. As mentioned earlier, I want a small sample from this document. This is easily accomplished using the LINQ "where" clause, demonstrating the power of LINQ to XML. The data I'm going to arbitrarily select will cover 6/1/2011 at 1:00 AM to 6/2/2011 at midnight, about a day. The timestamp data is contained in a child of "IntervalReading," named "timePeriod." This element contains two child elements: "start" and "duration." For this demonstration I'm going to ignore

EnerNex LLC
620 Mabry Hood Road, Suite 300 || Knoxville, TN 37932
Telephone: 865-218-4600 || Facsimile: 865-218-8999 || Website: www.enernex.com

duration, assuming that the intervals are all the same, and use only the Start element. Start is a timestamp containing the number of seconds since 1970, so to convert into a .NET DateTime, we just need to add those seconds to the UNIX reference we set up earlier. The value will have to be cast to a long for the AddSeconds method. Notice here that LINQ handles parsing the string values in the XML document into numbers for us, so we don't have to worry about that. For this query, we're looking for records between the "startTime" and "endTime" DateTime objects we added previously, so the where clause is:

```
…
where uRef.AddSeconds((long)x.Element(e + "timePeriod").Element(e + "start")) >= startTime
  &&
uRef.AddSeconds((long)x.Element(e + "timePeriod").Element(e + "start")) < endTime
…
```

1.3.6 Select

The next part of the LINQ query is the select clause. We want to use the timestamp, the value, and the cost. The value will be in energy units, Watt Hours in this sample file. The units and multiplier being used in the file can be determined from elements elsewhere in the document. The cost is in currency units. I'll cover those later.

To give the elements a name we can use later, we'll give the date the name "eventDate," the value we'll name "dataValue," and the cost we'll name "dataCost." There's a possibility that the value or the cost might not be present in the document, so we'll have to check for null values. If they're null we'll just set them to zero. There are other ways of handling nulls, but this will suffice for this demonstration. You could also choose to create a custom class to contain these selected values, but for this demo we'll just use an anonymous class. Note that here, unlike in the where clause, we *do* have to parse the XML string data into numbers.

```
…
select new {eventDate = uRef.AddSeconds(long.Parse(x.Element(e + "timePeriod").Element(e + "start").Value)),
 dataValue = (x.Element(e + "value") == null ? 0 : int.Parse(x.Element(e + "value").Value)),
 dataCost = (x.Element(e + "cost") == null ? 0 : int.Parse(x.Element(e + "cost").Value))};
```

So combining all these components whole LINQ query is simply:

```
var query =       from x in xml.Descendants(a + "feed").Elements(a + "entry")
                .Elements(a + "content").Elements(e + "IntervalBlock").Elements(e + "IntervalReading")
                  where uRef.AddSeconds((long)x.Element(e + "timePeriod").Element(e + "start")) >= startTime
                        &&
                  uRef.AddSeconds((long)x.Element(e + "timePeriod").Element(e + "start")) < endTime
        select new
                  {
                        eventDate = uRef.AddSeconds(
                              long.Parse(x.Element(e + "timePeriod").Element(e + "start").Value)),
                  dataValue =
                        (x.Element(e + "value") == null ? 0 : int.Parse(x.Element(e + "value").Value)),
                  dataCost =
                     (x.Element(e + "cost") == null ? 0 : int.Parse(x.Element(e + "cost").Value))
                  };
```


EnerNex LLC
620 Mabry Hood Road, Suite 300 || Knoxville, TN 37932
Telephone: 865-218-4600 || Facsimile: 865-218-8999 || Website: www.enernex.com

1.4 Take a look at the result

If you're working with LINQPad, at this point you can see what you've retrieved from the document with the Dump() method.

```
query.Dump();
```

Dump() is a LINQPad extension and won't work if you're using Visual Studio, where you'll have to use other methods to view the data. Of course for a deployed application you'd probably use other techniques to display the data for the user such as a DataGrid or a graph.

1.5 Grouping by hours

Ok, that's a start. However, I want to group these records to show hourly totals. LINQ makes it simple; I'll just group them in using new query. I'm also making another adjustment here that I probably should have made in the first query: The cost values are in hundred-thousandths, [ESPI schema documentation] so to convert them to dollars, I have to multiply by $1X10^{-5}$ or 1e-5.

```
var grp = from q in query
          group q by new {q.eventDate.Date, q.eventDate.Hour} into g
          select new {eventDate = g.Key.Date.AddHours((double)g.Key.Hour), dataValue = g.Sum(y => y.dataValue), dataCost =
g.Sum(y=> y.dataCost)* 1e-5};
```

The cost values are in U.S. Dollars in this sample file, but the Green Button standard supports other currencies. Currencies are specified in the "currency" element using ISO 4217 codes (840 in this example).

1.6 Totals

Now what if we want to see how much total energy was used during the period and what it cost. Use the LINQ Sum() method. Sum takes a Lambda expression as an argument, which in this case simply allows us to choose which value in the just-completed query to sum. We'll sum the cost and the value. Since we've established that the energy value is in Watt Hours, we'll divide by 1000 to convert to Kilowatt Hours, which is what most people are more familiar with. Instead of using the LINQ method Dump() to display, we'll use the C# WriteLine() method, with a format to label the results. The second query, grp, must be used to calculate cost rather than the previous query, since cost was adjusted to convert to Dollars. That conversion causes dataCost to be cast as a double, so the sum also needs to be a double.

```
double cost = grp.Sum(n=>n.dataCost);
int value = grp.Sum(n=>n.dataValue)/1000;
Console.WriteLine("Energy: {0} kwh, Cost {1:C2}",value,cost);
```


EnerNex LLC
620 Mabry Hood Road, Suite 300 || Knoxville, TN 37932
Telephone: 865-218-4600 || Facsimile: 865-218-8999 || Website: www.enernex.com

1.7 The Whole Demo

The whole demo is here. You can copy/paste this into LINQPad if you don't want to download the package.

```
DateTime uRef = new DateTime(1970,1,1,0,0,0); // UNIX Time Reference
// Namespaces used in the Green Button XML document
XNamespace a = @"http://www.w3.org/2005/Atom", e = @"http://naesb.org/espi";

// A block of dates to search for in the file.
// Normally these would be inserted as a user-provided parameter,
//      but are hard-coded here for demonstration purposes.
DateTime startTime = new DateTime(2011,6,1,1,0,0);
DateTime endTime = new DateTime(2011,6,2,0,0,0);

// Prepend the sample folder path to sample file using Util.CurrentQueryPath
var xml = XDocument.Load(Path.GetDirectoryName (Util.CurrentQueryPath)
      + @"\Green Button Samples\A1 Bakersfield pge_electric_interval_data_2011-04-04_to_2012-05-04 zipcode cost.xml");

var query =       from x in xml.Descendants(a + "feed").Elements(a + "entry")
                .Elements(a + "content").Elements(e + "IntervalBlock").Elements(e + "IntervalReading")
        where uRef.AddSeconds((long)x.Element(e + "timePeriod").Element(e + "start")) >= startTime
                  &&
        uRef.AddSeconds((long)x.Element(e + "timePeriod").Element(e + "start")) < endTime
        select new {
                                  eventDate = uRef.AddSeconds(long.Parse(x.Element(e + "timePeriod")
                                          .Element(e + "start").Value)),
                  dataValue = (x.Element(e + "value") == null
                                          ? 0 : int.Parse(x.Element(e + "value").Value)),
                  dataCost = (x.Element(e + "cost") == null
                                          ? 0 : int.Parse(x.Element(e + "cost").Value))
                                };
      //query.Dump();

var grp = from q in query
              group q by new {q.eventDate.Date, q.eventDate.Hour} into g
              select new {eventDate = g.Key.Date.AddHours((double)g.Key.Hour), dataValue = g.Sum(y => y.dataValue), dataCost =
g.Sum(y=> y.dataCost)* 1e-5};

grp.Dump();

// Retrieve information about the data
var typeQuery =
      from x in xml.Descendants(a + "feed").Elements(a + "entry").Elements(a + "content")
            .Elements(e + "ReadingType")
     select new {
                  multiplier = x.Element(e + "powerOfTenMultiplier").Value,
                  uom = x.Element(e + "uom").Value,
                  currency = x.Element(e + "currency")==null?0:int.Parse(x.Element(e + "currency").Value)
            };
typeQuery.Dump();

double cost = grp.Sum(n=>n.dataCost);
int value = grp.Sum(n=>n.dataValue)/1000;
Console.WriteLine("Energy: {0} kwh, Cost {1:C2}",value,cost);
```


EnerNex LLC
620 Mabry Hood Road, Suite 300 || Knoxville, TN 37932
Telephone: 865-218-4600 || Facsimile: 865-218-8999 || Website: www.enernex.com

1.8 Using the extracted data

The data from the query can be displayed many ways, depending on your application's needs.

- Use a graphing package like ZedGraph (http://zedgraph.sourceforge.net), to display charts. The var grp object is an IEnumerable collection and you can iterate over its elements with a foreach loop.

```
foreach (var dp in qry)
{
    // Load each dataPoint element into the graph
}
```

- List values in an object like a .NET GridView or ListView.
- Show a single value from LINQ's Sum() method, as demonstrated.

1.9 Metadata

How do I know these values are in watt hours? It's in the document. Here's a sample query that retrieves the units and multiplier, as well as the currency being used for the cost:

```
var typeQuery =
    from x in xml.Descendants(a + "feed").Elements(a + "entry").Elements(a + "content")
        .Elements(e + "ReadingType")
    select new {
            multiplier = x.Element(e + "powerOfTenMultiplier").Value,
            uom = x.Element(e + "uom").Value,
            currency = x.Element(e + "currency")==null
                ?0:int.Parse(x.Element(e + "currency").Value)
        };
```

In this sample document the multiplier is zero. The multiplier value is a power of 10 so this case the multiplier is $10^0 = 1$. These values are in tables in the ESPI schema (included in the package). In this sample file the Unit of Measure (uom) code is 72, which represents Watt Hours (also defined in the schema). The Green Button specification there are many values possible, so Green Button can be used for natural gas, water, or other utilities. The currency value 840 indicates that the cost values in this file are in U.S. Dollars. Note that the currency code might not always be present, just as the cost may not be present, so the sample code has to handle that possibility.

1.10 Compression

Some Green Button XML documents may be compressed, typically with the ZIP format. There are .NET .ZIP libraries that can be used to extract the .XML payload. I've used DotNetZip (http://dotnetzip.codeplex.com). A complete discussion of using DotNetZip or some other compression package is beyond the scope of this article, but I feel that some coverage of this subject is important. I have encountered files from one utility that were compressed but the

EnerNex LLC
620 Mabry Hood Road, Suite 300 || Knoxville, TN 37932
Telephone: 865-218-4600 || Facsimile: 865-218-8999 || Website: www.enernex.com

filename still had an .XML file extension. This means the extension cannot be relied upon to determine whether compression is being used. One way this problem can be handled is by attempting to open the file as a zip file, and if the compression package throws an exception then assume that the file is an actual XML file and open it normally.

1.11 Conclusion

This article is intended to introduce developers to the Green Button format and also to demonstrate how LINQ can make it easier to use. The next step is to design useful applications to allow users to access their energy usage data and help them interpret their meaning. Development of practical applications is left to the reader.

It should be remembered that this example omits most of the error-checking that should be done in a complete application.

EnerNex LLC
620 Mabry Hood Road, Suite 300 || Knoxville, TN 37932
Telephone: 865-218-4600 || Facsimile: 865-218-8999 || Website: www.enernex.com