Module 2 - Querying XML

Overview

In the previous module we saw how you can create XML using the XmlDocument class. An XmlDocument object is an in-memory representation of XML. You use it to build the XML step by step. When done, you can persist it to a file or just a string that can be passed from one component to another.

The XmlDocument class implements the W3C XML Document Object Model. The way the class works is not really specific to C#. Other languages have classes that also implement DOM. So learning this way of building XML helps you when you want to build XML in other languages as well (Java, Python, etc.).

Let's consider again a system where A sends XML to B (Figure 1). A and B can be anything. For example A can be a cloud Lambda function. In Module 1 we learned how A can build the XML. This module concentrates on how B can consume the XML. To consume XML in a fast and reliable way, we need a consistent method of parsing and searching XML, similar in a way of how we search a database.

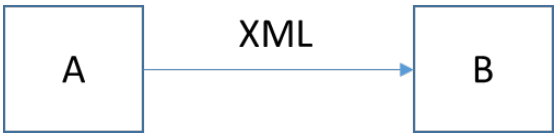

Figure 1: A system where component A builds XML and sends it to B.

XPath

When we search a relational database, we usually use a platform-neutral, language-neutral, and vendor-agnostic language called SQL. The equivalent of SQL for XML is XPath (XML Path Language). XPath uses a non-XML syntax that allows you to navigate and efficiently search an XML document. Like DOM, XPath is a web standard and its W3C Recommendation can be found here. Read the Abstract paragraph in the link which gives you a brief description of what it is.

In the few exercises below we will learn XPath by example. We won't cover every possible query syntax, but a few important ones that do the job for the majority of cases. Once you learn the pattern of using XPath, you can experiment with more advanced ones. Enclosed are two test files that are used in the examples (obtained from the MSDN website). They are bigger than the ones you created in Module 1. Drag each into a browser to see what it contains.

CustomersOrders.xml
PurchaseOrders.xml

Querying with XPath

For the exercises below, you do not need to create a new Visual Studio solution. Instead, You can use the XMLTutorial solution you created in Module 1.

1. Create a new project of type Console App (.NET Core) and name it XmlSearcher1. Make the project the startup project by right-clicking it and choosing menu **Set as Startup Project**.

   To file program.cs, add namespace System.Xml, and remove the Hello World print statement.

2. In the Solution Explorer pane, right-click on project XmlSearcher1 and choose menu **Add → New Folder**. Name the folder **Data**.

3. Right-click on the Data folder and choose menu **Add → Existing Item…**
   Browse to file PurchaseOrders.xml (included in this module) and select it. Visual Studio adds it to the Data folder.

4. In the Solution Explorer right-click PurchaseOrders.xml and select menu **Properties**.
   Set the value of property **Copy to Output Directory** to **Copy if newer**.
   Close the Properties window (or just leave it is anchored next to the Solution Explorer).

   What the above does is copy file PurchaseOrders.xml to the output directory (usually bin/debug, bin/release, or some other folder under bin). And because the file is in the output directory (where the executable XmlSearcher1.exe will be too), we can open the file by just using its relative path (which is "Data/PurchaseOrders.xml").

5. First test that you can load the PurchaseOrders.xml file. Try this code in Program.cs:

```
static void Main(string[] args)
{
    XmlDocument xmlDoc = new XmlDocument();     // Create an XmlDocument object
    xmlDoc.Load(@"Data/PurchaseOrders.xml");    // xmlDoc is now the in-memory representation of PurchaseOrders.xml

    Console.WriteLine(xmlDoc.DocumentElement.OuterXml);
    Console.ReadLine();
}
```

   Run the program. If you see the content of the file in the command window then the first two lines worked as expected.

   A few things to notice in the code:

   a. In Module 1 we used XmlDocument to build XML. The same object can be used to load an existing XML. The **Load** method loads XML from a file. The **LoadXml** method (xmlDoc.LoadXml) loads XML from a string. You use the appropriate method depending how XML is available to you (as a file or as a string).

b. Notice that we are using the relative path of the file ("Data/PurchaseOrders.xml"). It works because we told Visual Studio to copy that file to the output directory (what you did in step 4).

c. Notice the @ symbol in front of string Data/PurchaseOrders.xml: **@**"Data/PurchaseOrders.xml".

   The @ symbol makes the string be interpreted verbatim and all escape characters, line breaks, etc. are ignored. It is a path and we want that. We don't want slashes combined with some other characters (e.g., \n, etc.) to have any special meaning.

6. The Load method worked. You can delete the last 2 lines in the code. Just leave the first two lines as they are:

```
XmlDocument xmlDoc = new XmlDocument();     // Create an XmlDocument object
xmlDoc.Load(@"Data/PurchaseOrders.xml");    // xmlDoc is now the in-memory representation of PurchaseOrders.xml
```

   We will now experiment with querying PurchaseOrders.xml, now represented in memory by the xmlDoc object.

7. Open PurchaseOrder.xml and notice that it has 3 purchase orders (3 <PurchaseOrder> elements). We will use XPath to get those. Add the following code:

```
static void Main(string[] args)
{
    XmlDocument xmlDoc = new XmlDocument();     // Create an XmlDocument object
    xmlDoc.Load(@"Data/PurchaseOrders.xml");    // xmlDoc is now the in-memory representation of PurchaseOrders.xml

    XmlElement root = xmlDoc.DocumentElement;

    // Query the XML
    XmlNodeList results = root.SelectNodes("PurchaseOrder");        // What appears inside the quotes (PurchaseOrder) is the XPath query.
                                                                    // This query is very simple and means:  give me all <PurchaseOrder>
                                                                    // elements that are under root (root represents <PurchaseOrders> (with s))
    if (results != null)
    {
        Console.WriteLine("Found {0} orders\n", results.Count);     // Print how many <PurchaseOrder> were found

        // Print the content of each individual <PurchaseOrder> element
        foreach (XmlNode order in results)
        {
            Console.WriteLine("{0}\n", order.OuterXml);
        }

    }
}
```

   Run it and verify its output.

8. Once we find each <PurchaseOrder> element, we can drill down in a similar way to retrieve additional data. For example, let's assume I want to retrieve the delivery notes of all orders (look at <DeliveryNotes> in the XML). Change the above code to look like this:

```
if (results != null)
{
    foreach (XmlNode order in results)
    {
        XmlNode deliveryNote = order.SelectSingleNode("DeliveryNotes");      // Another XPath. Now querying under a sub-element
        if(deliveryNote != null)
        {
            Console.WriteLine("{0}", deliveryNote.InnerText);
        }

    }
}
```

Run it and verify its output.

9. We can change the XPath query syntax to immediately find all DeliveryNotes without needing to first find all <PurchaseOrder> elements. Try it:

```
static void Main(string[] args)
{
    XmlDocument xmlDoc = new XmlDocument();     // Create an XmlDocument object
    xmlDoc.Load(@"Data/PurchaseOrders.xml");    // xmlDoc is now the in-memory representation of PurchaseOrders.xml

    XmlElement root = xmlDoc.DocumentElement;

    // Query the XML. The Xpath query "PurchaseOrder/DeliveryNotes" means:
    // Give me DeliveryNotes elements under any PurchaseOrder element under root (because I am calling SelectNodes from root)
    // Notice the forward slash in XPath (used to separate parent/child).
    XmlNodeList results = root.SelectNodes("PurchaseOrder/DeliveryNotes");

    if (results != null)
    {
        foreach (XmlNode note in results)
        {
            Console.WriteLine("{0}", note.InnerText);
        }
    }
}
```

10. Note that in some case I am using SelectNodes and in other cases I am using SelectSingleNode. Read the documentation to learn about the difference (links below):

    SelectSingleNode
    SelectNodes

    If you are expecting multiple values of something then you should use SelectNodes. Can one make this assumption? Usually yes because usually the consumer of the XML is aware of the domain and business rules of whatever the XML is representing. Just like when you query a table or multiple tables in a database you usually know the layout of the tables, the entities they represent in your application, and the domain and business rules of the application. For example, as a developer I would typically know if say a database supports customers having multiple shipping addresses instead of one.

    Let's now modify the code to print all zip codes. Update the code as shown below:

```
static void Main(string[] args)
{
    XmlDocument xmlDoc = new XmlDocument();     // Create an XmlDocument object
    xmlDoc.Load(@"Data/PurchaseOrders.xml");    // xmlDoc is now the in-memory representation of PurchaseOrders.xml

    XmlElement root = xmlDoc.DocumentElement;

    XmlNodeList results = root.SelectNodes("PurchaseOrder/Address");
    if (results != null)
    {
        foreach (XmlNode address in results)
        {
            // Each <Address> element has one zip code. So I use SelectSingleNode
            XmlNode zipNode = address.SelectSingleNode("Zip");
            if(zipNode != null)
            {
                Console.WriteLine("{0}", zipNode.InnerText);
            }
        }
    }
}
```

Run it and verify the output.

11. Remember in Module 1 we said that XML is case sensitive. Try this to verify: in the code above change Zip to zip (lowercase z). That is:

```
XmlNode zipNode = address.SelectSingleNode("zip");
```

Run it. You will notice that nothing got printed. That's because XML treats <Zip> as different from <zip>

> **Is XML case-sensitive?** Is a common interview question.
> It tells the interviewer if you have worked with XML before or not.

12. Let's now experiment with XPath that is a bit more advanced. Look at each <PurchaseOrder> element and notice that it has a PurchaseOrderNumber attribute. Now let's assume I am interested in purchase order number 99505. Let's retrieve this order using XPath. Change the code to look like this.

```
XmlElement root = xmlDoc.DocumentElement;

string orderNumber = "99505";

// I suspect that purchase order numbers are unique. So I am using SelectSingleNode and XmlNode
XmlNode purchase = root.SelectSingleNode("PurchaseOrder[@PurchaseOrderNumber=\"" + orderNumber + "\"]");
if (purchase != null)
{
    // Find how many items are in this order.
    // I suspect than an order can have many items in it. So I will use SelectNodes and XmlNodeList (not XmlNode)
    XmlNodeList items = purchase.SelectNodes("Items/Item");
    if(items != null)
    {
        Console.WriteLine("There are {0} items in order number {1}", items.Count, orderNumber);
    }
}
```

Run and verify that order 99505 has 1 item. Look at the XML and verify that it is the case.

13. Change the value of variable orderNumber to the other orders in the XML (e.g., 99504). Does the output show the correct items number?

How can I find the correct XPath syntax to use?
Look at documentations. This cheat sheet is useful.

Look for example in the **Attribute selectors** table in the above link. Notice this:
`input[@type="submit"]`

This is what we used in step 12. The XPath
"PurchaseOrder[@PurchaseOrderNumber=\"" + orderNumber + "\"]"

evaluates to a string that matches the syntax shown above.

14. Note that the code in step 12 can also be done by one single XPath query instead of two. Try this:

```
string orderNumber = "99505";

// This XPath says:  Find under PurchaseOrder with PurchaseOrderNumber="99505" all Item elements under Items
XmlNodeList items = root.SelectNodes("PurchaseOrder[@PurchaseOrderNumber=\"" + orderNumber + "\"]/Items/Item");
if (items != null)
{
    Console.WriteLine("There are {0} items in order number {1}", items.Count, orderNumber);
}
```

Run it and observe the output. Try different order number values.

Exercise to Do On Your Own

1. In the same solution, create a new project of type Console App (.NET Core) and name it **XmlSearcher2**.

   Add a **Data** folder (like it was done in XmlSearcher1).
   Add file **CustomersOrders.xml** under Data. Set its **Copy to Output Directory** to **Copy if newer**.
   Make XmlSearcher2 the startup project

2. Write code that outputs the following:

   a. Phone and fax numbers of customer with ID “LAZYK”.

b. Full address of customer with ID “LETSS”. Output the address in a format that resembles this:

   John Smith
   10 Main Rd
   Bellevue, WA 98004

c. Total number of customers.
d. Total number of orders.
e. Number of orders for customer with ID “LAZYK”
f. Number of orders to the city of Eugene

What to Submit

Nothing to submit. Complete quiz **Module2-Quiz**.