SRP_Extract_Xml

Extracts elements and/or data from XML.

Syntax

```
Result = SRP_Extract_Xml(Xml, XPath, NameSpaces)
```

Returns

The XML fragment if found, or an error string prefixed with "ERROR: " if not. (See Error Handling below).

Parameters

Parameter	Description
Xml	The well-formed xml (whole or fragment) from which the extraction is to take place.
XPath	The XPath query specifying the fragment or data to be extracted.
NameSpaces	A space delimited list of XML namespaces expected to be encountered within the XML.

Remarks

The developers at SRP Computer Solutions, Inc. found themselves dealing with XML on an increasingly more frequent basis. While BASIC+ makes building XML easy, parsing XML requires a bit more work and quite a lot of assumptions. SRP_Extract_Xml takes advantage of the XPath query language to make data extraction from XML a one-line operation.

Under the hood, SRP_Extract_Xml uses the MSXML library, which is already installed on all up-to-date machines, so there should be no need to worry about depoloyment.

Xml

The Xml property expects a well formed XML document or fragment. You can pass "" to indicate that you'd like to continue using the previously loaded XML, but be aware of one very important caveat. If one call SRP_Extract_Xml does not yield results, then all subsequent calls in which the Xml parameter is set to "" will fail to yield results as well. This seems to be a quirk of MSXML, and the safest bet is to always set the Xml parameter to the XML fragment from which you wish to extract.

XPath

The XPath parameter expects the XPath query string. XPath looks much like a file path, drilling down into XML elements until you find the one you're looking for. XPath is far too broad a subject to teach here, but w3schools.com has excellent tutorials here.

Once you understand the XPath query language, simple pass the XML data in question and a query to extract the data you want. To get only an element's contents, append the query with "/text()".

```
<catalog>
    <book id="bk101">
        <author>Gambardella, Matthew</author>
        <title>XML Developer's Guide</title>
        <genre>Computer</genre>
        <price>44.95</price>
        <publish_date>2000-10-01</publish_date>
        <description>An in-depth look at creating applications
        with XML.</description>
    </book>
</catalog>
```

Given an XPath query of "/catalog/book/title", the return value would be:

```
<title>XML Developer's Guide</title>
```

However, append "/text()" at the end of the query, thus making the query "catalog/book/title/text()", the return value is:

```
XML Developer's Guide
```

This information is in the w3schools.com tutorials, but it an important point to mention to first time XPath users.

NameSpaces

XPath can act a little goofy when XML namespaces are involved. Normally, XML parsers recognize imbedded namespaces and know what to do with them, but XPath requires you to list namespaces in advance. Consider the following SOAP-XML snippet:

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
   <s:Body>
       <GetClient xmlns="http://www.srpcs.com">
           <clientId xmlns="">1</clientId>
       </GetClient>
   </s:Body>
</s:Envelope>
```

Notice the namespace is set three times. The SOAP namespace is set to 's', and the default namespace is set to the SRP Computer Solutions, Inc. website. The default namespace is changed again later to null. Without setting namespaces in advance, the following query will fail:

```
XPath = "/s:Envelope/s:Body/GetClient/clientId/text()"
Result = SRP_Extract_Xml(Xml, XPath)
```

The 's' namespace, while embedded in the XML itself, is not recognized by the XPath query. We have to specify the namespace in advance. We also have to specify the default namespace. So, it would seem that our query should be (note the space between each namespace):

```
XPath = "/s:Envelope/s:Body/GetClient/clientId/text()"
NameSpaces  = "xmlns:s='http://schemas.xmlsoap.org/soap/envelope/' "
NameSpaces := "xmlns='http://www.srpcs.com'"
Result = SRP_Extract_Xml(Xml, XPath, NameSpaces )
```

However, this will fail also. That is because XPath gets confused with the default namespace being changed within the XML. The following query does work:

```
XPath = "/s:Envelope/s:Body/srp:GetClient/clientId/text()"
NameSpaces  = "xmlns:s='http://schemas.xmlsoap.org/soap/envelope/' "
NameSpaces := "xmlns:srp='http://www.srpcs.com'"
Result = SRP_Extract_Xml(Xml, XPath, NameSpaces )
```

Notice that we named the srpcs.com namespace 'srp'. Then, in the XPath query, we use the 'srp' namespace to distinguish between the GetClient tag and the clientId tag. It should be noted, the namespaces passed to the SRP_Extract_Xml do not need to be named the same as the namespaces used within the XML itself.

XML namespaces can get confusing. If you need to read up on them, you can read w3schools tutorial on the subject here.

Multiple Node Results

Sometimes, a query returns more than one element. In this case, the elements are returned in an @FM delimited array. This is particularly useful when extracting an array of information. Consider the following XML:

```
<employees>
    <employee>Don</employee>
    <employee>Frank</employee>
    <employee>Paul</employee>
    <employee>Kevin</employee>
    <employee>Roger</employee>
</employees>
```

The following query:

```
Result = SRP_Extract_Xml(Xml, "/employees/employee/text()" )
```

Returns the following data:

```
<1> Don
<2> Frank
<3> Paul
<4> Kevin
<5> Roger
```

Even if you don't use "/text()" to get just the element contents, all fragments will still be @FM delimited. This conveniently allows you to use standard BASIC+ logic to count and iterate through your results. If we remove "/text()" from the previous query, we get these results:

```
<1> <employee>Don</employee>
<2> <employee>Frank</employee>
<3> <employee>Paul</employee>
<4> <employee>Kevin</employee>
<5> <employee>Roger</employee>
```

Error Handling

Many thanks to Dave Harmacek for notifying us of a glaring omission in our documentation: how error handling is reporting.

We really like it when our tools return errors in human readable sentences rather than enigmatic codes. Since OI is so great at string parsing, we opted to use the return value itself to report errors. If there was an error, then the return value is prefixed with "ERROR: ". So, error handling looks like this:

```
Result = SRP_Extract_Xml(Xml, "/employees/employee/text()" )
If Result[1, 6] EQ "ERROR:" then
   // Report error here
end else
   // All is well, so Result contains our extracted XML
end
```