Understanding XML Namespaces

Understanding XML namespaces is essential to understanding and building Web services. Unfortunately, most developers do not understand how XML namespaces work and get confused when they see all the colons and URLs. The funny thing is, namespaces are so simple that once you understand them you wonder why you had trouble understanding them in the first place. I wrote this article to help you get over the initial hump of understanding XML namespaces, and to show you how they are used in XML documents. This article is geared towards Visual Basic developers. If your work involves HTML more than programming, there’s an excellent namespace tutorial that speaks to your needs.

Why Namespaces

Namespaces have been a W3C standard since January 1999. The idea behind namespaces is simple: We need a way to fully qualify XML element and attribute names to prevent from confusing two elements that have the same name but mean different things. For example, you might have an accounting application that uses an element called <schedule> to mean an accounting schedule. A time management application might use the same element <schedule> to mean a time schedule. You might need to put accounting and time management data in one XML document. For example, you might be keeping track of all project information including accounting and project timeline in the same document. Now each application would not be able to tell which <schedule> element belongs to it. To solve this problem, you could prefix each <schedule> element with a unique prefix that indicates to which application it belongs. For example <accounting:schedule> and <time:schedule> would make things much clearer. In this example accounting and time are namespaces.

Namespaces are simply a way to uniquely identify elements and attributes within a document.

Namespaces and Validation

The role namespaces play is most obvious when you are validating an XML document against an XML Schema Definition (XSD) XSD schema. In case you are not familiar with XSD, it is an XML-based grammar used to define document structures and data types that you use in your document. You can think of XSD as a superset of Document Type Definitions (DTD). You don’t need to know much about XSD schemas for this article and I’ll explain the little bit you do need to know.

Imagine you have an XML document that contains employee information for a human resources application:

```xml
<employees>
  <employee>
    <id>49982</id>
    <name>Bart Simpson</name>
    <hireDate>2000-07-04</hireDate>
  </employee>
  <employee>
    <id>12345</id>
    <name>Hugo Simpson</name>
    <hireDate>2000-05-29</hireDate>
  </employee>
</employees>
```

You might create a schema for this document that defines a data type for the employee element like this:

```xml
<xsd:complexType name="employeeType">
  <xsd:sequence>
    <xsd:element name="id" type="xsd:int"/>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="hireDate" type="xsd:date"/>
  </xsd:sequence>
</xsd:complexType>
```
Without going into the details of XSD, the above snippet does two things: First, it defines a data type called employeeType that contains three elements: id, name and hireDate. Second, it declares an element called employee of the type employeeType. This is the XSD way of saying "the <employee> element will contain three elements in this order: <id>, <name>, and <hireDate>".

You can use the above schema snippet to validate the employees document and it’ll work just fine (provided you have a schema-aware validator like XML Spy V3.5). Now, imagine the payroll application wants to share this employee information and add some more to it. For example, the payroll application wants to keep track of employee salary and the taxes being deducted (this is way oversimplified, but who wants to learn the 2001 payroll taxes laws?):

```xml
<employees>
<employee>
  <id>49982</id>
  <name>Bart Simpson</name>
  <hireDate>2000-07-04</hireDate>
  <salary>4000765.00</salary>
  <taxes>3980765.27</taxes>
</employee>
<employee>
  <id>12345</id>
  <name>Hugo Simpson</name>
  <hireDate>2000-05-29</hireDate>
  <salary>82000.00</salary>
  <taxes>16567.87</taxes>
</employee>
</employees>
```

How should you handle this? Do you update the HR schema to reflect the new <salary> and <taxes> elements? That might seem like a good choice at first, but it results in two separate applications sharing the same schema document, which is likely to result in ownership and maintenance problems. It would be much better if you can separate the data types that belong to HR from the data types that belong to payroll and allow each application’s team to have control over there data types with no potential of messing up each other’s schemas. You can do that by simply placing those data types in different buckets when you define them. Those buckets are called namespaces. Lets say you define a bucket or namespace called HRData and another one called payrollData. You can then make the payroll application team in charge of maintaining data types in the payrollData namespace and the HR application team in charge of maintaining types in the HRData namespace. You will need a way to indicate that the <salary> and <taxes> elements belong to the payrollData namespace while all other elements belong to the HRData namespace. To do this you prefix each element name with the namespace and a colon like this:

```xml
<HRData:employees>
  <HRData:employee>
    <HRData:id>49982</HRData:id>
    <HRData:name>Bart Simpson</HRData:name>
    <HRData:hireDate>2000-07-04</HRData:hireDate>
    <payrollData:salary>4000765.00</payrollData:salary>
    <payrollData:taxes>3980765.27</payrollData:taxes>
  </HRData:employee>
  <HRData:employee>
    <HRData:id>12345</HRData:id>
    <HRData:name>Hugo Simpson</HRData:name>
    <HRData:hireDate>2000-05-29</HRData:hireDate>
    <payrollData:salary>82000.00</payrollData:salary>
    <payrollData:taxes>16567.87</payrollData:taxes>
  </HRData:employee>
</HRData:employees>
```

Don’t be fooled by the apparent complexity of this snippet. All I did is add the HRData and payrollData prefixes before each element name. I don’t know about you, but I’d rather keep the namespace prefixes as short as possible. To do this, you come up with a short prefix,
possibly as short as one letter, and map that prefix to the real namespace name. For example, you might decide to use py for payrollData and hr for HRData:

```xml
<hr:employees xmlns:hr="HRData" xmlns:py="payrollData">
  <hr:employee>
    <hr:id>49982</hr:id>
    <hr:name>Bart Simpson</hr:name>
    <hr:hireDate>2000-07-04</hr:hireDate>
    <py:salary>4000765.00</py:salary>
    <py:taxes>3980765.27</py:taxes>
  </hr:employee>
  
  <hr:employee>
    <hr:id>12345</hr:id>
    <hr:name>Hugo Simpson</hr:name>
    <hr:hireDate>2000-05-29</hr:hireDate>
    <py:salary>82000.00</py:salary>
    <py:taxes>16567.87</py:taxes>
  </hr:employee>
</hr:employees>
```

The syntax for defining a namespace-prefix mapping is: `xmlns:prefix="namespace"` where `prefix` is the short prefix you'll use in the document and `namespace` is the actual namespace name that the prefix refers to. Once you've defined the prefix, you can use it in your document instead of writing out the entire namespace name in front of each element name. When using namespaces, element and attribute names have two parts: the `prefix` e.g. hr or py and the `local name` e.g. employee or salary. The two parts together form the qualified name or QName, e.g. hr:employee or py:salary.

Now you can easily create two different schemas, one that defines the HR types in the HRData namespace, and one that defines the payroll types in the payrollData namespace. The syntax you use to do this is part of XSD and is beyond the scope of this article.

### Unique Names

The names payrollData and HRData are descriptive but they are also very common and two different organizations could potentially use the name payrollData for two different namespaces. This might become a problem if you start exchanging XML documents with other organizations. For example, you might want to send the payroll information to the IRS in XML. So you’d send them an XML document with a bunch of elements that belong to the payrollData namespace. If for some reason the IRS decides to process your payroll data along with payroll data from other companies, its likely another company will also have used the payrollData namespace because that name is just not unique. Its also likely that this company’s payroll document structure is different from yours. Bottom line: You’d have a situation where an application needs to deal with two different namespaces that have the same name.

To prevent such situations, namespace names must be globally unique. Technically, a namespace must be a Uniform Resource Identifier as defined in IETF RFC 2396. Basically, a URI is a unique string that can be classified as a locator (a Uniform Resource Locator or URL) or a name (a Uniform Resource Name or URN) or both. We all know URLs and use them regularly to access Web pages and other Internet resources. For example, `http://www.devxpert.com/` is a URL. A URN is globally unique resource name that sticks around forever and never goes away even if the resource itself no longer exists. For example, you could use a Universally Unique Identifier (UUID) as a namespace like this:

```
uuid:EB3859CB-0867-4053-9C40-CF9DED28C33F
```

This is considered a URN and is therefore a valid namespace name. So if you pick that as your namespace name for the payroll data (instead of payrollData), the XML document would become:

```xml
<hr:employees xmlns:hr="HRData" xmlns:py="uuid:EB3859CB-0867-4053-9C40-CF9DED28C33F">
  <hr:employee>
    <hr:id>49982</hr:id>
  </hr:employee>
</hr:employees>
```
All I did was change the namespace prefix mapping for the py prefix to point to the new namespace name. You can also use a URL as your namespace name, for example, http://www.devxpert.com/schemas/payroll/ for the payroll data and http://www.devxpert.com/schemas/hr/ for the HR data:

```xml
<employees xmlns:hr="http://www.devxpert.com/schemas/hr/"
            xmlns:py="http://www.devxpert.com/schemas/payroll/">
  <employee>
    <id>49982</id>
    <name>Bart Simpson</name>
    <hireDate>2000-07-04</hireDate>
    <py:salary>4000765.00</py:salary>
    <py:taxes>3980765.27</py:taxes>
  </employee>
  <employee>
    <id>12345</id>
    <name>Hugo Simpson</name>
    <hireDate>2000-05-29</hireDate>
    <py:salary>82000.00</py:salary>
    <py:taxes>16567.87</py:taxes>
  </employee>
</employees>
```

Note that the URL need not point to anything on the Internet. It is just used as a unique string. This is a source of confusion for most people who tend to think that there’s some document located at this URL. Just keep in mind a namespace name is simply a unique string, nothing more nothing less.

**Default Namespace**

If most of the elements in the document belong to the same namespace, it is not necessary to prefix each element name. Instead you can define a default namespace that applies to all non-prefixed elements and attributes. The syntax for defining a default namespace is `xmlns="namespace"`. For example, if you define the HR namespace as the default namespace, the document becomes easier to read:

```xml
<employees xmlns="http://www.devxpert.com/schemas/hr/"
            xmlns:py="http://www.devxpert.com/schemas/payroll/">
  <employee>
    <id>49982</id>
    <name>Bart Simpson</name>
    <hireDate>2000-07-04</hireDate>
    <py:salary>4000765.00</py:salary>
    <py:taxes>3980765.27</py:taxes>
  </employee>
  <employee>
    <id>12345</id>
    <name>Hugo Simpson</name>
    <hireDate>2000-05-29</hireDate>
    <py:salary>82000.00</py:salary>
    <py:taxes>16567.87</py:taxes>
  </employee>
</employees>
```

The default namespace applies to the element on which it was defined and all descendants of that element. If one of the descendants has another default namespace defined on it, this new namespace definition overrides the previous one and becomes the default for that namespace.
element and all its descendants. For example, in the following document the default namespace for the `<salary>` element (and its descendants if it had any) is http://www.devxpert.com/schemas/payroll/:

```xml
<employees xmlns="http://www.devxpert.com/schemas/hr/
xmlns:py="http://www.devxpert.com/schemas/payroll/">
  <employee>
    <id>49982</id>
    <name>Bart Simpson</name>
    <hireDate>2000-07-04</hireDate>
    <salary xmlns="http://www.devxpert.com/schemas/payroll/">
      4000765.00
    </salary>
    <py:taxes>3980765.27</py:taxes>
  </employee>
</employees>
```

### Namespaces and attributes

You can prefix attributes names with a namespace prefix just like element names. For example, in the following snippet, the value attribute of the `<salary>` element belongs to the payroll namespace http://www.devxpert.com/schemas/payroll/:

```xml
<employees xmlns="http://www.devxpert.com/schemas/hr/"
xmlns:py="http://www.devxpert.com/schemas/payroll/">
  <employee>
    <id>49982</id>
    <name>Bart Simpson</name>
    <hireDate>2000-07-04</hireDate>
    <py:salary py:value="4000765.00"></py:salary>
    <py:taxes>3980765.27</py:taxes>
  </employee>
</employees>
```

However, default the namespaces does not apply to attribute names. For example, in the following snippet, the `id` attribute does not belong to any namespace even though there's a default namespace declared:

```xml
<employee xmlns="http://www.devxpert.com/schemas/hr/" id="90385">
  </employee>
```

I hope that this article has helped you understand XML Namespaces and how they work. If you're not familiar with XSD schemas, stay tuned for my upcoming articles on this topic. Learning XSD will help you fully understand XML Namespaces.