By Harish Kamath

Table of Contents

What's In A Name?

Shakespeare once famously asked, "What's in a name?". Not much − unless you're dealing with XML, in which case, quite a lot.

Everyone knows that XML allows you to use descriptive tags to mark up text in a document. These tags are usually free−form; a document author has complete freedom to name these tags anything he or she desires. And while this flexibility is one of the reasons for XML's popularity, it's a double−edged sword, because it begs the question: what happens if tag names in different documents clash with each other?

In the concluding article in this series, I'm going to be answering this question by spending some time on the oft−misunderstood concept of XML namespaces, in an effort to clear the air about what they are, how they work, and why you should use them. Along the way, I'll also show you how to use schemas to enforce namespace usage in an XML document instance, on a global or case−by−case basis, for both elements and attibutes.

Stocking Up

First, the basics. A namespace is simply a prefix added to an element in order to distinguish it from other elements with the same name.

An example might help to make this clearer. Let's suppose that I decided to encode my stock portfolio as an XML document. Here's what it might look like:

```
<?xml version="1.0"?>
<portfolio>

<stock>Cisco Systems</stock>
<stock>Nortel Networks</stock>
<stock>eToys</stock>
<stock>IBM</stock>

</portfolio>
```

And now let's suppose that Tom, my next−door neighbour and the proud owner of his own computer store, hears about XML, gets really excited, and assigns some of his employees to the task of encoding his store's inventory into XML. Here's what his XML document might look like:

```
<?xml version="1.0"?>
<inventory>
<item name="Mouse C106">
<vendor>Logitech</ vendor>
<stock>100</stock>
</item>
<item name="Visor Deluxe">
<vendor>HandSpring</vendor>
<stock>23</stock>
</item>
<item name="Nomad">
<vendor>Creative</vendor>
<stock>2</stock>
</item>
</inventory>
```

Finally, let's suppose that Tom and I get together for a drink, tell each other about our XML experiments and (in a moment of tequila−induced clarity) decide to put XML's capabilities to the test by combining our two documents into one. However, since both documents include a tag named

```
<stock>
```

whose meaning is entirely dependent on its context, it's pretty obvious that our attempt at integration will fail, since an XML application would have no way of telling whether the data enclosed between <stock>...</stock> tags belonged to my portfolio or Tom's inventory.

It's precisely to avoid this kind of ambiguity that the XML specification now provides for namespaces. Namespaces are a way to uniquely identify specific elements within an XML document. This is accomplished by assigning a unique prefix to an element, thereby immediately associating it with a particular data universe and eliminating ambiguity.

The Name Game

Setting up a namespace is simple − here's the syntax:

```
<elementName xmlns:prefix="namespaceURL">
```

In this case, the prefix is the unique string used to identify the namespace; this is linked to a specific namespace URL.

A namespace is usually declared at the root element level, although authors are free to declare it at a lower level of the tree structure too.

Once the namespace has been declared within the document, it can be used by prefixing each element within that namespace with the unique namespace identifier. Take a look at my revised stock portfolio, which now uses a "mytrades" namespace to avoid name clashes.

```
<mytrades:portfolio
xmlns:mytrades="http://www.somedomain.com/namespaces/mytrades/">

<mytrades:stock>Cisco Systems</mytrades:stock>
<mytrades:stock>Nortel Networks</mytrades:stock>
<mytrades:stock>eToys</mytrades:stock>
<mytrades:stock>IBM</mytrades:stock>

</mytrades:portfolio>
```

And once Tom gets over his hangover, I guess he'd find it pretty easy to fix his document too.

```
<?xml version="1.0"?>
<ts:inventory>
<ts:item ts:name="Mouse C106">
<ts:vendor>Logitech</ts:vendor>
<ts:stock>100</ts:stock>
```

```
</ts:item>

<ts:item ts:name="Visor Deluxe">
<ts:vendor>HandSpring</ts:vendor>
<ts:stock>23</ts:stock>
</ts:item>

<ts:item ts:name="Nomad">
<ts:vendor>Creative</ts:vendor>
<ts:stock>2</ts:stock>
</ts:item>
</ts:inventory>
```

In case you're wondering, the namespace URL is simply a pointer to a Web address, and is meaningless in practical terms; the XML specification doesn't really care where the URL points, or even if it's a valid link.

All About Character

Here's an XML document instance which uses namespaces to qualify each element,

```
<?xml version="1.0" encoding="UTF-8"?>
<sw:gallery
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:sw="http://www.somedomain.com/ns/sw/"
xsi:schemaLocation="http://www.somedomain.com/ns/sw/starwars.xsd">
<sw:character>
<sw:name>Luke Skywalker</sw:name>
<sw:species>Human</sw:species>
<sw:language>Basic</sw:language>
<sw:home>Tatooine</sw:home>
</sw:character>

<sw:character>
<sw:name>Chewbacca</sw:name>
<sw:species>Wookie</sw:species>
<sw:language>Shyriiwook</sw:language>
<sw:home>Kashyyyk</sw:home>
</sw:character>

<sw:character>
<sw:name>Chief Chirpa</sw:name>
<sw:species>Ewok</sw:species>
<sw:language>Ewok</sw:language>
<sw:home>Endor</sw:home>
</sw:character>

</sw:gallery>
```

and here's the corresponding schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.somedomain.com/ns/sw/"
xmlns:sw="http://www.somedomain.com/ns/sw/"
elementFormDefault="qualified">

<!-- define a complex type -->
<xsd:complexType name="starWarsEntity">
<xsd:sequence>
<xsd:element name="name" type="xsd:string"/>
<xsd:element name="species" type="xsd:string"/>
```

```
<xsd:element name="language" type="xsd:string"/>
<xsd:element name="home" type="xsd:string"/>
</xsd:sequence>
</xsd:complexType>

<!-- define the root element and its contents -->
<xsd:element name="gallery">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="character"
type="sw:starWarsEntity" maxOccurs="unbounded"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>

</xsd:schema>
```

The most important line of code in this entire schema is the first line, which defines the namespace used to identify the elements and attributes in the XML document instance.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.somedomain.com/ns/sw/"
xmlns:sw="http://www.somedomain.com/ns/sw/"
elementFormDefault="qualified">

<!-- snip -->

</xsd:schema>
[/code]
```

The "targetNamespace" attribute specifies the namespace for this schema, and is what the XML parser/validator uses to differentiate between a <character> in the Star Wars universe, and a <character> from any other source. Note that the namespace URL in the schema matches the namespace URL in the document instance above.

Within the XML document instance itself, the "sw" namespace is defined at the top of the document as usual, together with a URL that specifies the location of the schema to be used for validation of that namespace (note that this URL is specified via the "schemaLocation" attribute).

```
<sw:gallery
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:sw="http://www.somedomain.com/ns/sw/"
xsi:schemaLocation="http://www.somedomain.com/ns/sw/sw.xsd">

<!-- snip -->
```

```
</sw:gallery>
```

Setting Policy

You can tell the validator to ensure that every element in the XML document instance is qualified with a namespace via the "elementFormDefault" attribute of the <xsd:schema> element.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.somedomain.com/ns/sw/"
xmlns:sw="http://www.somedomain.com/ns/sw/"
elementFormDefault="qualified">

<!-- snip --->

</xsd:schema>
```

This "elementFormDefault" attribute tells the validator that every element in the document instance must be qualified with a namespace identifier in order for the document to be valid. In order to verify this, you can try removing the namespace prefix from one of the <character> elements in the document instance above and validating the XML data − your validator should throw up a bunch of error messages about improperly−qualified elements.

Document authors can also satisfy this qualification requirement by specifying a default namespace for all the elements in the document instance − as in the following code segment.

```
<?xml version="1.0" encoding="UTF−8"?>
<gallery xmlns:xsi="http://www.w3.org/2001/XMLSchema−instance"
xmlns="http://www.somedomain.com/ns/sw/"
xsi:schemaLocation="http://www.somedomain.com/ns/sw/starwars.xsd">

<character>
<name>Luke Skywalker</name>
<species>Human</species>
<language>Basic</language>
<home>Tatooine</home>
</character>

<character>
<name>Chewbacca</name>
<species>Wookie</species>
<language>Shyriiwook</language>
<home>Kashyyyk</home>
</character>

<character>
<name>Chief Chirpa</name>
<species>Ewok</species>
```

```
<language>Ewok</language>
<home>Endor</home>
</character>
</gallery>
```

If you don't necessarily want to force the document author to prefix the namespace identifier to every element in each document instance, you can turn off this requirement by specifying a value of "unqualified" for the "elementFormDefault" attribute.

Old Friends And New

The XML Schema specification also notes the existence of an "attributeFormDefault" attribute of the <xsd:schema> element, which lets schema authors do to attributes what the "elementFormDefault" attribute does for elements − force qualification of each attribute with a namespace prefix. Here's an example that demonstrates how this works:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="http://www.somedomain.com/ns/sw/"
xmlns:sw="http://www.somedomain.com/ns/sw/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
attributeFormDefault="qualified">

<!-- define a complex type -->
<xsd:complexType name="starWarsEntity">
<xsd:sequence>
<xsd:element name="name" type="xsd:string"/>
<xsd:element name="species" type="xsd:string"/>
<xsd:element name="language" type="xsd:string"/>
<xsd:element name="home" type="xsd:string"/>
</xsd:sequence>
<xsd:attribute name="episode" type="xsd:string" />
</xsd:complexType>

<!-- define the root element and its contents -->
<xsd:element name="gallery">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="character"
type="sw:starWarsEntity" maxOccurs="unbounded"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>

</xsd:schema>
```

I've introduced a new attribute here for the <character> element − "episode" − which lists the Star Wars episode in which the character first appeared. My use of the "attributeFormDefault" attribute ensures that every occurrence of this attribute in an XML document instance will be prefixed with a namespace − as you can see in the following XML document:

```
<?xml version="1.0" encoding="UTF-8"?>

<sw:gallery
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:sw="http://www.somedomain.com/ns/sw/"
xsi:schemaLocation="http://www.somedomain.com/ns/sw/starwars.xsd">

<sw:character sw:episode="IV">
<sw:name>Luke Skywalker</sw:name>
<sw:species>Human</sw:species>
<sw:language>Basic</sw:language>
<sw:home>Tatooine</sw:home>
</sw:character>

<sw:character sw:episode="IV">
<sw:name>Chewbacca</sw:name>
<sw:species>Wookie</sw:species>
<sw:language>Shyriiwook</sw:language>
<sw:home>Kashyyyk</sw:home>
</sw:character>

<sw:character sw:episode="VI">
<sw:name>Chief Chirpa</sw:name>
<sw:species>Ewok</sw:species>
<sw:language>Ewok</sw:language>
<sw:home>Endor</sw:home>
</sw:character>

</sw:gallery>
```

Being Selective

The previous examples have demonstrated how a schema designer can force a document author to qualify an XML document with appropriate namespaces. However, based on what we've seen thus far, this is an all−or−nothing proposition − either every element is qualified or every element is unqualified − which is not very practical for real−world use.

Fortunately, the XML Schema specification also provides a "form" attribute for each element and attribute definition, which allows you to specify qualification rules on a case−by−case basis. Consider the following example, which demonstrates:

```
<?xml version="1.0" encoding="UTF−8"?>
<xsd:schema targetNamespace="http://www.somedomain.com/ns/sw/"
xmlns:sw="http://www.somedomain.com/ns/sw/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<!−− define a complex type −−>
<xsd:complexType name="starWarsEntity">
<xsd:sequence>
<xsd:element name="name" type="xsd:string"
form="qualified"/>
<xsd:element name="species" type="xsd:string"
form="unqualified"/>
<xsd:element name="language" type="xsd:string"
form="unqualified"/>
<xsd:element name="home" type="xsd:string"
form="qualified"/>
</xsd:sequence>
</xsd:complexType>

<!−− define the root element and its contents −−>
<xsd:element name="gallery">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="character"
type="sw:starWarsEntity" maxOccurs="unbounded"
form="qualified"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>

</xsd:schema>
```

As you can see, I've introduced the "form" attribute into the element definitions above. Like its schema−level cousins, this attribute too takes two values: "qualified" and "unqualified". In the example above, the <character>, <name> and <home> elements are all to be qualified with namespaces; the others may remain

unqualified.

And here's an XML document instance conforming to the schema above:

```
<?xml version="1.0" encoding="UTF-8"?>
<sw:gallery
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:sw="http://www.somedomain.com/ns/sw/"
xsi:schemaLocation="http://www.somedomain.com/ns/sw/starwars.xsd">

<sw:character>
<sw:name>Luke Skywalker</sw:name>
<species>Human</species>
<language>Basic</language>
<sw:home>Tatooine</sw:home>
</sw:character>

<sw:character>
<sw:name>Chewbacca</sw:name>
<species>Wookie</species>
<language>Shyriiwook</language>
<sw:home>Kashyyyk</sw:home>
</sw:character>

<sw:character>
<sw:name>Chief Chirpa</sw:name>
<species>Ewok</species>
<language>Ewok</language>
<sw:home>Endor</sw:home>
</sw:character>

</sw:gallery>
```

The XML Schema author can thus selectively enforce XML element qualification − useful if only some of the elements are likely to clash with others.

Closing Time

And that's about it for the moment. Over the four parts of this article, I've given you a crash course in some of the more advanced aspects of XML schema design, explaining how you can use built−in schema constructs to design more efficient and extensible schemas.

I've shown you how to derive new types from existing types, either by extending or restricting them, and how to break up your schema definitions into separate files for greater maintainability. I've also shown you how to enforce basic referential integrity in a schema with primary and foreign key references, and how to enforce the use of namespaces in document instances to avoid name clashes between different document instances.

If you'd like more information on any of these topics, here are a few links you should look at:

The W3C's XML Schema section, at http://www.w3.org/XML/Schema

The XML Schema Primer, at http://www.w3.org/TR/xmlschema−0/

XML Schema Structures, at http://www.w3.org/TR/xmlschema−1/

XML Schema Datatypes, at http://www.w3.org/TR/xmlschema−2/

Understanding XML Schema, at http://www.devshed.com/Server_Side/XML/XMLSchema

Till next time − happy XML−ing!

Note: All examples in this article have been tested on Linux/i586. Examples are illustrative only, and are not meant for a production environment. Melonfire provides no warranties or support for the source code described in this article. YMMV!