By Harish Kamath

Table of Contents

Revving Up

Back in the old days, when XML was still a dark and nebulous cloud on the horizon, the only way to verify the integrity of XML−encoded data was with a Document Type Definition (DTD). DTDs were incomprehensible beasts consisting of strange symbols and tangled acronyms, and it took a tremendous amount of patience (not to mention a fair amount of alcohol) to successfully write one that worked as it was supposed to.

Realizing that the arcane syntax used to construct DTDs was hindering rather than helping its efforts to make XML the de facto standard for data markup on the Web, the W3C came up with a kinder, gentler way of validating XML data. It was called XML Schema, and it offered developers all the capabilities of current DTDs while simultaneously adding a number of new capabilities designed to improve maintainability and extensibility.

As the name suggests, a "schema" is a blueprint for a specific class of XML document. It lays down rules for the types of elements and attributes allowed within an XML document, the types of values that accompany such elements, and the order and occurrence of these elements. It also addresses a number of issues which cannot be handled by DTDs: datatyping (including the ability to derive new datatypes from existing ones), inheritance, grouping, and database linkage.

Specific XML documents (referred to by the Working Group as "document instances") can be linked to a schema and validated against the rules contained within it. The XML Schema specification specifies the process by which document instances and schemas are linked together, and a number of tools are now available to perform this validation.

Now, if you've been paying attention to previous columns, you probably already know the basics of how schemas work (in case you don't, drop by http://www.devshed.com/Server_Side/XML/ and get yourself up to speed). In this series of articles, I'll be building on that basic knowledge to demonstrate some of the more advanced capabilities available to you via XML schemas, in the hope that it will assist you in fully exploiting the powers of this new tool. Keep reading!

Keeping It Simple

Let's begin with a quick refresher course in simple and complex element types. Consider the following XML document:

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

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

The XML Schema specification makes a basic distinction between "simple" and "complex" elements. Simple elements cannot contain other elements or possess additional attributes; complex elements can have additional attributes and serve as containers for other elements (which themselves may be either simple or complex).

Within a schema, these two element types are represented by the <xsd:simpleType> and <xsd:complexType> elements respectively.

The easiest method to represent simple elements in a schema is to use the <xsd:element> declaration with a built−in datatype − the following simple element

```
<name>Luke Skywalker</name>
```

would be represented in a schema by

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">;

<xsd:element name="name" type="xsd:string"/>

</xsd:schema>
```

When the datatype name is preceded by the "xsd:" prefix, it indicates a predefined datatype and not a new, user−defined type. The XML Schema specification lists about forty different built−in datatypes, including "string", "integer", "decimal", "float", "boolean", "time", "date", "dateTime" and "anyURI". However, in case these are too generic for you, it's also possible to derive your own custom datatype from the built−in ones, and then declare simple elements using this custom datatype.

A Complex Web

Let's now look at a complex element, as illustrated by the following XML snippet

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

As stated above, complex elements can contain other (simple or complex) elements, and may also possess additional attributes. Corresponding to this, complex element definitions within a schema can contain definitions for other (simple or complex) elements, definitions for element attributes (if any), and references to other element definitions within the schema.

When defining complex types, there are two ways in which the definition can be structured. The first involves declaring a complex type via the element, giving it a name, and then using this newly−minted type in a regular <xsd:element> declaration.

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

<!−− define a new datatype for a complex element −−>
<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>

<!−− create an element of this type −−>
<xsd:element name="character" type="starWarsEntity" />

</xsd:schema>
```

The second option involves combining the two steps above into a single step − the definition of the complex element is embedded within the <xsd:element> declaration itself.

Here's how the XML snippet above would be represented in a schema.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:element name="character">
<!-- this complex element definition has no name, and is
referred to as an "anonymous" element -->
<xsd:complexType>
<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>
</xsd:element>

</xsd:schema>
```

The advantage of creating a named type should be obvious − the new type, once defined, can be used in multiple places within the schema simply by referencing it by name.

Nesting Season

If a complex element contains child elements, these child element definitions appear nested within a <xsd:sequence> element. In the previous example, the elements nested within the "character" container element are all simple elements; however, it's also possible to have nested complex elements, as in the following XML document:

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

<gallery>

<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>
```

In this case, the element "character", which contains child elements of its own, is nested within the "gallery" element, which is itself a complex element containing many instances of "character". The corresponding schema definition would look like this:

```
<xsd:schema 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"/>
<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="starWarsEntity" maxOccurs="unbounded"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>

</xsd:schema>
```

In case you're wondering, the "maxOccurs" attribute is used to specify the maximum number of occurrences of the corresponding element (there's a corresponding "minOccurs" attribute to control the minimum number of occurrences). Both these attributes default to 1, unless they're explicitly assigned a value.

In this particular example, a value of "unbounded" for the "maxOccurs" attribute allows for an infinite number of "character" elements in the document instance.

I could also use the second technique discussed on the previous page to create a schema definition without using a named type. I'll leave this to you as an exercise, since it's usually better to name your types as you create them for greater re−use value.

Extending Yourself

One of the coolest things about schemas is that they allow you to extend previously defined datatypes to create new sub−types, in much the same way as OOP programmers extend existing classes. These sub−types will automatically inherit all the characteristics of the base type, but may also be further customized to specific requirements.

In order to better understand this, let's modify the example on the previous page to derive a new datatype from the base "starWarsEntity" type.

```
<xsd:schema 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"/>
<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>

<!−− extend base to derive a new sub−type −−>
<xsd:complexType name="Wookie">
<xsd:complexContent>
<xsd:extension base="starWarsEntity">
</xsd:extension>
</xsd:complexContent>
</xsd:complexType>

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

</xsd:schema>
```

In order to extend an existing datatype, you need to use the <xsd:extension> element, which goes hand−in−hand with a "base" attribute − this "base" attribute contains the name of the parent type (which must, obviously, exist).

What use is this, you ask? Not much at the moment...but look how easy it becomes to add species−specific attributes to the derived datatype:

```
<xsd:schema 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"/>
<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>

<!−− extend base to derive a new sub−type −−>
<xsd:complexType name="Wookie">
<xsd:complexContent>
<xsd:extension base="starWarsEntity">
<xsd:sequence>
<xsd:element name="battlecry"
type="xsd:string"/>
</xsd:sequence>
</xsd:extension>
</xsd:complexContent>
</xsd:complexType>

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

</xsd:schema>
```

So Wookies now have an additional characteristic − "battlecry".

Since it's so easy, let's derive a couple more:

```
<xsd:schema 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"/>
<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>

<!-- extend base to derive a new sub-type -->
<xsd:complexType name="Wookie">
<xsd:complexContent>
<xsd:extension base="starWarsEntity">
<xsd:sequence>
<xsd:element name="battlecry"
type="xsd:string"/>
</xsd:sequence>
</xsd:extension>
</xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="Human">
<xsd:complexContent>
<xsd:extension base="starWarsEntity">
<xsd:sequence>
<xsd:element name="gender"
type="xsd:string"/>
</xsd:sequence>
</xsd:extension>
</xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="Ewok">
<xsd:complexContent>
<xsd:extension base="starWarsEntity">
<xsd:sequence>
<xsd:element name="vehicle"
type="xsd:string"/>
<xsd:element name="society"
type="xsd:string"/>
</xsd:sequence>
</xsd:extension>
</xsd:complexContent>
</xsd:complexType>
```

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

</xsd:schema>
```

So Humans now include the additional characteristic "gender", while Ewoks include the characteristics "vehicle" and "society". Each of these derived datatypes thus carries its own special modifications, while simultaneously including all the characteristics of the base "starWarsEntity" datatype.

Now, how do I make my XML document use these derived datatypes? Pretty simple − just add the "type" attribute to each "character" element, so that the XML validator knows to look at the derived datatype rather than the base type. Here's how:

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

<gallery xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="starwars.xsd">

<character xsi:type="Human">
<name>Luke Skywalker</name>
<species>Human</species>
<language>Basic</language>
<home>Tatooine</home>
<gender>Male</gender>
</character>

<character xsi:type="Wookie">
<name>Chewbacca</name>
<species>Wookie</species>
<language>Shyriiwook</language>
<home>Kashyyyk</home>
<battlecry>AARGH!</battlecry>
</character>

<character xsi:type="Ewok">
<name>Chief Chirpa</name>
<species>Ewok</species>
<language>Ewok</language>
<home>Endor</home>
```

```
<vehicle>Glider</vehicle>
<society>Tribal</society>
</character>

</gallery>
```

Simple, huh?

Filing It All Away

In the examples on the previous pages, I have shown you how to extend an existing datatype to create new sub−types. As the number of type definitions in your schema goes on increasing, it becomes hard to manage them all in a single file. At some point, you're going to want to organize and classify these type definitions for easy maintenance.

The XML Schema specification addresses this requirement via the <xsd:include> element, which allows you to call an external XML Schema document and reference the data within it in the current one.

In order to better understand this, let's split the schema on the previous page into two separate files. I'll begin with the root element and base type definitions, which I'll place in the file "base−defs.xsd":

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

<!−− base definitions −−>

<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>

<xsd:element name="gallery">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="character"
type="starWarsEntity" maxOccurs="unbounded"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>
```

Next, I'll put all my derived types in the file "derived−defs.xsd".

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

<!−− include base types−−>
<xsd:include schemaLocation="base−defs.xsd"></xsd:include>
```

```
<!-- derived types -->
<xsd:complexType name="Human">
<xsd:complexContent>
<xsd:extension base="starWarsEntity">
<xsd:sequence>
<xsd:element name="gender"
type="xsd:string"/>
</xsd:sequence>
</xsd:extension>
</xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="Ewok">
<xsd:complexContent>
<xsd:extension base="starWarsEntity">
<xsd:sequence>
<xsd:element name="vehicle"
type="xsd:string"/>
<xsd:element name="society"
type="xsd:string"/>
</xsd:sequence>
</xsd:extension>
</xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="Human">
<xsd:complexContent>
<xsd:extension base="starWarsEntity">
<xsd:sequence>
<xsd:element name="gender"
type="xsd:string"/>
</xsd:sequence>
</xsd:extension>
</xsd:complexContent>
</xsd:complexType>

</xsd:schema>
```

Obviously, I need to link the derived types to the base type − a task easily accomplished via the <xsd:include> element, which includes a "schemaLocation" attribute specifying the location of the schema to be sourced into the current file.

Once the schemas are linked together, all I need to do is specify the location of the last link in the chain − here, "derived−defs.xsd" − in my XML file,

```
<?xml version="1.0" encoding="UTF-8"?>
<gallery xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xsi:noNamespaceSchemaLocation="derived-defs.xsd">
...
</gallery>
```

and all the required definitions will be automatically included and used when required by the XML validator.

And that's about it for the moment. In this article, I took a step into the deeper waters of advanced schema design, demonstrating how to build complex datatypes by combining and grouping simpler ones. I also showed you how to apply some basic OOP concepts − extensibility and inheritance − to schema design by extending existing type definitions to create new ones, and demonstrated how to separate definitions into files for greater maintainability.

In the next article, I'll be continuing this discussion, demonstrating how to derive new types by restricting (rather than extending) existing ones, create abstract definitions and redefine existing types. Make sure you come back for that...and, until then, go practise!

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!