Understanding XML Schema

By Harish Kamath

Table of Contents

Bad Medicine

If you're like most XML developers, you probably treat DTDs like bitter medicine − you have to use them, but you don't necessarily have to enjoy the process.

It's not hard to understand this attitude − the tangled syntactical constructs and difficult−to−remember symbols of a DTD are enough to give most developers a splitting headache (not to mention a lifelong aversion to the technology). Even more frustrating is the fact that when it comes to validating XML, DTDs are the only game in town − if you plan on using XML seriously, you need to know how to create and maintain a DTD, together with its associated !ELEMENTs and !ATTLISTs.

Or do you?

Taking note of the murderous feelings keeping XML developers up at night, the W3C has recently come up with a kinder, gentler way of validating XML data. It goes by the name XML Schema, offers all the capabilities of current DTDs, adds a bunch of powerful new features, and is a lot easier on the eye.

Sounds interesting? Keep reading!

The Next Generation

In case you're new to the whole XML scene, I'll first take a couple of minutes to explain where schemas fit in.

You probably already know that XML is a toolkit to describe data; it allows document authors to "mark up" textual data with descriptive tags, thereby adding value to the data and making it more useful to the applications that process it. XML also offers document authors a way to enforce certain grammatical rules on an XML document via DTDs, or Document Type Definitions, which specify a structure and format for an XML document to be considered valid.

However, DTDs suffer from two glaring problems. First, they're not terribly user−friendly − creating a DTD requires a working knowledge of a bunch of arcane commands and constructs, and the end result is usually hard to read and understand. Second, although DTDs are designed to impose a structure on XML data, they are themselves not written in XML.

In order to address these two issues, and also to lay the groundwork for the next generation of XML data validation, the W3C created the XML Schema Working Group to create a new schema language for XML data validation.

A "schema", as the name suggests, 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 the current generation of 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.

Unlike DTDs, XML Schemas are constructed using standard XML syntactical rules, immediately making them easier to read, understand and maintain.

An Evening At The Moulin Rouge

Perhaps the best way to illustrate the difference between a DTD and a schema is with an example − consider the following XML document,

```
<?xml version="1.0"?>
<movie>
<title>Moulin Rouge</title>
<director>Baz Luhrmann</director>
</movie>
```

and its associated DTD.

```
<!ELEMENT movie (title, director)>
<!ELEMENT director (#PCDATA)>
<!ELEMENT title (#PCDATA)>
```

If I were to convert this DTD into an XML Schema, here's what it would look like:

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

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

<xsd:element name="movie">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="title" type="xsd:string"/>
<xsd:element name="director" type="xsd:string"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>

</xsd:schema>
```

Let's dissect this a little:

A schema is a well−formed XML document. It even begins with the standard XML document prolog, which specifies the XML version and document character set.

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

The outermost element in a schema is the <xsd:schema> element, which includes a namespace declaration to associate the document with the XML Schema namespace.

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

This namespace declaration tells the parser that elements specific to the schema will be declared with the "xsd:" prefix; this is necessary both to avoid clashes with user−defined element names, and to allow the parser to distinguish between schema instructions and non−schema elements.

The outermost <xsd:schema> element is a container; it encloses the element and attribute definitions which constitute the validation rules of the schema. Don't worry about these for the moment, I'll be discussing them shortly.

Once all the validation rules have ended, the document is closed with a

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

Now, the schema itself is only one part of the puzzle − the other part is the document instance which will be validated against the schema. Typically, the two are linked together by adding a special attribute indicating the location of the schema to the XML document instance. Using the document instance above, we have

```
<movie xmlns:xsi="http://www.w3.org/2001/XMLSchema−instance"
xsi:noNamespaceSchemaLocation="movie.xsd">
<title>Moulin Rouge</title>
<director>Baz Luhrmann</director>
</movie>
```

The "xsi:noNamespaceSchemaLocation" attribute holds the location of the schema against which this document is to be validated. This attribute is used to validate specific document instances against a schema (which is why it comes from the XML Schema Instance namespace.)

An alternative to this is the "xsi:schemaLocation" attribute, which specifies a schema location with a corresponding namespace location (a detailed explanation of this is beyond the scope of this tutorial, but take a look at the XML Schema specification for examples and more information.)

```
<movie xmlns:xsi="http://www.w3.org/2001/XMLSchema−instance"
xsi:schemaLocation="http://www.mydomain.com/movie/movie−namespace
http://www.mydomain.com/movie/movie.xsd">
<title>Moulin Rouge</title>
<director>Baz Luhrmann</director>
</movie>
```

In addition to these two methods, different XML processors may allow you to specify schema locations in different ways (for example, on the command line.)

Finally, with your schema and document instance linked together, it's time to take them for a test drive. A good processor to use is XSV, an open−source XML Schema Validator from the University of Edinburgh and the W3C − you can download it from http://www.w3.org/XML/Schema#Tools. Alternatively, take a look at XMLSpy, a powerful and very cool XML editor available for download at http://www.xmlspy.com/, or visit the W3C's software page at http://www.w3.org/XML/Schema#Tools

Here's what XSV has to say when I run it on my "movie.xml" document instance above:

```
$ xsv movie.xml
<?xml version='1.0'?>
<xsv docElt='{None}movie' instanceAssessed='true'
instanceErrors='0'
rootType='[Anonymous]' schemaErrors='0' schemaLocs='None ->
movie.xsd'
target='/usr/local/apache/htdocs/movie.xml'
validation='strict'
version='XSV 1.202/1.105 of 2001/08/30 16:12:04'
xmlns='http://www.w3.org/2000/05/xsv'>
<schemaDocAttempt URI='/usr/local/apache/htdocs/movie.xsd'
outcome='success' source='schemaLoc'/>
</xsv>
```

Simple Simon

With the broad overview out of the way, let's now focus on the different elements (pun definitely intended!) that make up a schema.

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.

Simple elements can be represented in two ways. The first (and simplest) method is to use the <xsd:element> declaration with a built−in datatype − the following simple element

```
<title>Moulin Rouge</title>
```

would be represented in a schema by

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

<xsd:element name="title" 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.

Consider the following schema definition, which is equivalent to the one above:

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

<!-- define a new datatype -->
<xsd:simpleType name="simpleDType">
<xsd:restriction base="xsd:string"></xsd:restriction>
</xsd:simpleType>

<!-- declare an element of this type -->
<xsd:element name="title" type="simpleDType"/>
```

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

Typically, this second method is used only when a schema author needs to restrict the values of a particular simple element over and above the constraints inherent in a specific datatype. Here's a more constructive definition, which restricts the values of the "rating" element to an integer between 1 and 10.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!-- define a new datatype -->
<xsd:simpleType name="simpleDType">
<xsd:restriction base="xsd:integer">
<!-- restrict integer values between 1 and 10 -->
<xsd:minInclusive value="1"/>
<xsd:maxInclusive value="10"/>
</xsd:restriction>
</xsd:simpleType>


<!-- declare an element of this type -->
<xsd:element name="rating" type="simpleDType"/>
</xsd:schema>
```

I'll be discussing the derivation of new datatypes in more detail a little further down, so don't worry too much if the syntax seems a little unfamiliar. The important thing to note here is the two different options open to you while defining a simple element.

A Complex Web

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

```
<movie>
<title>Moulin Rouge</title>
<director>Baz Luhrmann</director>
</movie>
```

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 (more on this later).

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

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

<!-- define a new datatype for a complex element -->
<xsd:complexType name="complexDType">
<xsd:sequence>
<xsd:element name="title" type="xsd:string"/>
<xsd:element name="director" type="xsd:string"/>
</xsd:sequence>
</xsd:complexType>

<!-- create an element of this type -->
<xsd:element name="movie" type="complexDType" />

</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="movie">
<!-- this complex element definition has no name, and is
referred to as
```

```
an "anonymous" element -->
<xsd:complexType>
<xsd:sequence>
<xsd:element name="title" type="xsd:string"/>
<xsd:element name="director" 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. The following example illustrates this:

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

<!-- define a new datatype -->
<xsd:simpleType name="simpleDType">
<xsd:restriction base="xsd:integer">
<!-- restrict integer values between 1 and 10 -->
<xsd:minInclusive value="1"/>
<xsd:maxInclusive value="10"/>
</xsd:restriction>
</xsd:simpleType>

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

<!-- simple elements using built-in types -->
<xsd:element name="title" type="xsd:string"/>
<xsd:element name="director" type="xsd:string"/>

<!-- simple element using new derived type -->
<xsd:element name="rating" type="simpleDType"/>

</xsd:sequence>
</xsd:complexType>
</xsd:element>

</xsd:schema>
```

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 "movie" 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"?>
<movie>
<title>Moulin Rouge</title>
<cast>
<person>Nicole Kidman</person>
<person>Ewan McGregor</person>
</cast>
</movie>
```

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

```
<?xml version"1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="movie">
<xsd:complexType>
<xsd:sequence>
<!-- simple element definition -->
<xsd:element name="title" type="xsd:string"/>

<!-- complex element definition -->
<xsd:element name="cast">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="person" type="xsd:string"
maxOccurs="unbounded"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>

</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. And just incidentally, you can't achieve this level of precision with a DTD.

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

It's also possible to make certain elements optional. In order to illustrate this, let's add a couple of new elements to the schema definition above − a "release_date" element of type "date", and a "rating" element of type "integer". And let's also make the "release_date" element optional, through judicious use of the "minOccurs" and "maxOccurs" attributes.

```
<?xml version"1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="movie">
<xsd:complexType>
<xsd:sequence>
<!−− simple element definition −−>
<xsd:element name="title" type="xsd:string"/>

<!−− complex element definition −−>
<xsd:element name="cast">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="person" type="xsd:string"
maxOccurs="unbounded"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<!−− simple element definition −−>
<xsd:element name="release_date" type="xsd:date" maxOccurs="1"
minOccurs="0"/>

<!−− simple element definition −−>
<xsd:element name="rating" type="xsd:integer"/>

</xsd:sequence>
</xsd:complexType>
</xsd:element>

</xsd:schema>
```

Battle Of The Sexes

So that takes care of elements − but what about attributes?

Attribute definitions typically follow element definitions, and are declared with the <xsd:attribute> element.

```
<xsd:attribute name="genre" type="xsd:string" />
```

An optional "use" attribute can be used to specify whether or not the attribute is optional.

```
<!-- this attribute is optional -->
<xsd:attribute name="genre" type="xsd:string" use="optional"
/>

<!-- this attribute is required-->
<xsd:attribute name="id" type="xsd:integer" use="required" />

<!-- this attribute is not allowed -->
<xsd:attribute name="address" type="xsd:string"
use="prohibited" />
```

In order to see this in action, let's suppose I altered my document instance to include an attribute:

```
<?xml version="1.0"?>
<movie genre="romance">
<title>Moulin Rouge</title>
<cast>
<person>Nicole Kidman</person>
<person>Ewan McGregor</person>
</cast>
<rating>5</rating>
</movie>
```

Here's what my schema would look like:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="movie">
<xsd:complexType>
<xsd:sequence>
<!-- simple element definition -->
```

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

<!-- complex element definition -->
<xsd:element name="cast">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="person" type="xsd:string"
maxOccurs="unbounded" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<!-- simple element definition -->
<xsd:element name="release_date" type="xsd:date" maxOccurs="1"
minOccurs="0"/>

<!-- simple element definition -->
<xsd:element name="rating" type="xsd:integer"/>

</xsd:sequence>

<!-- attribute definition -->
<xsd:attribute name="genre" type="xsd:string" />

</xsd:complexType>
</xsd:element>

</xsd:schema>
```

How about adding a twist? Let's suppose every "person" element needs an additional "sex" attribute, like this:

```
<?xml version="1.0"?>
<movie genre="romance">
<title>Moulin Rouge</title>
<cast>
<person sex="female">Nicole Kidman</person>
<person sex="male">Ewan McGregor</person>
</cast>
<rating>5</rating>
</movie>
```

This apparently minor change has quite a significant impact on the schema definition, because the "person" element, previously defined as a simple type, must now be defined as a complex type. Since the "person" element does not have any nested child elements, we can pass up the <xsd:sequence> instruction within the element definition in favour of the <xsd:simpleContent> element and include an attribute definition within this element.

Here's the revised schema definition:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="movie">
<xsd:complexType>
<xsd:sequence>
<!-- simple element definition - title -->
<xsd:element name="title" type="xsd:string"/>

<!-- complex element definition - cast -->
<xsd:element name="cast">
<xsd:complexType>
<xsd:sequence>

<!-- since person has an attribute, it must be defined as a
complex
element -->
<xsd:element name="person" maxOccurs="unbounded">
<xsd:complexType>
<xsd:simpleContent>
<xsd:restriction base="xsd:string">
<!-- attribute definition - sex -->
<xsd:attribute name="sex" type="xsd:string" />
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>
</xsd:element>

</xsd:sequence>
</xsd:complexType>
</xsd:element>

<!-- simple element definition - release date -->
<xsd:element name="release_date" type="xsd:date" maxOccurs="1"
minOccurs="0"/>

<!-- simple element definition - rating -->
<xsd:element name="rating" type="xsd:integer"/>

</xsd:sequence>

<!-- attribute definition for movie genre -->
<xsd:attribute name="genre" type="xsd:string" />

</xsd:complexType>
</xsd:element>
```

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

Dealing With The Ref

Now, while it's certainly possible to build a schema in the manner just described, with the schema definition following the structure of the document instance, such a schema can prove difficult to maintain and read with long and complex documents. Consequently, the XML Schema specification allows for an alternative method of schema construction, whereby elements and attributes can be defined separately and then referenced wherever required.

An example might help to make this clearer − consider the following revision of the previous example:

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

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

<xsd:element name="release_date" type="xsd:date"/>

<xsd:element name="rating" type="xsd:integer"/>

<xsd:attribute name="genre" type="xsd:string"/>

<xsd:attribute name="sex" type="xsd:string"/>

<xsd:element name="person">
<xsd:complexType>
<xsd:simpleContent>
<xsd:restriction base="xsd:string">
<xsd:attribute ref="sex" />
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>
</xsd:element>


<xsd:element name="cast">
<xsd:complexType>
<xsd:sequence>
<xsd:element ref="person" maxOccurs="unbounded"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>


<!-- it all comes together here -->
<xsd:element name="movie">
<xsd:complexType>
<xsd:sequence>
```

```
<xsd:element ref="title"/>
<xsd:element ref="cast"/>
<xsd:element ref="release_date" maxOccurs="1" minOccurs="0"/>
<xsd:element ref="rating"/>
</xsd:sequence>
<xsd:attribute ref="genre"/>
</xsd:complexType>
</xsd:element>

</xsd:schema>
```

In this form of construction, each element and attribute is defined separately and can be referenced by any other element definition via the "ref" attribute. This type of schema definition is much easier to read than the previous one.

Different Flavours

Now, if your validation rules are not very stringent, you don't usually need to derive new datatypes; the built−in ones suffice for most requirements. If, on the other hand, you need to restrict your data to specific ranges or values, you will find it necessary to create new datatypes and use them within your schema definition. Let's move on to a closer examination of how this is accomplished.

The XML Schema specification allows for the derivation of new datatypes from the built−in types. Typically, these new derivations are created by placing a restriction on the allowed range of values for a built−in datatype, via the <xsd:restriction> element.

You've already seen this element in some of the previous examples − in fact, here's a quick snippet to jog your memory:

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

<!−− other definitions − snip! −−>

<!−− create a new datatype −−>
<xsd:simpleType name="simpleDType">
<xsd:restriction base="xsd:integer">
<xsd:minInclusive value="1"/>
<xsd:maxInclusive value="10"/>
</xsd:restriction>
</xsd:simpleType>

<!−− declare an element of this type −−>
<xsd:element name="rating" type="simpleDType"/>

</xsd:schema>
```

In this case, a new datatype named "simpleDType" has been created using the built−in "integer" datatype as a base. The <xsd:minInclusive> and <xsd:maxInclusive> elements are used to specify an allowable range of values for this new datatype; these restrictions are referred to in schema jargon as "facets".

It's also possible to create restrictions on the basis of patterns or regular expressions − consider the following element definition, which defines a datatype for temperature values using the <xsd:pattern> facet. This derived datatype allows for a value containing a maximum of three digits, prefixed by an optional minus sign.

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

<!−− other definitions − snip! −−>
```

```
<!-- create a new datatype -->
<xsd:simpleType name="temperatureDType">
<xsd:restriction base="xsd:string">
<xsd:pattern value="-?[0-9]{1,3}"/>
</xsd:restriction>
</xsd:simpleType>

<!-- declare an element of this type -->
<xsd:element name="temp" type="temperatureDType"/>

</xsd:schema>
```

You can restrict an element to certain values with the <xsd:enumeration> facet − the following example creates a datatype which, when applied to an element declaration, restricts it to one of four values:

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

<!-- other definitions - snip! -->

<!-- create a new datatype -->
<xsd:simpleType name="flavourDType">
<xsd:restriction base="xsd:string">
<xsd:enumeration value="vanilla"/>
<xsd:enumeration value="chocolate"/>
<xsd:enumeration value="strawberry"/>
<xsd:enumeration value="peach"/>
</xsd:restriction>
</xsd:simpleType>

<!-- declare an element of this type -->
<xsd:element name="flavour" type="flavourDType"/>

</xsd:schema>
```

Why stop at elements? You can use the datatype above to restrict attributes to specific values as well. Consider the following derived datatype, which restricts element or attribute values to specific genres of film:

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

<!-- other definitions - snip! -->

<!-- create a new datatype -->
<xsd:simpleType name="genreDType">
```

```
<xsd:restriction base="xsd:NMTOKEN">
<xsd:enumeration value="romance"/>
<xsd:enumeration value="comedy"/>
<xsd:enumeration value="drama"/>
<xsd:enumeration value="action"/>
<xsd:enumeration value="horror"/>
</xsd:restriction>
</xsd:simpleType>

<!-- declare an attribute of this type -->
<xsd:attribute name="genre" type="genreDType"/>

</xsd:schema>
```

There are an infinite number of creative things you can do with power like this − for more examples and information on the numerous facets available for the different datatypes, take a look at http://www.w3.org/TR/xmlschema−2/

When In Rome...

Finally, comments. Since a schema definition is a well−formed XML document, you can include as many comments within it as you like, by enclosing each within the standard comment indicators. In addition to this, the XML Schema specification also allows schema authors to include human−readable comments in the <xsd:annotation> element. The following example illustrates this:

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

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

<xsd:annotation>
<xsd:documentation>Hail, fellow Romans!</xsd:documentation>
</xsd:annotation>


<xsd:simpleType name="simpleDType">

<xsd:annotation>
<xsd:documentation>This datatype restricts values to integers
between 1
and 10, both inclusive</xsd:documentation>
</xsd:annotation>

<xsd:restriction base="xsd:integer">
<xsd:minInclusive value="1"/>
<xsd:maxInclusive value="10"/>
</xsd:restriction>
</xsd:simpleType>

</xsd:schema>
```

Test Drive

Now that you've got the basics down, let's look at a couple of composite examples just to put everything in perspective. Consider the following two document instances, and then see if you can write appropriate schema definitions for each:

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

<!-- id is a required attribute -->
<recipe id="3450">

<name>Chicken Tikka</name>
<author>Mr. Cluck</author>
<date>1999-06-08</date>

<ingredients>
<!-- quantity is a required attribute, units is optional -->
<item quantity="2">Boneless chicken breasts</item>
<item quantity="2">Chopped onions</item>
<item quantity="1" units="tsp">Ginger</item>
<item quantity="1" units="tsp">Garlic</item>
<item quantity="1" units="tsp">Red chili powder</item>
<item quantity="1" units="tsp">Coriander seeds</item>
<item quantity="2" units="tbsp">Lime juice</item>
<item quantity="2" units="tbsp">Butter</item>
</ingredients>

<process>
<step>Cut chicken into cubes, wash and apply lime juice and
salt</step>
<step>Add ginger, garlic, chili, coriander and lime juice in a
separate
bowl</step>
<step>Mix well, and add chicken to marinate for 3-4
hours</step>
<step>Place chicken pieces on skewers and barbeque</step>
<step>Remove, apply butter, and barbeque again until meat is
tender</step>
<step>Garnish with lemon and chopped onions</step>
</process>

</recipe>
```

Here's the second one:

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

<!-- id is a required attribute -->
<city id="52320">
<name>Boston</name>
<temperature>
<!-- units is a required attribute, restricted to values
"celsius" and
"fahrenheit" -->
<high units="celsius">23</high>
<low units="celsius">5</low>
</temperature>
<!-- forecast may be any one of "rain", "sun", "snow" or "fog"
-->
<forecast>snow</forecast>
</city>

<city id="9010">
<name>New York</name>
<temperature>
<high units="celsius">11</high>
<low units="celsius">-5</low>
</temperature>
<forecast>snow</forecast>
</city>

<city id="8239">
<name>London</name>
<temperature>
<high units="celsius">27</high>
<low units="celsius">12</low>
</temperature>
<forecast>sun</forecast>
</city>

</weather>
```

Here's a schema definition for the first one:

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

<xsd:annotation>
<xsd:documentation>This version uses references and anonymous
elements</xsd:documentation>
</xsd:annotation>
```

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

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

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

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

<xsd:attribute name="id" type="xsd:integer" />

<xsd:attribute name="units" type="xsd:string" />

<xsd:attribute name="quantity" type="xsd:integer" />

<xsd:element name="process">
<xsd:complexType>
<xsd:sequence>
<xsd:element ref="step" maxOccurs="unbounded" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:element name="item">
<xsd:complexType>
<xsd:simpleContent>
<xsd:restriction base="xsd:string">
<xsd:attribute ref="quantity" use="required" />
<xsd:attribute ref="units" use="optional" />
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>
</xsd:element>

<xsd:element name="ingredients">
<xsd:complexType>
<xsd:sequence>
<xsd:element ref="item" maxOccurs="unbounded" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:element name="recipe">
<xsd:complexType>
<xsd:sequence>
<xsd:element ref="name" />
<xsd:element ref="author" />
<xsd:element ref="date" />
<xsd:element ref="ingredients" />
```

```
<xsd:element ref="process" />
</xsd:sequence>
<xsd:attribute ref="id" use="required" />
</xsd:complexType>
</xsd:element>

</xsd:schema>
```

This version first defines various elements and then references those definitions to construct a schema. In case this doesn't work for you, you can derive and use named datatypes instead of references − here's an alternative version of the schema above:

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

<xsd:annotation>
<xsd:documentation>This version uses named
datatypes</xsd:documentation>
</xsd:annotation>

<xsd:complexType name="itemDType">
<xsd:simpleContent>
<xsd:restriction base="xsd:string">
<xsd:attribute name="quantity" type="xsd:integer"
use="required" />
<xsd:attribute name="units" type="xsd:string" use="optional"
/>
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>

<xsd:complexType name="ingredientsDType">
<xsd:sequence>
<xsd:element name="item" type="itemDType"
maxOccurs="unbounded" />
</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="processDType">
<xsd:sequence>
<xsd:element name="step" type="xsd:string"
maxOccurs="unbounded" />
</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="recipeDType">
<xsd:sequence>
```

```
<xsd:element name="name" type="xsd:string" />
<xsd:element name="author" type="xsd:string" />
<xsd:element name="date" type="xsd:date" />
<xsd:element name="ingredients" type="ingredientsDType" />
<xsd:element name="process" type="processDType" />
</xsd:sequence>
<xsd:attribute name="id" type="xsd:integer" use="required" />
</xsd:complexType>

<!-- all type definitions done, now simply create an element
of type
"recipe" -->
<xsd:element name="recipe" type="recipeDType" />

</xsd:schema>
```

In a similar manner, you can create a schema for the weather forecast example, using either references

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

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

<xsd:attribute name="id" type="xsd:integer"/>

<xsd:attribute name="units">
<xsd:simpleType>
<xsd:restriction base="xsd:string">
<xsd:enumeration value="fahrenheit"/>
<xsd:enumeration value="celsius"/>
</xsd:restriction>
</xsd:simpleType>
</xsd:attribute>

<xsd:element name="high">
<xsd:complexType>
<xsd:simpleContent>
<xsd:restriction base="xsd:integer">
<xsd:maxInclusive value="150"/>
<xsd:minInclusive value="-150"/>
<xsd:attribute ref="units" use="required"/>
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>
</xsd:element>

<xsd:element name="low">
```

```
<xsd:complexType>
<xsd:simpleContent>
<xsd:restriction base="xsd:integer">
<xsd:maxInclusive value="150"/>
<xsd:minInclusive value="−150"/>
<xsd:attribute ref="units" use="required"/>
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>
</xsd:element>

<xsd:element name="forecast">
<xsd:simpleType>
<xsd:restriction base="xsd:string">
<xsd:enumeration value="fog"/>
<xsd:enumeration value="rain"/>
<xsd:enumeration value="sun"/>
<xsd:enumeration value="snow"/>
</xsd:restriction>
</xsd:simpleType>
</xsd:element>

<xsd:element name="temperature">
<xsd:complexType>
<xsd:sequence>
<xsd:element ref="high"/>
<xsd:element ref="low"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:element name="city">
<xsd:complexType>
<xsd:sequence>
<xsd:element ref="name"/>
<xsd:element ref="temperature"/>
<xsd:element ref="forecast"/>
</xsd:sequence>
<xsd:attribute ref="id" use="required"/>
</xsd:complexType>
</xsd:element>

<xsd:element name="weather">
<xsd:complexType>
<xsd:sequence>
<xsd:element ref="city" maxOccurs="unbounded"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
```

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

or derived types.

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

<xsd:simpleType name="nameDType">
<xsd:restriction base="xsd:string"/>
</xsd:simpleType>

<xsd:simpleType name="forecastDType">
<xsd:restriction base="xsd:string">
<xsd:enumeration value="fog"/>
<xsd:enumeration value="rain"/>
<xsd:enumeration value="sun"/>
<xsd:enumeration value="snow"/>
</xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="unitsDType">
<xsd:restriction base="xsd:string">
<xsd:enumeration value="celsius"/>
<xsd:enumeration value="fahrenheit"/>
</xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="temperatureDType">
<xsd:simpleContent>
<xsd:restriction base="xsd:integer">
<xsd:maxInclusive value="150"/>
<xsd:minInclusive value="-150"/>
<xsd:attribute name="units" type="unitsDType" use="required"/>
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>

<xsd:complexType name="temperatureWrapperDType">
<xsd:sequence>
<xsd:element name="high" type="temperatureDType"/>
<xsd:element name="low" type="temperatureDType"/>
</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="cityDType">
<xsd:sequence>
```

```
<xsd:element name="name" type="nameDType"/>
<xsd:element name="temperature"
type="temperatureWrapperDType"/>
<xsd:element name="forecast" type="forecastDType"
minOccurs="0"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:integer" use="required"/>
</xsd:complexType>

<xsd:complexType name="weatherDType">
<xsd:sequence>
<xsd:element name="city" type="cityDType"
maxOccurs="unbounded"/>
</xsd:sequence>
</xsd:complexType>

<!-- now simply declare an element of type "weather" -->
<xsd:element name="weather" type="weatherDType"/>

</xsd:schema>
```

Of these two approaches, I've always found the derived types approach to be a bit more flexible, not to mention more clearly−structured and logical − although it does take a bit of getting used to.

The Next Step

And that's just about it for the moment. While you now know enough about the basics of XML Schema to begin using it in your XML development activities, you should be aware that the topics covered today are merely the tip of the iceberg. The XML Schema specification provides for a number of other powerful capabilities (including inheritance and grouping) and you should spend some time reading the official specification if you intend to get serious about the subject.

If you're interested, drop me a line and tell me if you'd like to read about these advanced capabilities in another article − and while you're at it, take a look at the following links as well:

The official XML Schema specification, at http://www.w3.org/XML/Schema#dev

XML.com's XML Schema section, at http://www.xml.com/pub/a/2000/11/29/schemas/part1.html

XML Schema School, at http://www.w3schools.com/schema/default.asp

Zvon's XML Schema reference, at http://zvon.org/xxl/xmlSchemaReference/Output/index.html

Until next time...stay healthy!

Note: All examples in this article have been tested with xsv. Examples are illustrative only, and are not meant for a production environment. YMMV!