XML – Schema en dos patadas

Unai Estébanez
unai@unainet.net

Motivación

Estos apuntes sobre Schema surgen de mi apremiante necesidad de validar una serie de documentos XML.

Tuve que programar un exportador de datos que transformaba datos de una base de datos a XML, estos XML debían mantenerse con un formato dado para que siguieran siendo compatibles con los que utilizaba una aplación ya existente.

El mayor problema con el que me encontré una vez finalizado el exportador fue el validar estos documentos (que no tenían en su diseño original ni un DTD ni un Schema). Encontré una solución interesante en los XML-Schemas, ahora bien, al comenzar a mirar por Internet sentí cierta frustración porque no me gustaban las explicaciones de la mayoría de los artículos, o bien eran muy "serias", rodeadas de una parafernalia indescifrable o bien eran demasiado simplones y repetían una y otra vez el mismo ejemplo estúpido.

Es por esto que he realizado estos apuntes con los siguientes objetivos:

- Explicar de forma sencilla que es Schema
- Dar un repaso breve a la elaboración de un Schema
- Crear un ejemplo para aplicar lo visto

Audiencia

Este documento está dirigido a alguien que sabe lo que es un XML y que tiene un cierto bagaje en informática.

El perfil ideal es el de un programador que trabaja habitualmente con XML y nunca le han formado en este área o bien alguien que comienza a trabajar con XML por primera vez.

¿Que es XML-Schema?

Un esquema XML no es más que un documento XML que a su vez describe la estructura de un otro documento XML que es el que se quiere validar.

Los motivos para validar un XML son variados, por ejemplo:

- Asegurarnos de que la transmisión de datos entre dos puntos ha sido correcta en lineas con alta tasa de fallos.
- Asegurarnos de que una de las dos partes que se comunican mediante XML no tiene bugs y transmite documentos erróneos que pueden hacer fallar a la otra parte
- Asegurarnos de que las dos partes transmiten la información en el mismo formato, como por ejemplo las fechas.

En definitiva se trata de poder, dado un xml, decir si se sigue un acuerdo preestablecido(el esquema) o no.

Una vez escrito el esquema que describe el xml que queremos tratar bastará con enlazar cada xml que queramos validar al esquema y de esta forma se podrá detectar si este está o no bien formado.

Enlazar un XML con un esquema

Fijémonos en el siguiente ejemplo:

```
<?xml version=”1.0” encoding=”UTF-8” standalone= “no”?>
<!DOCTYPE pedido >
<pedido xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”
xsi:noNamespaceSchemaLocation=”u4.xsd” codigo=”A1234567” observaciones=”urgente”>
```

Ahora lo observaremos linea a linea:

`<?xml version=”1.0” encoding=”UTF-8” standalone= “no”?>`

Es la declaración normal de un XML, marcamos que necesita otro documento con standalone = “no”.

`<!DOCTYPE pedido >`

Tipo de documento en el que sólo incluimos el nombre del ejemplar

`<pedido xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance” xsi:noNamespaceSchemaLocation=”u4.xsd” codigo=”A1234567” observaciones=”urgente”>`

El ejemplar, aquí incluimos un namespace con prefijo xsi con las reglas de Schema y una vez que lo tenemos cargado lo usamos con `xsi:noNamespaceSchemaLocation=”u4.xsd”` para cargar nuestro esquema al que, en este caso, no le ponemos prefijo.
Finalmente agragmos los atributos que tuviera el ejemplar como en cualquier XML.

Cabecera del xsd

Fijémonos en el siguiente ejemplo:

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

Nuevamente lo observaremos linea a linea:
`<?xml version="1.0" encoding="UTF-8">` Es la declaración normal de un XML.

`<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">`
Tcreamos un namespace con prefijo xs. http://www.w3.org/2001/XMLSchema es un URI no URL, es decir, no tiene porque apuntar a nada, pero por convenio lo hacemos así
Además el prefijo usado en este caso es xs, también por convenio, aunque otros autores usan xsd

Conceptos básicos

El esquema no es más que un xml como otro cualquiera y por lo tanto debe estar formado como tales. Tags, abiertos y cerrados, comentarios, elementos, atributos, etc.

El nodo padre del esquema se llamará **<schema>.**

Una vez abierto este tag <schema> dentro se especificará que es lo que debe contener todo documento que siga este esquema.

```
<schema>
"declaración de elementos del documento" ← Esto es lo que vamos a tener que "picar"
</schema>
```

En el esquema todo lo que vamos a hacer es definir los **elementos** que contiene el xml que sigue el esquema.

Los **elementos** serán **simples** o **complejos.**

Elementos simples

Los elementos simples son los que solamente contienen texto, no contiene otros elementos hijos ni tampoco atributos.

Eso si, el texto puede ser cualquier tipo de datos definido en schema (Integer,string, fecha-hora,booleano,etc) o cualquier tipo definido por nosotros, es posible incluso añadir restricciones (llamadas facetas) u obligar a que se cumpla algún patrón mediante expresiones regulares.

Ejemplo:

```
<elemento> solo contengo texto </elemento>
```

Para definir un elemento usaremos un tag llamado, como no, "element", por ejemplo:

```
<xs:element name="apellidos" type="xs:string"/>
<xs:element name="edad" type="xs:integer"/>
<xs:element name="nacimiento" type="xs:date"/>
```

Hemos visto en el ejemplo como definir tres elementos simples y hemos comentado que se pueden establecer restricciones sobre los datos, veamos como.

Restricciones sobre los datos

Podemos incluir los siguientes tipos de restricciones:

- Sobre los valores
- Sobre un conjunto de valores
- Sobre los espacios en blanco
- Sobre la longitud
- Sobre patrones

Sobre los valores

Veamos como limitar la edad de un individuo para que valga de 0 a 100 años.

```
<xs:element name="edad">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="100"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Sobre un conjunto de valores

Podemos obligar a que un valor se establezca dentro de unos límites:

```
<xs:element name="color">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="Rojo"/>
      <xs:enumeration value="Verde"/>
      <xs:enumeration value="Azul"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Si reescribimos lo anterior de la siguiente forma, podremos reaprovechar la definición de color RGB:

```
<xs:element name="color" type="colorRGB"/>

<xs:simpleType name="colorRGB">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Rojo"/>
    <xs:enumeration value="Verde"/>
    <xs:enumeration value="Azul"/>
  </xs:restriction>
</xs:simpleType>
```

Sobre los espacios en blanco

Podemos jugar con los espacios en blanco para **mantenerlos intocables,restringirlos o eliminarlos.**

El siguiente ejemplo los mantiene intocables, esto significa que retornos de carro, tabuladores y espacios en blanco no serán alterados dentro del campo de tipo dirección:

```
<xs:element name="direccion">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:whiteSpace value="preserve"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Este otro ejemplo, haremos que se sustituyan todos los tabuladores,retornos de carro(CR) y fines de linea (LF) por espacios en blanco:

```
<xs:element name="direccion">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:whiteSpace value="replace"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Colapsando es posible hacer que múltiples espacios en blancos, retornos de carro, tabuladores y demás se colapsen a un solo espacio en blanco.

```
<xs:element name="direccion">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:whiteSpace value="collapse"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Sobre la longitud

Podemos especificar un mínimo o máximo de longitud, veamos dos ejemplos:

```
<xs:element name="apellido">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:length value="8"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

```
<xs:element name="apellido">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:minLength value="8"/>
      <xs:maxLength value="255"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Vemos como es posible jugar con las anchuras de estos campos fijándolos a un tamaño concreto o indicando márgenes.

Existen más restricciones sobre tipos de datos en concreto como los numéricos, en cualquier manual de schema podréis encontrarlos.

Sobre patrones

Es posible también controlar mediante expresiones regulares los contenidos de un campo, esta es una forma muy potente de asegurarnos el contenido deseado en cada campo:

```
<xs:element name="mayusculas">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="([A-Z])*"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

En este ejemplo vemos como obligamos a que el elemento "mayusculas" tenga un contenido en mayúsculas mediante una expresión regular.

Atributos

Hemos comentado que los tipos simples no pueden tener, entre otras cosas, atributos. Ahora bien, los atributos, que serán utilizados en los tipos complejos, son declarados en si mismos como un tipo simple.

La forma general de declarar un atributo es como sigue:

```
<xs:attribute name="xxx" type="yyy"/>
```

Donde xxx es el nombre del atributo y yyy será un tipo de datos de los nativos de schema o de los definidos por nosotros, veamos como definir un atributo "Alias" para usarlo dentro de un elemento "nombre":

```
<xs:attribute name="alias" type="xs:string"/>
<!-- Ahora podemos usarlo dentro de un elemento nombre -->
<nombre alias="El cadenas">Iker</nombre>
```

Elementos complejos

Un elemento complejo es aquel que contenga otro elemento y/o atributos.

Hay cuatro tipos de elementos complejos:

- Elementos que contienen solo otros elementos
- Elementos vacios
- Elementos que contienen solo texto
- Elementos que contienen otros elementos y texto

En estas cuatro categorías además se pueden incluir atributos en los elementos.

Elementos que contienen solo otros elementos

Veamos como definir un tipo complejo (elemento que contiene solo otros elementos):

```
<xs:element name="alumno" type="persona"/>

<xs:complexType name="persona">
  <xs:sequence>
    <xs:element name="apellido" type="xs:string"/>
    <xs:element name="nombre" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

Esto se traduce en poder utilizar elementos como:

```
<persona>
      <apellido>Estébanez</apellido>
      <nombre>Unai</nombre>
</persona>
```

Hemos usado en esta definición un **indicador** secuencia (sequence) que explicaremos más adelante.

Se puede apreciar en el ejemplo como esto es similar a definir objetos en un lenguaje de programación. Además es posible "heredar", veamos como:

```
<xs:element name="destinatario" type="direcion"/>

<xs:complexType name="persona">
  <xs:sequence>
    <xs:element name="apellido" type="xs:string"/>
    <xs:element name="nombre" type="xs:string"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="direccion">
  <xs:complexContent>
    <xs:extension base="persona">
      <xs:sequence>
        <xs:element name="calle" type="xs:string"/>
        <xs:element name="ciudad" type="xs:string"/>
        <xs:element name="pais" type="xs:string"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Elementos vacios

Veamos como definirlos:

```
<xs:element name="nombre" type="nombre_t"/>

<xs:complexType name="nombre_t">
  <xs:attribute name="alias" type="xs:string"/>
</xs:complexType>
```

Esto nos permitiría definir elementos como estos:

<nombre alias="Capitan Morgan"/>

Elementos que contienen solo texto

Es posible definir elemento que solo contengan texto, lo podemos hacer por ejemplo utilizando un tipo base de schema y ampliandolo:

```
<xs:element name="Pie">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:integer">
        <xs:attribute name="pais" type="xs:string" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>

<pie pais="España">41</pie>
```

Elementos que contienen otros elementos y texto

Supongamos que queremos validar XMLs con este formato:

```
<carta>
  Querido Señor:<nombre>Alex Conceiro</nombre>.
  Su pedido <pedido>1032</pedido>
  será entregado el <fechaEntrega>2001-07-13</fechaEntrega>.
</carta>
```

Definiriamos un schema como el siguiente,ojo al atributo mixed, es necesario:

```
<xs:element name="carta" type="tipoCarta"/>

<xs:complexType name="tipoCarta" mixed="true">
  <xs:sequence>
    <xs:element name="nombre" type="xs:string"/>
    <xs:element name="pedido" type="xs:positiveInteger"/>
    <xs:element name="fechaEntrega" type="xs:date"/>
  </xs:sequence>
</xs:complexType>
```

Indicadores

Sirven para controlar como utilizar los elementos en los documentos

Hay siete tipos:

- Indicadores de orden
  - All
  - Choice
  - Sequence
- Indicadores de frecuencia de aparición
  - maxOccurs
  - minOccurs
- Indicadores de grupo
  - Grupos de elementos
  - Grupos de atributos

All

El indicador **all** especifica que los elementos hijos pueden aparecer en cualquier orden y que aparecen como mucho solo una vez

```
<xs:element name="persona">
  <xs:complexType>
    <xs:all>
      <xs:element name="nombre" type="xs:string"/>
      <xs:element name="apellido" type="xs:string"/>
    </xs:all>
  </xs:complexType>
</xs:element>
```

Choice

Indica que podran aparecer como elementos hijos uno u otro de los definidos, en este caso una persona puede ser trabajadora o miembro de un club por ejemplo:

```
<xs:element name="persona">
  <xs:complexType>
    <xs:choice>
      <xs:element name="empleado" type="empleado"/>
      <xs:element name="miembro" type="miembro"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

Sequence

Se indica así en que orden exacto deben aparecer los hijos y que deben aparecer todos una vez.

```
<xs:element name="persona">
   <xs:complexType>
    <xs:sequence>
      <xs:element name="nombre" type="xs:string"/>
      <xs:element name="apellido" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

maxOccurs y minOccurs

Indica respectivamente el máximo y mínimo número de veces que aparecerá un elemento, en este caso una persona con un nombre dado puede tener desde 0 hasta 10 nombres de hijos asociados.

```
<xs:element name="persona">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="nombre" type="xs:string"/>
      <xs:element name="nombreHijos" type="xs:string"
      maxOccurs="10" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Indicadores de grupo y de atributo

Sirven para agrupar elementos o atributos bajo una misma denominación:

```
<xs:group name="grupoPersona">
  <xs:sequence>
    <xs:element name="nombre" type="xs:string"/>
    <xs:element name="apellido" type="xs:string"/>
    <xs:element name="cumpleaños" type="xs:date"/>
  </xs:sequence>
</xs:group>
```

Ahora es posible incluir este grupo dentro de otra definición, observesé el atributo **href** para hacer referencia a una definición dada:

```
<xs:element name="persona" type="infoPersona"/>

<xs:complexType name="infoPersona">
  <xs:sequence>
    <xs:group ref="grupoPersona"/>
    <xs:element name="pais" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

Podemos hacer algo similar pero agrupando atributos

```
<xs:attributeGroup name="grupoAttrPersona">
  <xs:attribute name="nombre" type="xs:string"/>
  <xs:attribute name="apellidos" type="xs:string"/>
  <xs:attribute name="cumpleaños" type="xs:date"/>
</xs:attributeGroup>

<xs:element name="persona">
  <xs:complexType>
    <xs:attributeGroup ref="grupoAttrPersona"/>
  </xs:complexType>
</xs:element>
```

De esta forma se pueden reaprovechar muchas definiciones.

Namespaces

Los namespaces (espacios de nombres) son una forma de poder asociar los nombres de elementos que definimos a un nombre (el namespace) único para evitar duplicidad de nombres.

Me explicaré mejor, supongamos que definimos un grupo de atributos para definir tipos de dia (festivo,laboral,etc). Este grupo se define de la siguiente forma

```
<xs:attributeGroup name="DayType_attr">
      <xs:attribute name="Name" type="xs:string" use="required"/>
      <xs:attribute name="Abr" type="xs:string" use="required"/>
      <xs:attribute name="Type" type="xs:integer" use="required"/>
</xs:attributeGroup>
```

De esta forma podemos usar esta definición por ejemplo de la siguiente manera:

```
<DayType Name="Sabado" Abbr="SABA" Type="2"/>
```

Esto nos permitiría definir que el Sábado es de tipo “2” y nosotros en otra parte definiremos que significa que un dia sea de tipo “2”.

El caso es que supongamos que otra persona define en otro esquema un grupo de atributos y también le llama DayType_attr y nosotros incluimos ese esquema dentro del nuestro. ¿Como diferenciar cuando queremos usar uno u otro?, la solución son los namespaces.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.unainet.net"
xmlns:tns="http://www.unainet.net"
elementFormDefault="qualified">

...

</xs:schema>
```

Esto significa lo siguiente:

xmlns:xs=”http://www.w3.org/2001/XMLSchema"

Usamos un namespace xml (xmlsn), cuyo prefijo será “xs” y cuya nombre es el sigueinte URI “http://www.w3.org/2001/XMLSchema”.

Esto NO indica que haya un esquema en esa dirección ni nada por el estilo,simplemente se usan URIs para definir namespaces para asegurar de que no colisionan nombres de namespaces.

Ahora podemos usar los elementos del w3 consortium prefijandolos con “xs”, tal y como se puede ver en el ejemplo anterior.

Por otro lado tenemos el atributo **targetNamespace** que nos indica que los elementos definidos en este esquema pertenecerán a un namespace llamado “http://www.unainet.net” y ademas mediante el atributo **xmlns:tns="http://www.unainet.net"** indicamos que el

namesapce http://www.unainet.net se prefija con "tns".

Por último el atributo **elementFormDefault="qualified"** nos obliga a prefijar todos los elementos que usemos en el esquema.

Reaprovechar definiciones usando includes

Supongamos que queremos reaprovechar definiciones de elementos comunes a varios esquemas para no reescribir una y otra vez estas.

Se puede hacer un esquema coumun y enlazar con este desde el resto de esquemas.

Esto se hace mediante un **include**.

Veamos un ejemplo en el que reaprovechamos un elemento que consiste en un tipo de datos carácter mayúsucla. Metemos esta definicion en un fichero common.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:tlv="http://www.telvent.com"
targetNamespace="http://www.telvent.com" elementFormDefault="qualified">
<xs:simpleType name="SingleUpperCaseChar">
      <xs:restriction base="xs:string">
        <xs:pattern value="[A-Z]"/>
      </xs:restriction>
</xs:simpleType>
</xs:schema>
```

Ahora queremos reaprovecharla desde otro esquema, pongamos uno que se llama lin.xsd,fijaos como se utiliza el elemento **include**:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:tlv="http://www.telvent.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" targetNamespace="http://www.telvent.com"
elementFormDefault="qualified" attributeFormDefault="unqualified">
      <xs:include schemaLocation="common.xsd"/>
</xs:schema>
```

Ahora es posible utilizar SingleUpperCaseChar y cualquier otra definición que hayamos hecho en common.xsd