Tema 4. "Definición de esquemas y vocabularios en XML"

Actualizado al 17/11/2022

4.1. XML. Estructura y sintaxis

Hasta ahora hemos trabajado sin usar toda la potencia de XML.

Nos hemos limitado a confeccionar documentos bien formados, lo que quiere decir que pueden ser leídos por un proceso automatizado, interpretando bien sus etiquetas, atributos y contenido.

Pero hay una segunda parte. A los documentos, además de estar bien formados, se les puede pedir que cumplan ciertas reglas, para que la información que contienen disfrute de una buena salud. A grandes rasgos, esas reglas tienen que ver con:

- Poder indicar qué nombres de etiquetas pueden y/o deben estar presentes.
- De qué forma deben aparecer estructurados unos elementos dentro de otros.
- Cuántas veces puede aparecer un elemento como mínimo y como máximo.
- Qué atributos pueden y/o deben tener los elementos y de qué tipo son sus valores.
- Etc.

Cuando un documento cumple con un conjunto de reglas diseñado para él se dice que, además de estar bien formado, es un documento **válido**.

En este tema abordaremos, por tanto, la **validación de documentos XML**.

En la primera unidad vimos que un documento XML básico estaba formado por:

- **Prólogo:** Informa al intérprete encargado de procesar el documento de todos aquellos datos que necesita para realizar su trabajo. Consta de dos partes:
    - Definición de XML: Donde se indica la versión de XML que se utiliza, el código de los datos a procesar y la autonomía del documento. Este último dato hasta ahora siempre ha sido "yes", puesto que los documentos no requerían otros.
    - Declaración del tipo de documento: Hasta el momento sólo hemos dicho que es el nombre del ejemplar precedido de la cadena <!DOCTYPE y separado de ésta por (al menos) un espacio. Veremos más posteriormente.
- **Ejemplar:** Contiene los datos del documento que se quiere procesar. Es el elemento raíz del documento y ha de ser único. Está compuesto de elementos estructurados según una estructura de árbol en la que el elemento raíz es el ejemplar y las hojas los elementos terminales, es decir, aquellos que no contienen elementos. Los elementos pueden estar a su vez formados por atributos.

Lo que nos falta por abordar es la “validez” de dichos documentos, en términos de poder definir con qué estructura y de qué naturaleza son los datos que aparecen en él, qué cosas deben aparecer obligatoriamente, qué valores se espera que sean usados, etc. Como ya se ha dicho, podremos así pasar a generar documentos XML “válidos”, y no sólo “bien formados”.

Veamos a continuación algunos conceptos previos necesarios para abordar con garantías las dos técnicas de validación de documentos XML más extendidas: DTD y XML Schema.

4.1.1. Declaración de tipo de documento (para DTDs)

Ya habíamos visto que permite al autor definir restricciones y características en el documento, aunque no habíamos profundizado en las partes que la forman:

4.1.1.1. Declaración

Es la declaración del tipo de documento propiamente dicha. Comienza con *<!DOCTYPE*, seguido de al menos un espacio, y después el nombre del tipo, que ha de ser idéntico al del ejemplar del documento XML en el que se está trabajando. Veamos un ejemplo que únicamente contiene la declaración (ya lo conocemos):

```
<!DOCTYPE nombre_ejemplar>
```

4.1.1.2. Definición

Permite asociar al documento una definición de tipo DTD (Document Type Definition), la cual se encarga de definir las cualidades del tipo. Es decir, define los tipos de los elementos, atributos y notaciones que se pueden utilizar en el documento, así como las restricciones del documento, valores por defecto, etc. Se abordará en profundidad en el apartado 4.2.

De momento sólo diremos que los DTD's se componen de declaraciones de marcado, las cuales pueden ser internas o externas.

- Internas: definidas en el mismo documento. No se comparten con ningún otro. Se localizan dentro de unos corchetes que siguen a la declaración de tipo del documento:

```
<!DOCTYPE nombre_ejemplar [
  <!ELEMENT elemento1 (elemento2, elemento3+)>
  <!ELEMENT elemento2 (#PCDATA)>
  <!ELEMENT elemento3 (#PCDATA)>
]>
```

- Externas: definidas en un archivo separado (generalmente con extensión .dtd). Pueden ser compartidas por varios archivos XML:

```
<!DOCTYPE nombre_ejemplar SYSTEM "../dtds/fic_definicion.dtd">
```

En un archivo XML puede coexistir una mezcla de declaraciones internas y externas.

Si existen externas, se debe indicar además dos cosas:

- Dónde encontrar las declaraciones, lo cual se hará mediante una URI.
  - Es lo que hemos indicado mediante "*SYSTEM "…""*
- Indicar en la declaración de XML que el documento no es autónomo.
  - Un documento autónomo se indica mediante *standalone="yes"*.
  - Un documento que debe ser validado por DTD, como en este caso, se indica mediante *standalone="no"*.

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

Sobre estas declaraciones externas, caben dos posibilidades:

- Usar identificadores de sistema (SYSTEM, como en el ejemplo anterior)
- Usar identificadores públicos.

La diferencia reside en que los primeros se destinan a un uso privado, mientras que los públicos se usan para identificar una entrada en un catálogo, supuestamente público y del que tienen conocimiento los sistemas a los que se destinan los XMLs.

La sintaxis es diferente, pues en los públicos se indica el identificador y la URI separadamente, mientras que en los SYSTEM se acepta que el identificador y la URI de localización del DTD son la misma cosa.

En los públicos, el identificador puede ser utilizado por el procesador XML para intentar generar un URI alternativo, posiblemente basado en alguna tabla.

Ya hemos visto un ejemplo con SYSTEM. Veamos un ejemplo con PUBLIC (id+URI), concretamente esta es la forma en la que se declara un documento XHTML:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
```

También conviene saber que primero se procesa el subconjunto interno y después el externo. Es decir, las declaraciones internas prevalecen sobre las externas, lo que permite sobrescribir declaraciones externas compartidas entre varios documentos para ajustarlas a un documento concreto de forma más específica.

4.1.2. Definición de la sintaxis de documentos XML

Recordamos que en estos documentos las etiquetas de marcado describen la estructura del documento.

4.1.2.1. Elementos

Un elemento es un grupo formado por una etiqueta de apertura (donde puede opcionalmente contener atributos), otra de cierre y el contenido que hay entre ambas.

En los documentos de lenguajes de marcas, la distribución de los elementos está jerarquizada según una estructura de árbol, lo que implica que es posible anidarlos, pero no entrelazarlos.

El orden de los elementos es importante, pues se pueden recorrer secuencialmente (sobre todo si su etiqueta es la misma).

4.1.2.2. Atributos

El orden de los atributos no es importante, pero no puede haber dos con el mismo nombre.

Sabemos que los atributos no pueden tener nodos que dependan de ellos, por tanto, solo pueden corresponder con hojas de la estructura de árbol que jerarquiza los datos. ¿Significa

esto que todas las hojas van a ser atributos? Pues no, es cierto que los atributos son hojas, pero las hojas pueden ser atributos o elementos.

4.1.2.3. Elección de diseño: ¿atributo o elemento?

¿Qué criterios podemos utilizar para decidir si un dato del documento que se pretende estructurar ha de representarse mediante un elemento o un atributo? Aunque no siempre se respetan, podemos usar los siguientes criterios:

El dato será un elemento si cumple alguna de las siguientes condiciones:

- Contiene subestructuras de datos (es un dato complejo).
- Es de un tamaño considerablemente grande.
- Su valor posee una longitud o una naturaleza muy variable.
- Su valor va a ser mostrado a un usuario o aplicación que requiere de formato, como el texto enriquecido.

Los casos en los que el dato será un atributo son:

- El dato es de pequeño tamaño y su valor raramente cambia.
- El dato tiene una naturaleza y conjunto de posibles valores bien definidos.
- El dato guía el procesamiento XML, pero no se va a mostrar.

4.1.2.4. Namespaces

Los espacios de nombres nos permiten:

- Diferenciar entre los elementos y atributos de distintos vocabularios (con diferentes significados) que comparten nombre de identificador.
- Agrupar todos los elementos y atributos relacionados de una aplicación XML para que el software pueda reconocerlos con facilidad.

¿Cómo se declaran?

```
xmlns:"URI_namespace"
```

La declaración anterior establece el espacio de nombres predeterminado (por defecto) para cualquier etiqueta que aparezca en el fichero XML.

En ocasiones, es necesario hacer referencia a más de un espacio de nombres en el mismo XML, en cuyo caso necesitamos “alias”, en forma de prefijos, que diferencien a unos de otros:

```
xmlns:prefijo="URI_namespace"
```

En ambos casos *URI_namespace* es la localización del conjunto del vocabulario del espacio de nombres al que se hace referencia.

4.2. DTD - Definiciones de tipo de documento

Ya hemos visto que es una técnica de validación consistente en el establecimiento de una serie de declaraciones que imponen requisitos que el XML ha de cumplir para garantizar su validez.

Estas definiciones están formadas por una relación precisa de qué elementos pueden aparecer en un documento y dónde, así como el contenido y los atributos del mismo. Garantizan que los datos del documento XML cumplen las restricciones que se les haya impuesto en el DTD, ya que estas últimas permiten:

- Especificar la estructura del documento.
- Reflejar una restricción de integridad referencial mínima utilizando (ID e IDREF).
- Utilizar unos pequeños mecanismos de abstracción comparables a las macros, que son las entidades.
- Incluir documentos externos.

¿Cuáles son los inconvenientes de los DTD? Los principales son:

- Su sintaxis no es XML.
- No soportan espacios de nombres.
- No definen tipos para los datos. Solo hay un tipo de elementos terminales, que son los datos textuales.
- No permite las secuencias no ordenadas.
- No es posible formar claves a partir de varios atributos o elementos.
- Una vez que se define un DTD no es posible añadir nuevos vocabularios.

También hemos visto que:

- Cuando están definidas dentro del documento XML se ubican entre corchetes, después del nombre del ejemplar, en el elemento <!DOCTYPE>.
- Cuando está definido en un fichero, éste es de texto plano, con extensión ".dtd".

4.2.1. Declaraciones de tipos de elementos terminales

La declaración de tipos de elementos está formada por la cadena "<!ELEMENT" seguida de un espacio, del nombre del elemento XML que se declara y, por último, la declaración del contenido que puede tener dicho elemento. Por ejemplo:

```
<!ELEMENT nombre_elemento declaración_de_contenido>
```

Los tipos terminales son aquellos elementos que se corresponden con hojas de la estructura de árbol formada por los datos del documento XML asociado al DTD.

TIPO TERMINAL = ELEMENTO SIN SUB-ELEMENTOS

En el caso de elementos terminales, la declaración de contenido es dada por uno de los siguientes valores:

- EMPTY: Indica que el elemento no es contenedor. Por ejemplo, la siguiente definición muestra un elemento "E" que no contiene nada:

```
<!ELEMENT E EMPTY>
```

- ANY: Permite que el contenido del elemento sea cualquier cosa. Un ejemplo de definición de un elemento "A" de este tipo es:

```
<!ELEMENT A ANY>
```

- (#PCDATA): Parsed Character Data. Indica que el elemento contiene texto pero que será analizado en busca de etiquetas. Por ello, no puede contener elementos, es decir solo puede contener datos de tipo carácter, pero exceptuando los siguientes:

  < & [ ] >

  Si es de este tipo, el elemento "P" tendrá una definición como:

```
<!ELEMENT P (#PCDATA)>
```

4.2.2. Declaraciones de tipos de elementos no terminales

Nos referimos ahora a los elementos que están formados por otros elementos. Para definirlos utilizamos referencias a los grupos que los componen tal y como muestra el ejemplo:

```
<!ELEMENT A (B,C)>
```

En este caso se ha definido un elemento "A" que está formado por un elemento "B" seguido de un elemento "C".

¿Y qué sucede cuando un elemento puede aparecer en el documento varias veces, hay que indicarlo de algún modo? Pues sí, también hay que indicar cuando un elemento puede no aparecer. Para ello usamos los siguientes operadores, que nos permiten definir la cardinalidad de un elemento:

4.2.2.1. **Opción: ?**

Indica que el elemento no es obligatorio. En el siguiente ejemplo el subelemento *trabajo* es opcional, mientras que *casa* no lo es. Ambos sólo pueden aparecer una vez como mucho.

```
<!ELEMENT telefono ( trabajo? , casa )
```

4.2.2.2. **Uno-o-más: +**

El elemento debe estar presente al menos una vez, pudiendo aparecer varias veces.

En este ejemplo vemos la definición de un elemento "pc", que se compone de una placa, una CPU y uno o varios discos duros:

```
<!ELEMENT pc ( placa , cpu , disco+ )
```

La repetición puede afectar a un grupo de sub-elementos. En este otro ejemplo definimos un elemento “provincia”, formado por su nombre y otro grupo, que puede repetirse varias veces:

```
<!ELEMENT provincia ( nombre , ( cp , ciudad )+ )
```

4.2.2.3. **Cero-o-más: ***

El elemento es opcional, pudiendo estar presente cero, una o varias veces. Si en el ejemplo anterior quisiéramos dejar el grupo (cp, ciudad) como opcional, lo habríamos hecho así:

```
<!ELEMENT provincia (nombre , ( cp , ciudad )* )
```

4.2.2.4. **Elección: |**

Cuando se utiliza (sustituyendo a las comas en la declaración de grupos) indica que para formar el documento XML hay que elegir entre uno de los elementos separados por este operador.

En el ejemplo siguiente, el documento XML tendrá elementos “autorización”, que contendrán un sub-elemento “fecha”, otro “alumno” y otro a elegir entre “padre”, “madre” o “tutor”:

```
<!ELEMENT autorización ( fecha , alumno , (padre|madre|tutor) )
```

4.2.3. Declaraciones de atributos

Ya sabemos cómo declarar elementos, ahora veamos el modo de declarar los atributos, los cuales necesariamente irán asociados a un elemento.

Para ello utilizamos la cadena <!ATTLIST seguida del nombre del elemento asociado al atributo que se declara, luego el nombre del atributo propiamente dicho, seguido del tipo de atributo y del modificador:

```
<!ATTLIST elemento atributo tipo modificador>
```

- El tipo define su “naturaleza”, algo parecido al tipo de dato.
- El modificador determina la obligatoriedad y si se refiere a un valor constante.

!ATTLIST puede usarse de dos formas:

- Para declarar varios atributos asociados a un mismo elemento, todos de una vez.

```
<!ATTLIST elemento
  atributo1 tipo1 modificador1
  atributo2 tipo2 modificador2
  atributo3 tipo3 modificador3
>
```

- Repetirse varias veces para enumerar uno a uno los atributos de un elemento.

```
<!ATTLIST elemento atributo1 tipo1 modificador1>
<!ATTLIST elemento atributo2 tipo2 modificador2>
<!ATTLIST elemento atributo3 tipo3 modificador3>
```

Las dos definiciones anteriores serían equivalentes.

4.2.3.1. **Tipos de atributos**

A continuación, veremos los posibles valores para el "tipo" de los atributos.

Enumeración

El atributo solo puede tomar uno de los valores determinados dentro de un paréntesis y separados por el operador |.

```
<!ATTLIST fecha dia_semana
  (lunes|martes|miércoles|jueves|viernes|sábado|domingo)
  #REQUIRED >
```

CDATA

Se utiliza cuando el atributo es una cadena de texto, sin ninguna restricción en su contenido.

```
<!ATTLIST persona nombre CDATA #REQUIRED >
```

ID

Permite declarar un atributo identificador en un elemento, es decir, determinar que no puede repetirse el valor de dicho atributo, pues ha de ser único en el documento.

Además, el valor ha de ser un nombre válido en XML. Hay que tener en cuenta que los números no son nombres de identificadores válidos en XML, por tanto no son un identificador legal de XML. Para resolverlo suele incluirse un prefijo en los valores, con caracteres alfabéticos. Por ejemplo, dada la definición:

```
<!ATTLIST ticket numero ID #REQUIRED >
```

Serían válidos estos casos:

```
...
   <ticket numero="TK00001" fecha="1/1/2016">...</ticket>
   <ticket numero="TK00002" fecha="1/1/2016">...</ticket>
   <ticket numero="TK00003" fecha="1/1/2016">...</ticket>
   <ticket numero="TK00004" fecha="2/1/2016">...</ticket>
...
```

Pero no serían válidos identificadores de tickets que fuesen números enteros simplemente.

IDREF

Permite hacer referencias a identificadores. En este caso el valor del atributo ha de corresponder con el de un identificador de un elemento existente en el documento. Por ejemplo podría existir un elemento “abono” que hiciera referencia al “ticket” que se abona:

```
<!ATTLIST abono
  numero ID #REQUIRED
  ticket IDREF #REQUIRED
  fecha...
  >
```

Siguiendo el mismo ejemplo de documento XML anterior, podría aparecer una referencia así:

```
...
   <abono
      numero="AB00002"
      ticket=”TK00003”
      fecha="4/1/2016">
      ...
   </abono>
...
```

Que representaría el abono realizado por la devolución del tercer ticket del ejemplo anterior.

NMTOKEN

Permite determinar que el valor de un atributo ha de ser una sola palabra compuesta por los caracteres permitidos por XML para componer nombres de etiquetas, atributos e identificadores en general.

4.2.3.2. **Modificadores de atributos**

Respecto a los posibles modificadores, son los siguientes:

#IMPLIED

Determina que el atributo sobre el que se aplica es opcional.

```
<!ATTLIST persona color_ojos CDATA #IMPLIED >
```

#REQUIRED

Determina que el atributo tiene carácter obligatorio, como en los ejemplos anteriores.

#FIXED

Permite definir un valor fijo para un atributo, independientemente de que ese atributo se defina explícitamente en una instancia del elemento en el documento XML (como debe ser). Al menos, al definirlo así forzamos a que no puede contener otro valor que no sea ese. Al usar este modificador, debemos indicar también dicho valor fijo. Por ejemplo:

```
<!ATTLIST regla habilitada CDATA #FIXED "true" >
```

Define un atributo “habilitada” para la entidad “regla” que, en caso de estar presente, sólo puede presentar el valor “true”. Se entenderá que la ausencia de este atributo es la que marca la inhabilitación de esta, ya que #FIXED no confiere un carácter de obligatoriedad en cuanto a “presencia”, sino sólo en cuanto a su valor. No tiene sentido declarar un atributo que sea obligatorio y con un valor fijo, pues estaría siempre presente y con el mismo valor.

4.2.4. Declaraciones de entidades.

Las entidades nos permiten definir constantes en un documento XML. Cuando se usan dentro del documento XML se limitan por "&" y ";", por ejemplo:

```
&entidad;
```

Al procesar el documento XML, el intérprete sustituye la entidad por el valor que se le ha asociado en el DTD.

No admiten recursividad, es decir, una entidad no puede hacer referencia a ella misma.

Para definir una entidad en un DTD se usa el elemento <!ENTITY>

Las entidades pueden ser de tres tipos: externas, internas y de parámetro.

4.2.4.1. Internas

Existen cinco entidades predefinidas en el lenguaje, son:

Entidad	Valor (literal)	Valor (nombre)
&lt;	<	Signo “menor qué”
&gt;	>	Signo “mayor qué”
&quot;	“	Comillas dobles
&apos;	‘	Comilla simple (apóstrofe)
&amp;	&	Ampersand

Es posible definir una entidad interna nueva, mediante la sintaxis:

```
<!ENTITY nombre_entidad "valor de la entidad">
```

Por ejemplo, la definición:

```
<!ENTITY dtd "Definiciones de Tipo de Documento">
```

Podría ser usada:

```
<doc id="D0001" tipo="&dtd;">
  Este es un documento de tipo &quot;&dtd;&quot;
</doc>
```

4.2.4.2. Externas

El valor de la entidad no se indicará entre comillas, junto a su nombre, sino que estará contenido en un archivo separado.

A diferencia de las entidades internas, el contenido de los ficheros es analizado, por lo que:

- Puede que contengan un mero texto plano, con el valor de la entidad.
- O puede que contenga un trozo de XML interpretable.

En todo caso hay que entender que su contenido se reemplazará por la entidad, allí donde esta aparezca y que, tras ello, se procesará como parte del XML en el que aparece la entidad.

Un ejemplo de declaración de una entidad externa de sistema es:

```
<!ENTITY nom_entidad SYSTEM "http://localhost/dtd/entidad.txt">
```

Cuando es necesario incluir ficheros con formatos binarios, es decir ficheros que no se analicen, se utiliza la palabra reservada NDATA en la definición de la entidad. Por ejemplo:

```
<!ENTITY dibujo SYSTEM "imagen.gif" NDATA gif>
```

No obstante, se requiere además de una entidad de notación, que son un tipo diferente y que vamos a estudiar en el apartado 4.2.5 (a continuación).

4.2.4.3. De parámetro

Permite dar nombres a partes de un DTD y hacer referencia a ellas a lo largo del mismo DTD. Son especialmente útiles cuando varios elementos del DTD comparten listas de atributos o especificaciones de contenidos. Se denotan por *%entidad;*

```
<!ENTITY %direccion "calle, numero?, ciudad, cp">
<!ELEMENT alumno (dni, %direccion;)>
<!ELEMENT ies (nombre, %direccion;)>
```

4.2.4.4. De parámetro externas

Son como las anteriores, sólo que la definición de la entidad puede residir en otro archivo DTD:

```
<!ENTITY %persona SYSTEM "persona.dtd">
```

4.2.5. Declaraciones de notación

Cuando se incluyen ficheros binarios en un fichero XML, ¿cómo le decimos qué aplicación ha de hacerse cargo de ellos? La respuesta es utilizando notaciones. La sintaxis para ello es:

```
 <!NOTATION nombre SYSTEM aplicacion>
```

Siguiendo con el ejemplo de la entidad externa vista en 4.2.4, una notación llamada *gif* donde se indica que se hace referencia a un editor de formatos gif para visualizar imágenes será:

```
<!NOTATION gif SYSTEM "gifEditor.exe">
```

No obstante, no es rigurosamente necesario indicar la aplicación. Si no se desea vincular el contenido binario a una aplicación, bastaba con la declaración de la entidad ya vista:

```
<!ENTITY dibujo SYSTEM "imagen.gif" NDATA gif>
```

4.2.6. Secciones condicionales

Permiten incluir o ignorar partes de la declaración de un DTD. Para ello se usan dos tokens:

4.2.6.1. **INCLUDE**

Permite que se vea esa parte de la declaración del DTD. Su sintaxis es:

```
<![INCLUDE [Declaraciones visibles] ] >
```

Por ejemplo:

```
<![INCLUDE [ <!ELEMENT nombre (#PCDATA)>] ] >
```

4.2.6.2. **IGNORE**

Permite ocultar esa sección de declaraciones dentro del DTD. La forma de uso es:

```
<![IGNORE [Declaraciones ocultas] ] >
```

Por ejemplo:

```
<![IGNORE [<!ELEMENT clave (#PCDATA)>] ] >
```

4.2.6.3. **Ejemplo de secciones condicionadas y entidades de parámetros**

A continuación se muestra un DTD en el que aparecen dos secciones, que resultarán condicionadas según el valor de las entidades de parámetros (%datos_basicos y %datos_ampliados):

```
<![ %datos_basicos; [
   <!ELEMENT persona (nombre, edad)>
]]>
<![ %datos_ampliados; [
   <!ELEMENT persona (nombre, apellidos, edad, ciudad)>
]]>
<!ELEMENT nombre (#PCDATA)>
<!ELEMENT apellidos (#PCDATA)>
<!ELEMENT edad (#PCDATA)>
<!ELEMENT ciudad (#PCDATA)>
```

Así que, según sean los valores de %datos_basicos y %datos_ampliados, puede optarse por forzar a que el elemento “persona” posea 2 o 4 sub-elementos.

La aplicación práctica podría verse en estos dos ejemplos de documento XML, cada uno de los cuales opta por una posibilidad:

Para limitar a datos básicos:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE persona SYSTEM "persona.dtd" [
   <!ENTITY % datos_basicos "INCLUDE">
   <!ENTITY % datos_ampliados "IGNORE">
]>
<persona>
   <nombre>Elsa</nombre>
   <edad>23</edad>
</persona>
```

Para obligar a datos ampliados:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE persona SYSTEM "persona.dtd" [
   <!ENTITY % datos_basicos "IGNORE">
   <!ENTITY % datos_ampliados "INCLUDE">
]>
<persona>
   <nombre>Ana</nombre>
   <apellidos>Sanz Tin</apellidos>
   <edad>19</edad>
   <ciudad>Pamplona</ciudad>
</persona>
```

Haber puesto IGNORE en ambos o INCLUIDE en ambos podría haber ocasionado que ningún XML fuera válido.

4.3. XML Schema

Es una técnica de validación de XML diferente. DTD presenta algunas carencias, que están subsanadas con XML schema. Las principales son:

- Poder especificar tipos de datos para los elementos y atributos, indicando además algunas restricciones en sus valores.
- Poder hacer uso de espacios de nombres para las reglas de validación.
- Usar sintaxis XML, al igual que en el documento a validar.

Los archivos serán de texto plano, aunque para ellos se usa la extensión ".xsd".

Los elementos XML que se utilizan para generar un esquema han de pertenecer al espacio de nombre "XML Schema", que es: http://www.w3.org/2001/XMLSchema.

El ejemplar de estos ficheros es <xs:schema>, contiene declaraciones para todos los elementos y atributos que puedan aparecer en un documento XML asociado válido. Con lo dicho hasta ahora, un fichero XSD básico tendría el aspecto:

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

Los elementos hijos inmediatos de este ejemplar (que irán en los puntos suspensivos) son etiquetas <xs:element> (elementos). Obviamente, el ejemplar del XML a validar debe figurar como elemento. Si imaginamos que el XSD anterior valida un XML con raíz <nota>, su aspecto será:

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

4.3.1. Tipos de datos simples

Siguiendo el ejemplo anterior, imaginemos que el XML sólo contiene el elemento raíz (<nota>) y, para que sea válido, deseamos indicar que su contenido ha de ser un texto. Por ejemplo:

```
<?xml version="1.0" encoding="UTF-8"?>
<nota>
  Hola mundo.
</nota>
```

Para ello, en el XSD, usamos el atributo “type” de la siguiente forma:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="nota" type="xs:string">
  </xs:element>
</xs:schema>
```

Estos tipos de datos se pueden aplicar al contenido de los elementos y a los valores de los atributos. A continuación se relacionan los más importantes:

- string — Cadena de caracteres UNICODE.
- boolean — Valores lógicos “true” o “false”.
- integer — Número entero, positivo o negativo.
- positiveInteger — Número entero positivo.
- negativeInteger — Número entero negativo.
- decimal — Número decimal, en coma fija, por ejemplo: 8,97.
- dateTime — Fecha y hora absolutas (no una duración de tiempo).
- date — Fecha absoluta, en formato CCYY-MM-DD (año-mes-día) *(*)*.
- time — Hora, en el formato hh:mm:ss (horas:minutos:segundos).
- gYearMonth — Mes y año determinado, con formato CCYY-MM*(*)*.
- gYear — Año gregoriano, el formato usado es CCYY*(*)*.
- gMonthDay — Día (1-31) y mes (1-12), mediante el formato –MM-DD.
- gDay — Día del mes (1-31), mediante el formato –DD.
- gMonth — Mes, mediante el formato –MM. Por ejemplo, febrero es –02.
- anyURI — Cualquier URI válida.

**CCYY significa que el año puede expresarse con 4 dígitos o sólo con los 2 últimos. Los valores cuyo carácter alias se duplica significan que debe rellenarse con ceros por la izquierda. Así, el mes (MM) de abril es “04”, las 8:05 (hora hh:mm:ss) es “08:05:00”.*

Algunos tipos más:

duration

Representa una duración de tiempo expresado en años, meses, días, horas, minutos segundos. El formato utilizado es: PnYnMnDTnHnMnS.

Por ejemplo para representar una duración de 2 años, 4 meses, 3 días, 5 horas, 6 minutos y 10 segundos habría que poner:

```
P2Y4M3DT5H6M7S
```

Se pueden omitir los valores nulos, luego una duración de 2 años será P2Y. Para indicar una duración negativa se pone un signo – precediendo a la P.

language

Representa identificadores de idiomas, sus valores están definidos en el estándar RFC 1766.

Varios presentes en DTD

Nos referimos a los tipos: ID, IDREF, ENTITY, NOTATION, MTOKEN. Representan lo mismo que en los DTD's (ver apartado 4.2.3).

Podéis ampliar información sobre tipos de datos en la URL:

https://www.w3.org/TR/xmlschema-2/

4.3.2. Restricciones de los tipos de datos simples

Se refieren a las restricciones que podemos aplicar sobre los valores de los datos de un elemento o atributo (longitud máxima, enumeración de posibilidades…).

Se aplican al definir lo que se denomina como "tipos simples" (tipos de datos definidos por el usuario, pero que almacenan valores simples). Los tipos simples se definen mediante la etiqueta <xs:simpleType>.

En el siguiente ejemplo vamos a definir un tipo simple para los valores del elemento "nota":

```
<xs:element name="nota">
  <xs:simpleType>...</xs:simpleType>
</xs:element>
```

Entre los puntos suspensivos habrá que indicar las restricciones. Se hace mediante la etiqueta <xs:restriction>, quedando el conjunto:

```
<xs:element name="nota">
  <xs:simpleType>
    <xs:restriction base="xs:string">...</xs:restriction>
  </xs:simpleType>
</xs:element>
```

Como puede verse, al definir la restricción se indica también el tipo "base", que como su nombre indica es el tipo de dato que sirve como base y sobre el cual aplicar las restricciones.

Si no existieran restricciones, habría bastado con el ejemplo anterior, usando *type="xs:string"*.

A continuación, veremos algunas posibilidades para terminar de dar forma a esas restricciones.

length, minlength, maxlentgh

Longitud del tipo de datos (fija, mínima y máxima). En este ejemplo, se limita el texto del contenido de la etiqueta “nota” a 160 caracteres como máximo:

```
<xs:element name="nota">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:maxLength value="160"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

enumeration

Restringe a un determinado conjunto de valores. En este ejemplo se define un elemento “estado” cuyo contenido puede corresponderse con “conectado” u “ocupado”:

```
<xs:element name="estado">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="conectado"/>
      <xs:enumeration value="ocupado"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

maxInclusive, minInclusive, maxExclusive, minExclusive

Fijan los límites superiores/inferiores del tipo de datos. Cuando son “Inclusive” el valor que se determine es parte del conjunto de valores válidos para el dato, mientras que cuando se utiliza “Exclusive”, el valor dado no pertenece al conjunto de valores válidos.

totalDigits, fractionDigits

Número de dígitos totales y decimales de un número decimal.

El siguiente ejemplo serviría para validar notas, asumiendo que:

- Han de ser estrictamente mayores de 0
- Su valor no puede ser superior a 10 (pero sí igual).
- Deben expresarse a lo sumo con dos decimales.

```
<xs:element name="nota">
  <xs:simpleType>
    <xs:restriction base="xs:decimal">
      <xs:totalDigits value="4"/>
      <xs:fractionDigits value="2"/>
      <xs:minExclusive value="0"/>
      <xs:maxInclusive value="10"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

NOTA: El número total de dígitos es 4 porque hay que descontarle los 2 que serán parte decimal (2 para la parte entera + 2 para la parte decimal = 4 dígitos totales).

pattern

Permite construir máscaras que han de cumplir los datos de un elemento. La siguiente tabla muestra algunos de los caracteres que tienen un significado especial para la generación de las máscaras:

Patrón	Significado	Patrón	Significado
[A-Z a-z]	Letra.	AB	Cadena que es la concatenación de las cadenas A y B.
[A-Z]	Letra mayúscula.	A?	Cero o una vez la cadena A.
[a-z]	Letra minúscula.	A+	Una o más veces la cadena A.
[0-9]	Dígitos decimales.	A*	Cero o más veces la cadena A.
\D	Cualquier carácter excepto un dígito decimal.	[abcd]	Alguno de los caracteres que están entre corchetes.
(A)	Cadena que coincide con A.	[^abcd]	Cualquier carácter que no esté entre corchetes.
A | B	Cadena que es igual a la cadena A o a la B.	\t	Tabulación.

Las máscaras son un mecanismo muy potente para validar valores alfanuméricos. A continuación, vemos un ejemplo de cómo validar que un DNI sea introducido a base de 8 dígitos y una letra:

```
<xs:element name="dni">
 <xs:simpleType>
  <xs:restriction base="xs:string">
   <xs:pattern value="[0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][A-Z]"/>
  </xs:restriction>
 </xs:simpleType>
</xs:element>
```

4.3.3. Elementos del lenguaje

INFO: El contenido de este punto 4.3.3 es diferente y complementario a los contenidos de la plataforma. Se recomienda leer primero el punto 4.3.4

Veamos algunas técnicas adicionales para sacar partido a la declaración de XSDs.

4.3.3.1. Definición de atributos

Casi todo lo visto para el contenido de un elemento, es válido también para los atributos. Caben dos salvedades importantes:

- Los atributos se definen con la etiqueta <attribute>, dentro del elemento al que han de pertenecer.
- Los tipos de datos sólo pueden ser simples.

Un ejemplo de definición de atributos:

```
<xs:element name="terminal">
  <xs:attribute name="numero" type=”xs:string” />
  <xs:attribute name="marca" type=”xs:string” />
  <xs:attribute name="pin" type=”xs:string”>
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:pattern value="[0-9][0-9][0-9][0-9] "/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
</xs:element>
```

Donde vemos que “terminal” puede poseer como atributos “numero”, “marca” y “pin”. Este último posee una restricción de que ha de componerse con 4 guarismos del 0 al 9.

Podemos complementar la definición de los atributos con algunos atributos (valga la redundancia) de la etiqueta <attribute>. Algunos de ellos muy interesantes:

Establecer un valor por defecto:

```
<xs:attribute name="idioma" type="xs:string" default="EN"/>
```

Fijar un valor constante:

```
 <xs:attribute name=" idioma " type="xs:string" fixed="EN"/>
```

Hacer que sea obligado:

Si no se indica lo contrario, todos los atributos son opcionales.

```
<xs:attribute name="lang" type="xs:string" use="required"/>
```

4.3.3.2. Reutilizar tipos declarados

Los tipos de datos pueden declararse y reutilizarse. Para ello basta con:

- No declararlos en el ámbito de un elemento, sino fuera de él.
- Indicar un nombre mediante su atributo “name”.
- Reutilizarlo donde haga falta, mediante el atributo “type”.

Ejemplo. Veamos como declarar un tipo compuesto “dirección” y usarlo después:

```
<xs:complexType name="direccion">
  <xs:sequence>
    <xs:element name="tipo_via"   type="xs:string"  />
    <xs:element name="nombre_via" type="xs:string"  />
    <xs:element name="numero”     type="xs:integer" />
    <xs:element name="localidad”  type="xs:string"  />
    <xs:element name="provincia”  type="xs:string"  />
  </xs:sequence>
</xs:complexType>
...
<xs:element name="persona">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="nombre" type="xs:string" />
      <xs:element name="direccion" type="direccion"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

4.3.3.3. Cardinalidad de los elementos

Tras estudiar el punto 4.3.4, vemos que se pueden componer tipos complejos a base de indicar secuencias (ordenadas o no) y selecciones de elementos.

La cardinalidad se refiere al número de ocurrencias mínimo y máximo con el que cada elemento de estas estructuras complejas puede presentarse. Por defecto:

- Cada elemento de una secuencia ha de aparecer una y sólo una vez (ordenada o no).
- Por cada selección debe aparecer obligatoriamente uno y sólo uno de sus opciones.

Podemos cambiar este comportamiento indicando el número mínimo y/o máximo de veces que un elemento puede aparecer en una secuencia.

Supongamos una forma de validar una etiqueta <familia>, como sigue:

```
<xs:element name="familia">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="padre" type="xs:string" />
      <xs:element name="madre" type="xs:string" />
      <xs:element name="hijo" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Ahora entendamos que cualquiera de los elementos es opcional. Quedaría así:

```
<xs:element name="familia">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="padre" type="xs:string" minOccurs="0" />
      <xs:element name="madre" type="xs:string" minOccurs="0" />
      <xs:element name="hijo" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Supongamos que queremos limitar la enumeración a 2 padres, 2 madres y 20 hijos:

```
<xs:element name="familia">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="padre" type="xs:string"
                               minOccurs="0" maxOccurs="2" />
      <xs:element name="madre" type="xs:string"
                               minOccurs="0" maxOccurs="2" />
      <xs:element name="hijo" type="xs:string"
                               minOccurs="0" maxOccurs="20" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Es decir, podrán aparecer 0, 1 o 2 padres, después 0, 1 o 2 madres y luego podrán aparecer hijos o no, hasta un máximo de 20.

El orden es importante porque hemos utilizado <sequence> y no <all>, en cuyo caso podrían haber aparecido los elementos con las mismas restricciones de cardinalidad, pero en cualquier orden.

4.3.3.4. Agrupaciones

La etiqueta *<xs:group>* permite nombrar agrupaciones de elementos y de atributos, para hacer referencia a ellas en múltiples puntos en los que puede hacer falta. Por ejemplo:

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

<xs:group name="datos_cliente">
  <xs:sequence>
    <xs:element name="razon_social" type="xs:string"/>
    <xs:element name="nombre_comercial" type="xs:string"/>
    <xs:element name="facturar_a" type="xs:string"/>
    <xs:element name="enviar_a" type="xs:string"/>
  </xs:sequence>
</xs:group>

<xs:complexType name="tipo_pedido">
  <xs:group ref="datos_cliente"/>
  <xs:attribute name="fecha" type="xs:date"/>
  <xs:attribute name="estado" type="xs:string"/>
</xs:complexType>

<xs:element name="pedido" type="tipo_pedido"/>

</xs:schema>
```

En este XSD se ha definido el grupo “datos_cliente”, que es utilizado en la definición del tipo “tipo_pedido”, que finalmente será el que tipifique al elemento “pedido” que aparece abajo.

4.3.3.5. Contenido mixto

Se refiere a la posibilidad de que un tipo de datos complejo pueda, a su vez, tener contenido propio como si se tratara de un elemento terminal.

Se consigue dando el valor *true* al atributo *mixed* del elemento *<xs:complexType>*.

Si quisiéramos que carta pudiera contener contenido mixto, lo declararíamos así:

```
<xs:element name="carta">
  <xs:complexType mixed="true">
    <xs:sequence>
      <xs:element name="nombre" type="xs:string"/>
      <xs:element name="id_pedido" type="xs:positiveInteger"/>
      <xs:element name="fecha_envio" type="xs:date"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Y luego podríamos hacer un uso de una <carta> como el que figura a continuación:

```
<carta>
  Dear Mr.<nombre>John Smith</nombre>.
  Your order <id_pedido>1032</id_pedido>
  will be shipped on <fecha_envio>2001-07-13</fecha_envio>.
</carta>
```

4.3.4. Tipos de datos complejos

Existen como forma de construir tipos estructurados, partiendo de la base de los tipos simples ya vistos en 4.3.1. Es decir, los tipos complejos se refieren a elementos que se componen de otros elementos.

Para definirlos usamos la etiqueta *<xs:complexType>.* Por ejemplo, un elemento <correo> (referido a un correo electrónico) se compone de varios sub-elementos. Podemos verlo así:

```
<xs:element name="correo">
  <xs:complexType>... </xs:complexType>
</xs:element>
```

Para especificar cómo se realiza esa composición, existen varias alternativas:

4.3.4.1. xs:sequence

Define que los elementos englobados en ella han de aparecer en ese orden. Volviendo a nuestro ejemplo, si entendemos que el correo se compone de “para”, “de”, “asunto” y “cuerpo” (en ese orden), el aspecto del XSD sería:

```
<xs:element name="correo">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="para" type="xs:string"/>
      <xs:element name="de" type="xs:string"/>
      <xs:element name="asunto" type="xs:string"/>
      <xs:element name="cuerpo" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

4.3.4.2. xs:all

Secuencias no ordenadas. Representa secuencias, como las anteriores, salvo por el hecho de que no importa en qué orden aparezcan los elementos contenidos en ella.

4.3.4.3. xs:choice

Representa alternativas, hay que tener en cuenta que es una o-exclusiva. Es decir, de entre los elementos incluidos sólo será válido que aparezca uno de ellos.

Asumiendo que dentro de una etiqueta <personal> sólo puede aparecer otra, a elegir entre <docente> o <laboral>, el XSD quedaría:

```
<xs:element name="personal">
  <xs:complexType>
    <xs:choice>
      <xs:element name="docente" />
      <xs:element name="laboral" />
    </xs:choice>
  </xs:complexType>
</xs:element>
```

4.3.5. Asociación con documentos XML

Vamos a ver un ejemplo práctico, donde un fichero “test.xsd” se encargará de validar a otro “test.xml”. Para ellos vamos a usar Notepad++

En primer lugar, debemos asegurarnos de tener en Notepad++ el plugin “XML Tools”. De estarlo, veremos la opción en el menú principal “Plugins” > “XML Tools”.

Si no lo está, podemos instalarlo en el menú principal “Plugins”, yendo a “Plugin manager” y ejecutando “Show Plugin Manager”. En la lista de la pestaña “Available” debe aparecer por orden alfabético (casi al final). Lo marcamos y aceptamos su instalación.

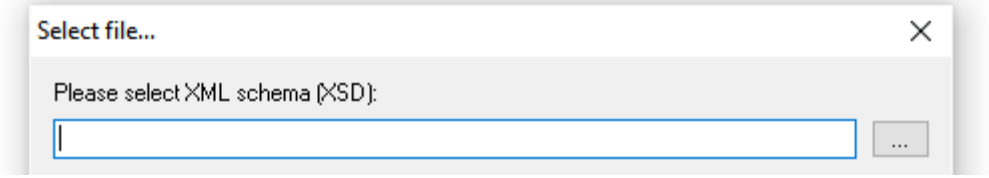
En segundo lugar, creamos el archivo XSD, con el siguiente contenido, guardándolo en una ubicación a nuestra conveniencia, con el nombre “test.xsd”:

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3      <xs:element name="carta" type="xs:string" />
4  </xs:schema>
```

Ahora creamos el documento a validar. Lo llamaremos “test.xml” y tendrá este contenido:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <carta>
3    Hola
4  </carta>
```

Podemos invocar la comprobación de su validación mediante el menú “Plugins” > “XML Tools” > “Validate now”, o bien el atajo de teclado Ctrl+Alt+May+M. Aparecerá la ventana:



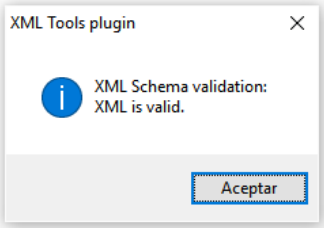
Esto es así porque aún no hemos vinculado este archivo con su XSD. Para hay que declarar el siguiente espacio de nombres:

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

Y el atributo “schemaLocation” de dicho espacio de nombres deberá contener la URI del archivo XSD validador. El resultado quedaría:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <carta xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3        xsi:schemaLocation="test.xsd">
4    Hola
5  </carta>
```

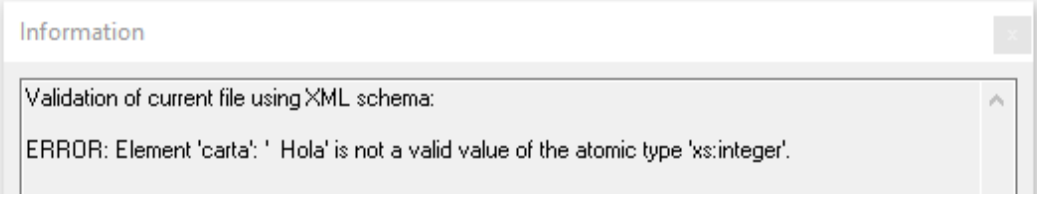
Si volvemos a solicitar su validación (y asumiendo que ambos archivos están en la misma carpeta) el resultado debería ser afirmativo:



Si, por ejemplo, cambiamos la definición del tipo de dato para <*carta*> a *integer*…

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

…entonces la validación fallará ya que el contenido del elemento es un texto no interpretable como número entero:



Que traduciendo del inglés, viene a decir que “Hola” no es un valor válido del tipo “xs:integer”.