UNIDAD 5

XML. Tratamiento y Recuperación de Datos

Curso: LMSXI – DAW 1

Centro: CIFP Rodolfo Ucha Piñeiro

Docente:

v 1.0 – marzo 2021

Tabla de contenido

1. Introducción

El objetivo del tema es tratar cómo se almacena y se recupera información de archivos XML igual que si fuesen bases de datos.

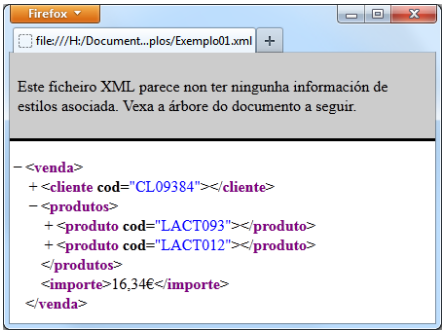
Por un lado existen bases de datos XML nativas, que son una alternativa a las relacionales en las que las tablas son sustituidas por documentos XML, incluso existen bases de datos relacionales (por ejemplo Oracle) que contemplan tipos de datos específicos para incorporar como campos estructuras o datos XML. Una de las más conocidas es BaseX.

Por otro lado tenemos tecnologías como lenguajes de consulta estandarizados por el W3C **XPath,** y un superconjunto de este que es XQuery.

Antes de hablar de estos recursos necesitamos revisar el concepto de parser o analizador de sintaxis de un documento XML.

2. Procesamiento de documentos XML

Los archivos XML se analizan e interpretan por medio de una herramienta llamada Parser (Analizador de sintaxis), cuyo objetivo es leer y estructurar los elementos del documento. Para ello interpreta su jerarquía y aplica el formato establecido para cada dato. Los parsers nos ayudan a obtener la información de un documento XML de forma automatizada.



Cada lenguaje de programación implementa sus propias librerías para parsear documentos XML (XmlDocument, Saxon, Xerces, ElementTree, etc). Los navegadores también disponen de sus propios parsers que les permiten convertir el contenido de un archivo XML a texto con formato HTML.

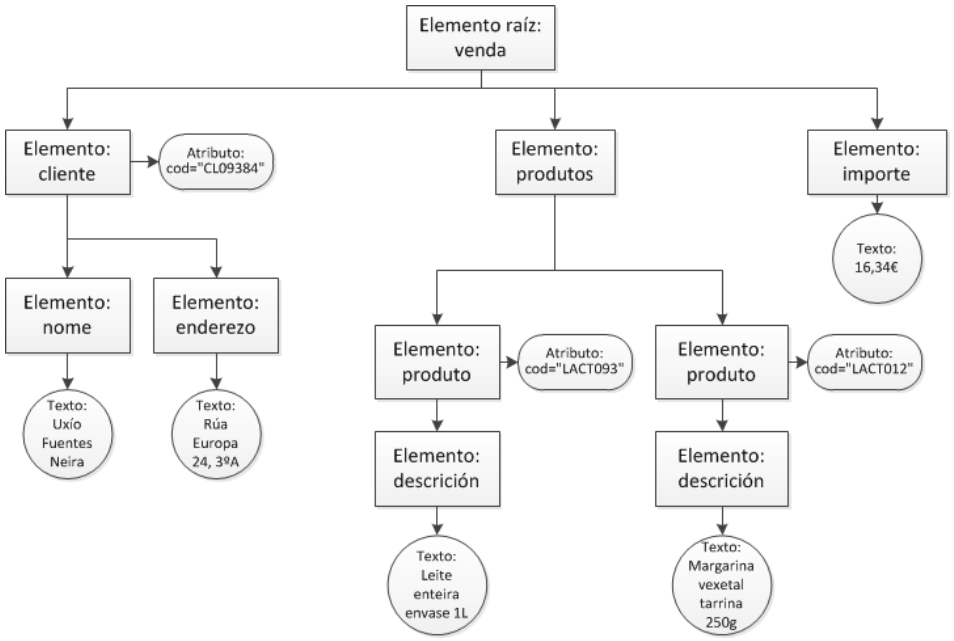
2.1 DOM

El modelo más empleado para almacenar y procesar los árboles XML es DOM (Document Objetc Model, Modelo de Objetos del Documento). DOM es una API de programación para documentos estándar del W3C (World Wide Web Consortium) que también se emplea en el procesamiento de documentos HTML. Es importante destacar que DOM no es un parser, sino que es una interfaz de programación que define los métodos y propiedades que los parsers tendrán que emplear para acceder al contenido del documento.

Por ejemplo, el siguiente documento XML:

```
<?xml version="1.0" encoding="utf-8"?>
<venta>
  <cliente cod="CL09384">
    <nombre>Uxío Fuentes Neira</nombre>
    <enderezo>Rúa Europa 24, 3ºA</enderezo>
  </cliente>
  <productos>
    <producto cod="LACT093">
      <descripción>Leite enteira envase 1L</descripción>
    </producto>
    <producto cod="LACT012">
      <descripción>Margarina vexetal tarrina 250g</descripción>
    </producto>
  </productos>
  <importe>16,34€</importe>
</venta>
```

Se estructuraría usando el modelo DOM XML en el siguiente árbol:



Todos los componentes de un documento XML son nodos. Parar crear el árbol se debe tener en cuenta el orden en que aparecen los elementos dentro del documento XML; por ejemplo, el producto de código "LACT093" es anterior al de código "LACT012". El orden de los atributos de un elemento, sin embargo, no es relevante.

Los nodos del árbol pueden ser de distintos tipos:

- **Nodo raíz o nodo documento (denominado /):** Representa el documento XML entero. Del nodo raíz cuelga un único hijo, el elemento raíz, que corresponde a la primera etiqueta del documento (que necesariamente por la especificación de XML abarca todo el documento).

- **Nodos elemento:** los elementos del documento son representados por nodos elemento dentro del árbol (uno de los cuales es el propio elemento raíz). Todos los nodos elemento tienen su nodo elemento padre (excepto el raíz) y pueden tener hijos. Los nodos elemento pueden tener identificadores únicos (se puede especificar en el documento de validación) para acceder a ellos de manera directa.

- **Nodos texto:** todos los caracteres que están entre las etiquetas de apertura y cierre de cada elemento. Estos nodos no pueden tener hijos, y su nodo anterior y su nodo posterior nunca serán de texto.

- **Nodos atributo:** aunque se les denomina nodos, en la estructura de árbol no son hijos de los elementos a los que califican, sino que es una información añadida a ellos.

- **Otros tipos de nodos:** comentarios *(contienen el texto de los <!-- comentarios --> del documento)*, de procesamiento de instrucciones *(<? ?>)* y de espacios de nombres *(xmlns:).*

Por ejemplo, el árbol del siguiente documento constaría del nodo raíz del que colgaría el elemento raíz (libro). De este colgarían cinco nodos elemento (los capítulos), de cada capítulo colgarían los nodos elemento de los párrafos ( ) y de cada párrafo colgarían los nodos de tipo texto (con el contenido textual que se encuentra encerrado entre las etiquetas y):

```
<?xml version="1.0" encoding="utf-8"?>
<libro>
      <capitulo titulo="Introducción a XML" numPáginas="4">
                  <p>...</p>
      </capitulo>
      <capitulo titulo="Validación con DTDs" numPáginas="6">
                  <p>...</p>
                  <p estilo="código">...</p>
      </capitulo>
      <capitulo titulo="Validación con XML Schemas" numPáginas="17">
                  <p>...</p>
                  <p>...</p>
                  <p estilo="resaltado">...</p>
      </capitulo>
      <capitulo titulo="XPath" numPáginas="7">
                  <p>...</p>
                  <p>...</p>
      </capitulo>
      <capitulo titulo="Transformacións XSLT" numPáginas="15">
                  <p>...</p>
                  <p estilo="código">...</p>
                  <p>...</p>
                  <p>...</p>
      </capitulo>
</libro>
```

Entre los nodos del árbol se establecen las siguientes relaciones y restricciones:

Solamente existe un nodo raíz, de tipo "Elemento".

Un nodo "Elemento" puede tener o no nodos hijos. Un nodo "Elemento" sin hijos es un nodo hoja. Los nodos que no son elemento no pueden tener hijos.

Todos los nodos a excepción del elemento raíz tienen uno y solamente un nodo padre. Tampoco tienen padre los nodos de tipo "Atributo", ya que cuelgan del nodo "Elemento" que los contiene, pero no se consideran hijos de este.

Se llaman nodos hermanos a todos los nodos "Elemento" que tienen el mismo padre.

El estándar DOM es una API de programación, por lo que además de estructurar la información en un árbol, define las propiedades y los métodos necesarios para acceder a ellos (para obtener su contenido, modificarlos, eliminarlos o añadir nuevos nodos):

Propiedades:

- **nodeName:** para obtener el nombre de un nodo.
- **nodeType:** tipo de nodo.
- **nodeValue:** texto que contienen los nodos de texto.
- **parentNode:** para obtener el padre de un nodo.
- **childNodes:** lista de hijos.
- **firstChild y lastChild:** primer y último hijos.
- **nextSibling y previousSibling:** hermanos siguiente y anterior respectivamente (null si no los hay).
- **attributes:** lista con los atributos de un nodo.

Métodos:

- **getElementsByTagName(nombre),** obtiene los elementos con una etiqueta determinada.
- **hasAttributes(),** indica si un elemento tiene o no atributos.
- **removeChild(nodo),** elimina el hijo que se indique de un nodo.

Si quisiésemos eliminar de un documento el nodo correspondiente a la dirección del cliente podríamos hacer:

```
direccion=doc.getElementsByTagName("direccion")[0];
cliente=direccion.parentNode;
cliente.removeChild(direccion);
```

Las expresiones que emplean DOM para recorrer y localizar los nodos de un árbol XML son en muchos casos complejas y muy largas. **El lenguaje XPath ofrece una forma más sencilla y efectiva de hacer esta tarea**.

» DOM Living Standard https://dom.spec.whatwg.org/

2.2 SAX y otras API's para lenguajes de programación.

Uno de los principales problemas de DOM es que antes de poder hacer cualquier cosa con el documento, es necesario parsearlo por completo y cargarlo en memoria. Si hubiese cualquier error en el documento o la cantidad de memoria del equipo fuese menor al tamaño del documento, no sería posible crear el árbol DOM y por tanto no podríamos acceder a ninguna información del archivo.

Al igual que DOM, SAX (Simple API for XML) es una API de programación, creada inicialmente para Java pero disponible actualmente en casi todos los lenguajes, que se encarga de procesar o analizar la información del documento XML con un enfoque diferente a como lo hace DOM: en lugar de crear un árbol en memoria, trabaja con eventos.

Para ello, SAX lee el documento secuencialmente de principio a fin, sin cargarlo en memoria, de forma que cuando encuentra un elemento se encarga de lanzar su evento asociado. Cuando el evento es lanzado éste se procesa por medio de un manejador de eventos que se encarga de realizar la función que corresponda.

Estas características hacen que SAX sea apropiado para la lectura y validación de documentos en los que sólo sea necesario procesarlos una sola vez, ya que SAX no crea un modelo y lo mantiene en memoria para su posterior uso. También es ideal en el caso de tener archivos de gran tamaño ya que lee el documento sin ocupación de memoria.

Otra de las APIS de Java es JAXP que aprovecha los estándares de los analizadores simples SAX y DOM.

En Java también podemos emplear una API más moderna y que se distribuye en paquetes independientes denominada JAXB (Java Architecture for XML Binding).

» Java Architecture for XML Binding (JAXB) https://www.oracle.com/technical-resources/articles/javase/jaxb.html

» Java API for XML Processing (JAXP) https://www.oracle.com/java/technologies/jaxp-introduction.html

» SAX http://www.saxproject.org/

3. BaseX: Base de Datos XML nativa

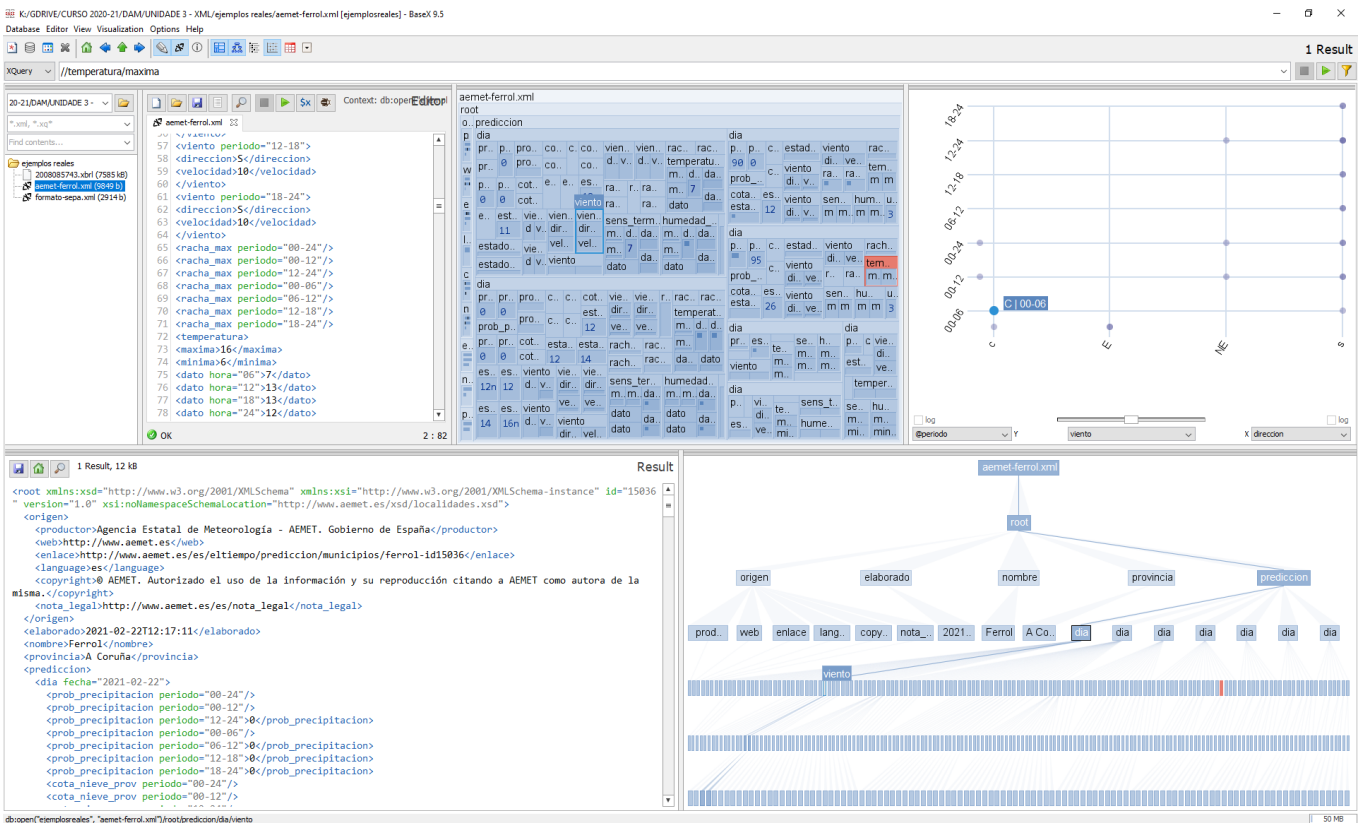
BaseX es un motor de bases de datos nativo XML, ligero, de alto rendimiento en las operaciones y fácilmente escalable. Incluye, así mismo, procesadores de XPath y XQuery.

Permite crear una base de datos constituida por uno o más documentos XML. En estos sistemas no encontramos tablas encontramos documentos XML. Empleando los lenguajes de proceso mencionados podemos acceder mediante consultas al contenido de dichos documentos.

Una de las aportaciones de XBase es que el contenido de los documentos XML se puede visualizar de distintas maneras mediante distintas vistas sincronizadas entre si:

- Texto Plano
- Mapa
- Árbol
- Simulación de Carpetas y Archivos
- Tabla de Datos
- Diagrama de dispersión

+



Ejemplo de interfaz gráfica de BaseX

En la parte superios vemos un desplegable que permite elegir entre:
Command: Permite emplear comandos de gestión propios de las bases de datos (CREATE DB, OPEN, CREATE INDEX, CREATE USER, ADD, DELETE, REPLACE…)
Find: Permite buscar información mediante XPath
XQuery: Permite emplear el lenguaje de consulta XQuery

» Web oficial BaseX https://basex.org/

» Pequeño tutorial ejemplo BaseX https://www.informaticcity.com/2017/08/como-usar-basex.html

4. XPath

XPath (XML Path Language) es un lenguaje declarativo (especificación del W3C) que define la construcción de expresiones que nos permitan acceder a ciertas partes del documento XML. No está definido en XML, sino que tiene una sintaxis propia.

XPath se utiliza en la mayoría de las tecnologías existentes en el mundo XML, como pueden ser XSLT, XSL-FO, XPointer, XLink, XQuery, etc. puesto que todas ellas necesitan seleccionar partes concretas del documento XML, y esa selección se hace por medio de XPath.

XPath analiza el documento XML creando una estructura de árbol como la del modelo DOM, donde los nodos XML representan las distintas partes del documento. La forma de seleccionar información se basa en dicha estructura de árbol, pues crearemos expresiones XPath que se evaluarán sobre él produciendo un resultado.

» Estándares y borradores XPath https://www.w3.org/TR/xpath/all/
» Para probar XPath podemos emplear entornos como BaseX o XMLCopyEditor
» Analizadores de expresiones en línea (mejor para funciones avanzadas):
https://www.freeformatter.com/xpath-tester.html
https://codebeautify.org/Xpath-Tester

En algunos ejemplos nos basaremos en el siguiente archivo (comerciales.xml):

```
<empresa xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="comerciales.xsd">

<comercial>
    <nombre>Alberto </nombre>
    <apellidos>Abalde Sanromán</apellidos>
    <telefono>666 444 444</telefono>
    <mail>albertoas@empresa.com</mail>
    <coche>2222KDL</coche>
    <salario>1900</salario>
</comercial>
<comercial>
    <nombre>Juan </nombre>
    <apellidos>Sola Iturralde</apellidos>
    <telefono>666 444 555</telefono>
    <mail>juansi@empresa.com</mail>
    <coche>8888LFH</coche>
    <salario>1875</salario>
</comercial>
<comercial>
    <nombre>Lucia</nombre>
    <apellidos>Castro González</apellidos>
    <telefono>666 444 666</telefono>
    <mail>luciacg@empresa.com</mail>
    <coche>4444LLL</coche>
    <salario>1700</salario>
</comercial>

<vehiculo marca="FORD" matricula="2222KDL" fechac="10-01-2017">
        <nombre>Transit Courier EcoBoost 100cv</nombre>
</vehiculo>
<vehiculo marca="CITROEN" matricula="7777HJK" fechac="10-05-2017">
    <nombre>Berlingo BLUEHDI 100 </nombre>
</vehiculo>
<vehiculo marca="FORD" matricula="1144JZX" fechac="11-06-2018">
      <nombre>Transit Courier EcoBoost 100cv</nombre>
</vehiculo>
<vehiculo marca="FORD" matricula="4444LLL" fechac="11-06-2018">
      <nombre>Transit Connect  1,5 TDCi EcoBlue75cv</nombre>
</vehiculo>
<vehiculo marca="CITROEN" matricula="8888LFH" fechac="01-05-2020">
      <nombre>Jumpy Combi BLUEHDI 100 S</nombre>
</vehiculo>
</empresa>
```

4.1 Expresiones XPath

Las rutas de localización *(location paths)* son el tipo de expresión más importante en XPath. Sirven para describir la localización de los datos que nos interesan en un documento XML.

Una ruta de localización consiste en uno o más pasos de localización *(location steps),* cada uno de los cuales va refinando la búsqueda a través de los nodos del árbol.

> Las expresiones XPath construidas a través de las palabras clave del lenguaje, los símbolos y los operadores existentes, nos devolverán resultados que pueden ser escalares (booleanos, numéricos o cadenas) y lo que se denomina *node-set* (conjunto de nodos). **Es decir, el resultado no tiene por qué ser un documento XML.**

Existe un cierto paralelismo entre el uso de expresiones XPath con un documento XML y el uso de rutas en un sistema de archivos (por ejemplo Linux):

- Ambos separan los elementos (archivos o nodos) con el carácter "/".
- Los nodos de un archivo XML se pueden anidar unos dentro de otros, igual que los archivos y las carpetas
- Existen ciertos símbolos para seleccionar simultáneamente diversos archivos o diversos nodos (*, por ejemplo)
- Igual que existe el concepto de carpeta actual, existe también el concepto de nodo contexto (nodo sobre el que se evaluará la próxima expresión).
- Las rutas (*location paths en el caso de XPath)* pueden ser también absolutas o relativas. Si la ruta de localización **comienza con un / significa que empezamos la búsqueda por el nodo raíz del documento**

Los pasos de localización *(location steps)* son cada uno de los elementos de una ruta de localización. Constan de un eje *(axis),* un test de nodo *(node test)* y opcionalmente de un predicado. La sintaxis es la siguiente:

```
eje::test-nodo[predicado]
```

4.1.1 Rutas de localización – *(location paths)*

Como se ha introducido al principio son las expresiones que permiten seleccionar un nodo o un conjunto de nodos, indicando cual es su localización exacta (relativa o absoluta).

En XMLCopyEditor, al evaluador XPath(F9) para indicarle una ruta absoluta es necesario comenzar con / y para indicarle una ruta relativa con //

Esta expresión nos devolverá todos los nodos nombre de los comerciales

```
/empresa/comercial/nombre
```

```
<nombre>Alberto </nombre>
<nombre>Juan </nombre>
<nombre>Lucia</nombre>
```

Esta otra expresión nos devolverá todos los nodos que se llamen nombre (comerciales o vehículos)

```
//nombre
```

```
<nombre>Alberto </nombre>
<nombre>Juan </nombre>
<nombre>Lucia</nombre>
<nombre>Transit Courier EcoBoost 100cv</nombre>
<nombre>Berlingo BLUEHDI 100 </nombre>
<nombre>Transit Courier EcoBoost 100cv</nombre>
<nombre>Transit Connect  1,5 TDCi
EcoBlue75cv</nombre>
<nombre>Jumpy Combi BLUEHDI 100 S</nombre>
```

Como se ve en el ejemplo estas rutas relativas (//) también pueden emplearse a partir de un nodo dado para que encuentre todos los descendientes concretos (hijos,hermano,nietos…)

La siguiente tabla muestra las expresiones Xpath más comunes:

Expresión	Resultado esperado	Sobre el ejemplo
elemento	Elemento de nombre *elemento*	*comercial* Devuelve todos los elementos de comercial
/elemento	Devuelve el *elemento* ubicado en la raíz del documento	*/empresa* Devuelve todo el elemento empresa con sus descendientes
e1/e2	Elementos *e2* que sean hijos directos de *e1*	*comercial/apellidos* Devuelve los elementos apellidos con sus valores.
e1//e2	Elementos descendientes de *e1 (hijos, nietos, bisnietos…)* que se llamen *e2*	*empresa//nombre* Devuelve todos los elementos que encuentren llamados nombre en todos los niveles a partir de empresa.
//elemento	Elemento de nombre *elemento* ubicado en cualquier nivel debajo la raíz del documento	*//telefono* Debuelve todos los elementos llamado teléfonos de cualquier nivel.
@atributo	El atributo y su valor de nombre *atributo*	*@matricula* Devuelve todas las matrículas con su nombre y valor.
*****	Todos los elementos	
@*	Todos los atributos	
.	Nodo actual	
..	Nodo Padre	
espNom:*		pr:* (cuando existan espacios de nombres)
@espNom:*	Todos los atributos en el espacio de nombres de prefijo *espNom*	@pr:* (cuando existan espacios de nombres)

4.1.2 Acceso textual a elementos y atributos

Por norma general, al emplear un analizador y buscar introducir expresiones de rutas de localización para elementos o atributos, si no encuentra la información devolverá un conjunto vacío o si la encuentra nos devolverá un conjunto normalmente formado por los elementos completos solicitados o bien por los atributos con sus valores.

- **Contenido textual de los nodos:** Si lo que se desea es indicar al analizador que se devuelva el contenido textual del nodo o nodos accedidos se emplea la función `text()`

```
/empresa/comercial/nombre/text()
```

```
Alberto
Juan
Lucia
```

- **Contenido textual (valor) del atributo:** La función empleada es `data()`.

```
//@matricula/data()
```

```
Alberto
Juan
Lucia
```

A partir de XPath 2.0 se puede emplear también `string()` para devolver la información textual de nodos (elementos) y atributos.

4.1.3 Sintaxis abreviada y compleja

Existen dos sintaxis diferentes a la hora de escribir consultas XPath:

- **Sintaxis abreviada:** más sencilla y fácil de leer. Todos los ejemplos anteriores corresponden a esta sintaxis.
- **Sintaxis compleja:** más larga y compleja de leer. Utiliza los ejes para nombrar los elementos.

Los ejes son expresiones que permiten acceder a trozos del árbol XML apoyándose en las relaciones de parentesco entre los nodos. En las siguientes definiciones, el nodo de contexto se refiere al nodo al que se le está aplicando el eje.

Expresión	Resultado esperado
self::	Devuelve el propio nodo de contexto. Equivale a .
child::	Devuelve los nodos hijo del nodo de contexto.
parent::	Devuelve el nodo padre del nodo de contexto. Equivale a ..
ancestor::	Devuelve los nodos antepasados (padre, abuelo, ...) del nodo de contexto.
ancestor-or-self::	Devuelve los nodos anteasados (padre, abuelo, ...) además del propio nodo de contexto.
descendant::	Devuelve los nodos descendientes (hiejo, nieto, ...) del nodo de contexto.
descendant-or-self::	Devuelve los nodos descendientes (hijo, nieto, ...) además del propio nodo de contexto. Equivale a //
following::	Devuelve los nodos que aparezcan después del nodo de contexto en el documento, excluyendo a los nodos descendientes, los atributos y los nodos de espacio de nombres.
preceding::	Devuelve los nodos que aparezcan antes del nodo de contexto en el documento, excluyendo a los nodos ascendientes, los atributos y los nodos de espacio de nombres.
following-sibling::	Devuelve los hermanos menores del nodo de contexto.
preceding-sibling::	Devuelve los hermanos mayores del nodo de contexto.
attribute::	Atributos del nodo de contexto.
namespace::	Espacio de nombres del nodo de contexto.

4.2 Filtrar el acceso a elementos

El direccionamiento de XPath también permite filtrar el conjunto de nodos o información a la que se accede mediante la consulta utilizando condiciones en nodos. **El filtro se especifica mediante corchetes [ ]** seguidos del nodo al que se le aplica dicho filtro.

```
//vehiculo[@marca="CITROEN"]
```

```
<vehiculo marca="CITROEN" matricula="7777HJK" fechac="10-05-2017">
  <nombre>Berlingo BLUEHDI 100</nombre>
</vehiculo>
<vehiculo marca="CITROEN" matricula="8888LFH" fechac="01-05-2020">
  <nombre>Jumpy Combi BLUEHDI 100 S</nombre>
</vehiculo>
```

- **Operadores**

and	
or	
not	
=	
!=	
<	
>	
<=	
>=	
to	Rango
+	Suma
-	Resta
*****	Multiplicación
div	División
mod	Resto de la división
|	Unión de resultados

- **Funciones numéricas**

round()	Redondeo	round(6.48) = 6
abs()	Valor absoluto	abs(-6) = 6
floor()	Redondeo inferior	floor(6.2) = 6
ceiling	Redondeo superior	ceiling(6.2) = 7

- **Funciones de cadena**

substring()	Subcadena	substring('Berlingo', 1, 5) = Berli
starts-with()	Cadena comienza por	starts-with('Berlingo', 'B') = true
ends-with()	Cadena finaliza por	ends-with('Berlingo', 'B') = false
contains()	Cadena contiene	contains('Berlingo', 'li') = true
normalize-space()	Espacios normalizados	normalize-space(' Transit Courier ') = 'Transit Courier'
translate()	Reemplaza caracteres en una cadena	translate('Transit Courier', 'Transit', 'Scouter') = 'Scouter Courier'
string-length()	Longitud de una cadena	string-length('Berlingo') = 8
upper-case()	Cadena a mayúsculas	upper-case('berlingo') = 'BERLINGO'
lower-case()	Cadena a minúsculas	lower-case('BERLINGO') = 'berlingo'

- **Funciones de posición de elementos**

position() = n	Nodo que se encuentra en la posición 'n'	//nombre[position]=2 → <nombre>Juan</nombre>
elemento[n]	Nodo en la posición 'n' de los que se llaman nodo	//nombre[3] → <nombre>Lucia</nombre>
last()	El último nodo de un conjunto	//mail[last()] → <mail>luciacg@empresa.com</mail>
last() - i	El último menos i nodos	//nombre[last()-1] → <mail>juansi@empresa.com</mail>

- **Funciones de nodos**

name()	Nombre del nodo actual	
text()	Contenido textual del nodo	
root()	Elemento raíz	
node()	Nodos descendientes del actual	
comment()	Comentarios del nodo	
processing-instruction()	Instrucciones de procesamiento	
exists()	Si existe el nodo o no	
empty()	Si el nodo está vacío o no	

- **Funciones de agregado**

count()	Contar los nodos	count(//comercial) → 3
avg()	Media del contenido de los nodos	avg(//salario) → 1825
max()	Valor máximo del contenido de los nodos	max(//salario) → 1900
min()	Valor mínimo del contenido nodos	min(//salario) → 1700
sum()	Suma del contenido de los nodos	sum(//salario) → 5475

» Listado completo de Funciones https://www.w3.org/TR/xpath-functions-30/

4.3 Ejemplos

Mostrar mail y teléfono comerciales. Utilizaremos el operador unión para unir los dos conjuntos de nodos:

```
//mail|//telefono
```

```
<telefono>666 444 444</telefono>
<mail>albertoas@empresa.com</mail>
<telefono>666 444 555</telefono>
<mail>juansi@empresa.com</mail>
<telefono>666 444 666</telefono>
<mail>luciacg@empresa.com</mail>
```

Mostrar vehículos del número 2 al número 3:

```
//vehiculo[position()=2 to 3]
```

```
<vehiculo fechac="10-05-2017" marca="CITROEN" matricula="7777HJK">
    <nombre>Berlingo BLUEHDI 100 </nombre>
</vehiculo>
<vehiculo fechac="11-06-2018" marca="FORD" matricula="1144JZX">
      <nombre>Transit Courier EcoBoost 100cv</nombre>
 </vehiculo>
```

Mostrar los comerciales que ganan menos o igual de 1875€. *Los números no necesitan comillas.*

```
//comercial[salario<=1875]
```

```
<comercial>
    <nombre>Juan </nombre>
      …
    <salario>1875</salario>
</comercial>
<comercial>
    <nombre>Lucia</nombre>
     …
    <salario>1700</salario>
</comercial>
```

Mostrar las matrículas de los vehículos que son FORD

```
//vehiculo[@marca="FORD"]/@matricula
```

```
matricula=2222KDL
matricula=1144JZX
matricula=4444LLL
```

4.4 Consultas XPath anidadas

Las consultas XPath anidadas consisten en incluir una consulta XPath que devuelta un cierto valor dentro de la condición de otra consulta XPath.

Imaginemos que se nos pide el siguiente enunciado:

Encontrar todos los nombres de los comerciales que conducen un FORD.

1) Debemos encontrar que matrículas corresponden a los vehículos FORD

```
//vehiculo[@marca="FORD"]/@matricula
```

```
matricula=2222KDL
matricula=1144JZX
matricula=4444LLL
```

2) Ahora debemos encontrar los comerciales cuyo coche coincida con las del conjunto anterior.

```
//comercial[coche=consulta]/nombre
```

```
<nombre>Alberto </nombre>
<nombre>Lucia</nombre>
```

Sustituiremos consulta por la anterior

```
//comercial[coche=//vehiculo[@marca="FORD"]/@matricula]/nombre
```

5. XQuery

XQuery es un lenguaje de consulta que permite extraer información de bases de datos o documentos XML. Se puede decir que XQuery es a XML lo mismo que SQL a las bases de datos relacionales.

XQuery se basa en el lenguaje XPath para el acceso a los nodos XML, pudiendo utilizar todos sus operadores y funciones, de ahí la consideración de XQuery como un superconjunto de XPath.

» Recomendación W3C https://www.w3.org/TR/xquery-30/

El lenguaje XQuery es muy amplio y complejo, así que para el curso nos centraremos en la base de su funcionamiento sin entrar en funciones excesivamente avanzadas y complejas.

5.1 Consultas FLWOR

Las consultas XQuery se componen de cinco cláusulas, que debido a sus iniciales se las conoce como FLWOR. Definimos cada una de ellas:

- **FOR**: Indica qué nodos se van a seleccionar desde la base de datos XML o desde un documento XML.
- **LET**: Permite declarar variables a las que se le asignan valores.
- **WHERE**: Permite introducir condiciones que deben cumplir los nodos seleccionados por la cláusula "for".
- **ORDER BY**: Permite ordenar los nodos antes de su visualización.
- **RETURN**: Devuelve los resultados. Es la única cláusula obligatoria.

Aunque también se añade a éstas la siguiente cláusula:

- **GROUP BY:** Permite agrupar por nodos similares.

Cláusula "for" y "return"

Con la cláusula "for" recuperaremos una serie de nodos mediante una consulta XPath y los introduciremos en una variable para poder utilizarla en la cláusula "return". Hay que señalar que la cláusula "return" se ejecutará una vez por cada nodo que devuelva la cláusula "for".

```
for $comercial in /empresa/comercial
return $comercial/nombre
```

```
<nombre>Alberto</nombre>
<nombre>Juan</nombre>
<nombre>Lucia</nombre>
```

Como no hemos indicado ningún documento tras "in" la consulta se lanzará contra la base de datos que tengamos abierta en nuestro programa. El resto de ejemplos se realizarán de esta manera, pero si queremos lanzar la consulta contra **un documento XML que no es una base de datos podemos hacerlo usando "doc":**

```
for $comercial in doc("comerciales.xml")/empresa/comercial
return $comercial/nombre
```

Si queremos imprimir **nuestras propias etiquetas** en la cláusula "return", tendremos que encerrar la variable **entre llaves { }:**

```
for $comercial in /empresa/comercial

return <comercial>{$comercial/nombre/text()," ",$comercial/apellidos/text()}</comercial>
```

```
<comercial>Alberto Abalde Sanromán</comercial>

<comercial>Juan Sola Iturralde</comercial>

<comercial>Lucia Castro González</comercial>
```

Podemos **utilizar "at" dentro de la cláusula "for**" para obtener **una variable con la numeración** de los nodos que se van a recorrer:

```
for $comercial at $i in /empresa/comercial
return <comercial>{$i} - {$comercial/nombre/text()," ",$comercial/apellidos/text()}</comercial>
```

```
<comercial>1 - Alberto Abalde Sanromán</comercial>
<comercial>2 - Juan Sola Iturralde</comercial>
<comercial>3 - Lucia Castro González</comercial>
```

Si quisiéramos englobar **todas las etiquetas anteriores en una superior**, tendríamos que **encerrar la consulta completa entre llaves { }**.

```
<comerciales>
 {
  for $comercial at $i in /empresa/comercial
  return <comercial>{$i} - {$comercial/nombre/text()," ",$comercial/apellidos/text()}</comercial>
 }
</comerciales>
```

```
<comerciales>
 <comercial>1 - Alberto Abalde Sanromán</comercial>
 <comercial>2 - Juan Sola Iturralde</comercial>
 <comercial>3 - Lucia Castro González</comercial>
</comerciales>
```

También podemos **utilizar la estrucutra condicional "if"** dentro de la cláusula "return" para **modificar el resultado en función de alguna condición:**

```
<comerciales>
{
for $comercial at $i in /empresa/comercial
return <comercial>
         {$i} - {$comercial/nombre/text()," ",$comercial/apellidos/text()}
         { if ($comercial/salario>1800) then (
             <alto/>
         ) else (
             <bajo/>
         )}
     </comercial>
}
</comerciales>
```

```
<comerciales>
  <comercial>1 - Alberto Abalde Sanromán<alto/>
  </comercial>
  <comercial>2 - Juan Sola Iturralde<alto/>
  </comercial>
  <comercial>3 - Lucia Castro González<bajo/>
  </comercial>
</comerciales>
```

Cláusula "let"

La cláusula "let" nos va a permitir crear variables con cierto contenido. La diferencia con "for" es que ésta sólo se ejecutaría una sola vez con la cláusula "return". La cláusula "let" asigna las variables mediante los caracteres ":=".

```
let $comerciales:=/empresa/comercial
return <comerciales> {$comerciales/nombre}</comerciales>
```

```
<comerciales>
  <nombre>Alberto</nombre>
  <nombre>Juan</nombre>
  <nombre>Lucia</nombre>
</comerciales>
```

Podemos observar como la etiqueta "comerciales" sólo aparece una vez, es decir, no se repite para cada nodo como en el caso de la cláusula "for".

La clasula "let" nos va a permitir utilizar funciones de agrupación, como calcular la media, la suma, contar, etc. Estas son las mismas funciones que las que se utilizan en el leguaje XPath. Podemos por ejemplo buscar el salario más alto que exista mediante la función "max" para ver el comercial con más salario:

```
let $comerciales:=/empresa/comercial
return <salario_máximo> {max($comerciales/salario)} </salario_máximo>
```

```
<salario_máximo>1900</salario_máximo>
```

También podemos crear varias variables seguidas con coma, por ejemplo para buscar el salario más alto y el más bajo:

```
let $maximo:=max(/empresa/comercial/salario), $minimo:=min(/empresa/comercial/salario)
return <salarios>
       <máximo> {$maximo }</máximo>
       <mínimo> {$minimo} </mínimo>
       </salarios>
```

```
<salarios>
  <máximo>1900</máximo>
  <mínimo>1700</mínimo>
</salarios>
```

La cláusula "let" también nos permite asignar el valor de una variable a partir de otro consulta:

```
let $comerciales := (
for $comercial in /empresa/comercial
return <comercial>{$comercial/nombre/text()," ",$comercial/apellidos/text()}</comercial>)
return $comerciales
```

```
<comercial>Alberto Abalde Sanromán</comercial>
<comercial>Juan Sola Iturralde</comercial>
<comercial>Lucia Castro González</comercial>
```

Cláusula "for" y "let"

Podemos combinar las cláusulas "for" y "let". De esta manera conseguimos que la clasula "let" se ejecute una vez por cada nodo, al igual que hace la clasula "return".

Cláusula "where"

Con la cláusula "where" podemos filtrar los nodos que se seleccionan en la cláusula "for", para ello también podemos utilizar los mismos operadores y funciones que en el leguaje XPath. MUY IMPORTANTE, la cláusula "where" NO filtraría los nodos si los estamos obteniedo con "let".

```
for $vehiculos in /empresa/vehiculo
where $vehiculos/@marca="FORD"
return $vehiculos/@matricula
```

```
matricula="2222KDL"
matricula="1144JZX"
matricula="4444LLL"
```

La misma consulta anterior se podría realizar de igual manera filtrando los nodos en la consulta XPath sin tener que utilizar la cláusula "where":

```
for $vehiculos in /empresa/vehiculo[@marca="FORD"]
return $vehiculos/@matricula
```

```
matricula="2222KDL"
matricula="1144JZX"
matricula="4444LLL"
```

Un ejemplo más con la cláusula "where" utilizando una función:

```
for $vehiculos in /empresa/vehiculo
where starts-with($vehiculos/nombre,'T')
return $vehiculos/nombre/text()
```

```
Transit Courier EcoBoost 100cv
Transit Courier EcoBoost 100cv
Transit Connect  1,5 TDCi EcoBlue75cv
```

Otro ejemplo más con la cláusula "where" en la que se unen varias condiciones:

```
for $vehiculos in /empresa/vehiculo
where starts-with($vehiculos/nombre,'B') or starts-with($vehiculos/nombre,'J')
return $vehiculos/nombre/text()
```

```
Berlingo BLUEHDI 100
Jumpy Combi BLUEHDI 100 S
```

Cláusula "order by"

Con la cláusula "order by" podemos ordenar los nodos antes de que empiece a ejecutarse la cláusula "return", ya que como sabemos, la salida será la misma que el orden que tengan los nodos en el documento o base de datos XML.

```
for $comerciales in /empresa/comercial
order by $comerciales/apellidos
return concat($comerciales/apellidos/text(),", ",$comerciales/nombre/text())
```

```
Abalde Sanromán, Alberto
Castro González, Lucia
Sola Iturralde, Juan
```

Podemos ordenar por diferentes campos separándolos por coma.

```
for $comerciales in /empresa/comercial
order by $comerciales/apellidos, $comerciales/nombre
return concat($comerciales/apellidos/text(),", ",$comerciales/nombre/text())
```

Podemos ordenar de manera descendente (descending), ya que por defecto se ordena de manera ascendente (ascending).

```
for $comerciales in /empresa/comercial
order by $comerciales/apellidos descending
return concat($comerciales/apellidos/text(),", ",$comerciales/nombre/text())
```

```
Sola Iturralde, Juan
Castro González, Lucia
Abalde Sanromán, Alberto
```

Cláusula "group by"

Con la cláusula "group by" podemos agrupar los nodos en función de un valor de nodo o de atributo:

```
for $vehiculos in /empresa/vehiculo
group by $marca:=$vehiculos/@marca
return <grupo marca="{$marca}" total="{count($vehiculos)}"/>
```

```
<grupo marca="FORD" total="3"/>
<grupo marca="CITROEN" total="2"/>
```

El operador except

El operador "except" nos permite eliminar nodos de la salida de la consulta, pero para ello es obligatorio utilizar "/*" en el nodo donde vayamos a utilizarlo como vemos en el ejemplo. En este caso también se devuelve la información sin la etiqueta de comercial de ahí que le hayamos incluido una en el ejemplo.

```
for $comerciales in /empresa/comercial
return <comercial>
         {$comerciales/* except $comerciales/telefono except $comerciales/mail }
       </comercial>
```

```
<comercial>
  <nombre>Alberto</nombre>
  <apellidos>Abalde Sanromán</apellidos>
  <coche>2222KDL</coche>
  <salario>1900</salario>
</comercial>
<comercial>
  <nombre>Juan</nombre>
  <apellidos>Sola Iturralde</apellidos>
  <coche>1144JZX</coche>
  <salario>1875</salario>
</comercial>
<comercial>
  <nombre>Lucia</nombre>
  <apellidos>Castro González</apellidos>
  <coche>4444LLL</coche>
  <salario>1700</salario>
</comercial>
```

Bibliografía y Recursos

- Fundamentos de XML Jorge Sanchez http://jorgesanchez.net/manuales/xml/fundamentos-xml.html
- Qué es XML (Rafael Morales) https://www.ticarte.com/contenido/que-es-xml
- XML: Lenguaje de marcas extensible (Bartolomé Sintes Marco) https://www.mclibre.org/consultar/xml/
- W3schools CSS: https://www.w3schools.com/xml/
- Tutorial de XML – Carlos Pes https://www.abrirllave.com/xml/
- Extensible Markup Language (XML) https://www.w3.org/XML/

Créditos y fuentes extra

- Temario LMSXI de Cristina Puga
- Material para a formación profesional inicial

  Módulo profesional MP0373 Linguaxes de marcas e sistemas de xestión de información

  Autores: Amalia Falcón Docampo, aura Fernández Nocelo e Marta Rey López
- Lenguajes de marcas y sistemas de gestión de información. J.M. Castro y J.R. Rodríguez. 2012.

  Ed. Ibergarceta