Módulo: Acceso a datos

UT6: Bases de datos XML

Código **0486**

Nivel **Ciclo Formativo de Grado Superior**
Ciclo **Desarrollo de Aplicaciones Multiplataforma**

Índice

Resultados de aprendizaje y criterios de evaluación de la UT6

¿QUÉ DEBEMOS APRENDER DURANTE LA UNIDAD DE TRABAJO Y QUÉ SE VA A EVALUAR?

RESULTADO DE APRENDIZAJE:

RA5. Desarrolla aplicaciones que gestionan la información almacenada en bases de datos documentales nativas evaluando y utilizando clases específicas.

CRITERIOS DE EVALUACIÓN:

a) Se han valorado las ventajas e inconvenientes de utilizar bases de datos documentales nativas.
b) Se ha establecido la conexión con la base de datos.
c) Se han desarrollado aplicaciones que efectúan consultas sobre el contenido de la base de datos.
d) Se han añadido y eliminado colecciones de la base de datos.
e) Se han desarrollado aplicaciones para añadir, modificar y eliminar documentos de la base de datos.

Glosario de términos

Base de datos nativa de XML: base de datos para documentos XML que, para su almacenamiento, utiliza estructuras diseñadas y optimizadas para este tipo de documento.

XML:DB : API para bases e datos XML que permite realizar todo tipo de operaciones.

XML: Lenguaje que permite representar cualquier tipo de información en una estructura jerárquica dentro de un documento de texto.

XPath: Lenguaje de consultas para documentos XML que permite recuperar un conjunto de nodos de su estructura jerárquica.

XQJ (XQuery for Java): API para bases de datos XML que permite realizar operaciones de consulta utilizando XQuery, y operaciones de modificación utilizando extensiones de XQuery.

XQuery: Lenguaje de consulta para documentos XML, más potente que XPath, y del que XPath es un subconjunto. Se han propuesto varias extensiones de XPath que permiten modificar documentos de XML. Entre ellas está XQUF, que es un estándar de W3C.

1 Introducción a las bases de datos nativas XML

1.1 Bases de datos NoSQL

Aunque hoy en día la mayoría de sistemas de gestión de bases de datos siguen basándose en el modelo relacional, cada vez van ganando más relevancia sistemas basados en otros paradigmas de base de datos, como las orientadas a objetos que vimos en unidades anteriores, o las basadas en el lenguaje XML, cuyas principales características han sido abordadas en gran parte por los SGBD relacionales.

Bajo el término de **NoSQL** se engloban todas aquellas alternativas a los sistemas tradicionales que no utilizan el modelo relacional como base, aunque también se le ha dado el significado de *not only SQL* en referencia a que algunos de estos sistemas sí utilizan SQL como lenguaje de consulta, pero no se basan en un modelo relacional.

Con la llegada de la web 2.0, el crecimiento de datos en Internet aumentó de forma exponencial de la mano, principalmente, de las redes sociales y del contenido multimedia (Facebook, Twitter, YouTube, etc.), plataformas donde son los propios usuarios quienes generan la mayor parte del contenido. Esto provocó un rápido crecimiento de la necesidad de procesar grandes volúmenes de información para lo que las bases de datos tradicionales no estaban preparadas. Con la web 3.0, 4.0 y el auge de la inteligencia artificial, Internet se ha convertido en una gran base de datos, no solo para proporcionar contenido a la web, sino para cualquier tipo de aplicación.

Las bases de datos NoSQL permiten asumir estos escenarios donde las bases de datos relacionales presentan problemas de **escalabilidad** y **rendimiento** cuando miles de usuarios acceden simultáneamente a la información.

No significa, sin embargo, que las bases de datos tradicionales vayan a ser reemplazadas por las NoSQL. Estamos en un panorama en el que es difícil encontrar una única tecnología apropiada para todos los escenarios. Por ello, existe hoy en día un amplio abanico de soluciones específicas para abordar diferentes problemáticas, todas ellas agrupadas bajo el movimiento NoSQL.

1.1.1 Tipos de bases de datos NoSQL

Dentro del panorama NoSQL, encontramos diversos tipos de bases de datos según su forma de almacenar los datos. A continuación, se destacan los tres más representativos:

A. Bases de datos clave-valor

- Se basan en un modelo sencillo y popular, donde cada elemento se identifica por una clave única (similar a tablas hash).
- El valor asociado a la clave suele ser un objeto binario (*BLOB*).
- Ejemplos de este tipo son **Cassandra** (Apache), **Bigtable** (Google) o **Dynamo** (Amazon).

B. Bases de datos documentales

- Almacenan la información en forma de documentos, típicamente en XML o JSON.

- Cada documento se asocia a una clave única, lo que permite búsquedas estilo clave-valor.
- A diferencia del tipo anterior, el "valor" es un documento completo, no un dato binario.
- El exponente más conocido de este tipo es **MongoDB** (con JSON) y, en el caso de formato **XML**, las propias bases de datos nativas XML.

C. Bases de datos en grafo

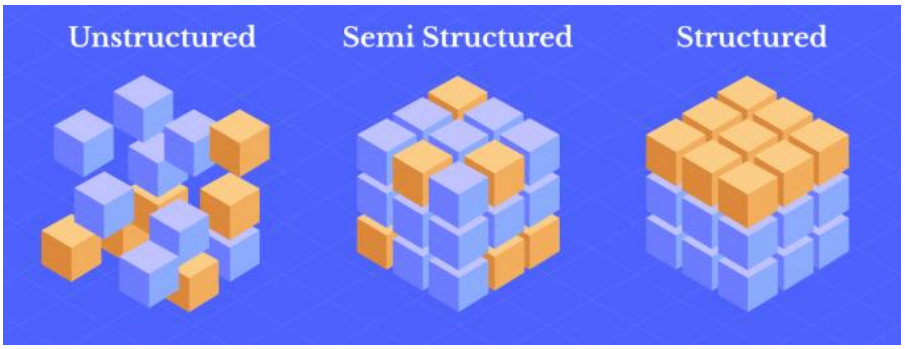
- Se basan en la representación de datos como nodos y aristas de un **grafo**, aprovechando la teoría de grafos para recorrer la información de forma óptima.
- Las relaciones entre datos se modelan como aristas entre nodos.
- Ejemplos de este tipo son **Amazon Neptune**, **JanusGraph** (Apache), **SQL Server** (Microsoft, con su extensión para grafos) o **Neo4j**.

1.2 Definición y características de las bases de datos documentales

Estructuración de los datos

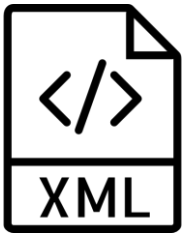
Antes de profundizar en las bases de datos XML, es importante entender que no todos los datos se presentan de la misma forma. Podemos distinguir:

- **Datos estructurados**: tienen un formato estricto y uniforme (por ejemplo, tablas con filas y columnas en bases de datos relacionales).
- **Datos desestructurados**: carecen de una estructura clara (por ejemplo, un documento de texto plano o un archivo de vídeo).
- **Datos semiestructurados**: tienen cierta estructura, pero no toda la información sigue el mismo formato, y puede variar dinámicamente.



Las **bases de datos tradicionales (relacionales)** resultan muy eficientes para datos estructurados. Sin embargo, cada vez es más frecuente encontrarnos con grandes volúmenes de datos que no encajan en un modelo tan rígido —por ejemplo, documentos que cambian en su estructura o contienen abundante texto—. Para atender esas necesidades, se ha ido popularizando el uso de **XML (eXtensible Markup Language)** como formato de almacenamiento e intercambio de información.

¿Por qué XML?



XML es un estándar del W3C ampliamente adoptado por su sencillez y flexibilidad. Entre sus características destacan:

- **Bien formado**: la sintaxis XML exige que todas las etiquetas se cierren correctamente.
- **Extensible**: permite ampliar el lenguaje con nuevas etiquetas o incluso crear lenguajes específicos sobre XML.
- **Legible y autodescriptivo**: su formato en texto claro facilita la lectura y la comprensión, dado que cada etiqueta describe el dato que contiene.
- **Portable**: se puede intercambiar fácilmente entre sistemas heterogéneos.
- **Parser universal**: cualquier aplicación o biblioteca que implemente un analizador (parser) XML puede procesar estos documentos sin importar el lenguaje de programación o la plataforma subyacente.

Gracias a estas ventajas, se ha convertido en uno de los lenguajes de marcas más utilizados para almacenar e intercambiar información, lo que a su vez ha impulsado la necesidad de **bases de datos específicas para XML**.

Documentos XML centrados en datos y documentos centrados en texto

A grandes rasgos, se pueden clasificar los documentos XML en dos categorías principales:

1. **Documentos XML centrados en datos**
    - Presentan muchos elementos de datos pequeños.
    - Su estructura es regular y está bien definida (p. ej.: facturas, pedidos, fichas de alumnos).
    - Se orientan a su procesamiento por aplicaciones o máquinas.
    - Típicamente manejan datos muy estructurados o semiestructurados.

2. **Documentos XML centrados en texto**
    - Contienen grandes bloques de texto y pocos elementos de datos.
    - Su estructura es menos predecible y puede variar notablemente (p. ej.: libros, artículos, informes).
    - Se orientan a la lectura humana y a la gestión de contenidos.
    - Típicamente manejan datos poco estructurados.

En la práctica, ambos tipos de documentos pueden convivir y es deseable contar con un sistema capaz de almacenar y recuperar ambos con eficiencia.

1.3 Comparativa con bases de datos relacionales

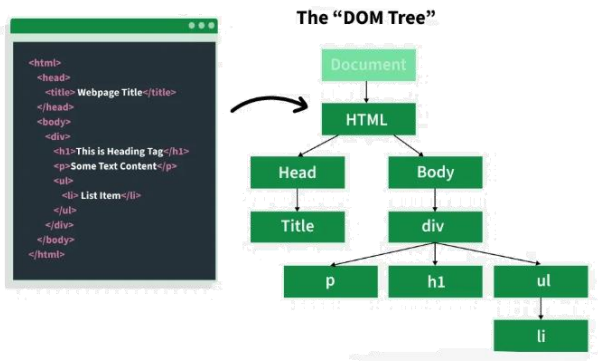
Las **Bases de Datos Relacionales (BDR)** se diseñaron para manejar datos altamente estructurados y normalizados. Se apoyan en un modelo tabular y en un lenguaje de consultas muy sólido, SQL.

En cambio, en la actualidad, los requisitos de **almacenar grandes volúmenes de datos semi o poco estructurados** hacen que las bases de datos relacionales resulten menos flexibles, especialmente cuando se pretende capturar la estructura y el contenido de documentos de texto variables.

Por ello, se ha impulsado el uso de **XML** (y otros formatos como JSON) para cubrir estos escenarios. Las bases de datos nativas XML:

- **Respetan la estructura del documento** de forma íntegra.
- Permiten **consultas** directamente sobre los nodos y las etiquetas del documento, sin necesidad de convertirlos a tablas (mapping).
- Pueden recuperar el documento **tal y como se insertó**, a diferencia de las BDR que, para almacenar XML, suelen fragmentarlo o guardarlo como un campo binario (BLOB).

En la práctica, cada vez más **SGBD relacionales** ofrecen extensiones para soportar documentos XML, dando lugar a **bases de datos XML-compatibles** (o XML-enabled). Sin embargo, cuando se quiere trabajar de forma íntegra y avanzada con documentos complejos, las **bases de datos nativas XML** son generalmente la solución más adecuada.



1.4 Ventajas e inconvenientes de las bases de datos nativas XML.

Ventajas

1. **Flexibilidad**: No exigen un esquema tabular rígido, lo que permite ajustarse mejor a datos con estructura variable o en constante evolución.
2. **Integridad del documento**: Se conserva la estructura y el contenido del XML, especialmente útil en documentos de texto extenso o con formatos complejos.
3. **Facilidad de consulta especializada**: Disponen de lenguajes potentes como **XQuery** o **XPath**, diseñados para navegar e interrogar estructuras XML de forma nativa.
4. **Estandarización**: Aprovechan estándares del W3C muy consolidados, lo que facilita la interoperabilidad y la portabilidad.

Inconvenientes

1. **Maduros, pero no tan extendidos**: Aunque están en continuo crecimiento, las bases de datos relacionales siguen dominando el mercado, por lo que la adopción masiva de las bases de datos nativas XML es menor.
2. **Gestión de transacciones y rendimiento**: Algunas implementaciones pueden no estar tan optimizadas como los SGBD relacionales, especialmente en transacciones complejas o de muy alto volumen.
3. **Curva de aprendizaje**: El personal de desarrollo y administración quizá necesite formarse en **XQuery, XPath y XSLT**, lenguajes menos habituales que SQL en muchos entornos.

1.5 Conceptos previos fundamentales.

1.5.1 El lenguaje XML

- **Significado y origen**:

  XML son las siglas de *eXtensible Markup Language*, un estándar del W3C introducido en 1998. Surge como un lenguaje de marcado pensado para representar datos de cualquier tipo, separándolos de su presentación visual (a diferencia de HTML).

- **Características principales**:
  - Fácil de leer, tanto para humanos como para máquinas.
  - Muy **verbose** (ocupa más espacio que formatos binarios).
  - Permite crear nuevos lenguajes basados en sus reglas (es extensible).
  - Es un estándar muy utilizado para el intercambio de información en la web.

1.5.2 Estructura de un documento XML

- **Documento de texto bien formado**:

  Un documento XML debe cumplir la sintaxis del lenguaje; si no, deja de ser XML.

- **Ejemplo simplificado**:

```
<?xml version="1.0" encoding="UTF-8"?>
<clientes>
  <cliente DNI="78901234X">
    <apellidos>NADALES</apellidos>
    <CP>44126</CP>
  </cliente>
</clientes>
```

- **Componentes esenciales**:

  1. **Cabecera**: Indica versión de XML y codificación (ej. <?xml version="1.0" encoding="UTF-8"?>).
  2. **Elementos**: Cada bloque entre etiquetas de apertura y cierre (<clientes>...</clientes>). Pueden contener otros elementos o texto.
  3. **Atributos**: Se declaran en las etiquetas de apertura (por ejemplo, DNI="78901234X").

1.5.3 DOM

- **Definición**:
  DOM (*Document Object Model*) es un modelo que representa un documento XML como un árbol en memoria, donde cada parte del documento (elementos, textos) se corresponde con nodos del árbol.

- **Características**:

  - Permite acceder y modificar los contenidos del documento.
  - Los atributos se guardan asociados a los elementos, no como nodos independientes.
  - Incluye nodos de tipo texto que pueden contener datos o incluso espacios y saltos de línea.

1.5.4 Parsers o analizadores sintácticos, serialización y deserialización

- **Parser**:
  Programa que comprueba si un documento XML está bien formado y, si es válido, puede construir una estructura de datos que lo represente (por ejemplo, un árbol DOM).

- **Deserialización y serialización**:

- **Deserializar**: convertir un documento de texto (secuencia de caracteres) en una estructura de datos interna (ej. DOM).
- **Serializar**: proceso inverso; generar un documento de texto a partir de la estructura de datos.

- **DOM vs. SAX**:
  - DOM crea un árbol completo en memoria, muy útil para accesos y modificaciones rápidas, pero puede resultar costoso en ficheros muy grandes.
  - SAX analiza secuencialmente sin almacenar el documento completo, adecuado cuando se necesitan procesar grandes volúmenes de datos sin cargarlos íntegramente en memoria.

2 Gestores de bases de datos XML

2.1 Tipos de gestores comerciales y libres (eXist-db, BaseX, MarkLogic, etc.)

Los *gestores nativos XML* se caracterizan por almacenar y gestionar documentos XML **sin mapeos** a tablas u otros modelos, respetando así su estructura jerárquica original. Existen gestores de código propietario y gestores de código libre u *open source*, cuyas funcionalidades pueden variar en cuanto a índices, transacciones y APIs de desarrollo.

- **Ejemplos de gestores comerciales**
    1. **TaminoXML Server (SoftwareAG)**
        - Primeros SGBD XML nativos en el mercado.
        - Almacena documentos en una base de datos propia, sin transformarlos.
        - Dispone de espacio dedicado para índices y documentos.
        - Soporta XQuery y ofrece APIs para Java, C y .NET.
    2. **TEXTML (Isiasoft)**
        - Almacena documentos en su formato nativo, sin mapeo.
        - Permite almacenar documentos con o sin DTD o esquema.
        - Ofrece índices sobre la estructura del documento y la posibilidad de usar varios a la vez.
        - Incluye APIs para Java, WebDAV, OLE DB y .NET.
- **Ejemplos de gestores libres**
    1. **eXist**
        - Sistema de almacenamiento propio (árboles B+ y archivos paginados).
        - Puede funcionar como servidor independiente o como biblioteca embebida en aplicaciones Java.
        - Organización en colecciones, soporta XQuery y XUpdate.
        - Permite almacenar documentos con o sin DTD/esquema.
    2. **MongoDB**
        - Orientado a documentos, trabaja principalmente con JSON o BSON (no estrictamente XML).
        - Código abierto, escrito en C++.
        - Modelo de *esquema dinámico*, sin rigidez tabular.

- Aunque no es un *SGBD XML nativo* en sentido estricto, se incluye a menudo en la familia NoSQL por su filosofía orientada a documentos.

2.2 Estrategias de almacenamiento e indexación

Las bases de datos nativas XML pueden diferenciarse en función de cómo guardan internamente los documentos:

1. **Almacenamiento basado en texto**
    - El documento se almacena tal cual (texto completo), a menudo con compresión e índices para agilizar la búsqueda.
    - Puede realizarse sobre un sistema de ficheros o BBDD tradicionales (guardando el XML como BLOB) y añadiendo índices específicos.
    - Se conserva la estructura original, pero la gestión interna depende de la implementación para consultas y transacciones.
2. **Almacenamiento basado en modelo**
    - Se define un modelo lógico de datos (por ejemplo, el DOM) para la estructura jerárquica.
    - El DOM se traduce y guarda en un almacén, que puede ser relacional, orientado a objetos o específicamente diseñado para XML.
    - Ejemplos de enfoques:
        - Traducir elementos y atributos a tablas relacionales.
        - Convertir a objetos en una base de datos orientada a objetos.
        - Emplear un *almacén específico* para documentos XML, sin conversión a modelos ajenos.

2.3 Colecciones y documentos

Una base de datos XML nativa suele organizar la información en **colecciones** y **documentos**:

- **Colecciones**
    - Conjunto de documentos agrupados normalmente por tema o tipo de datos.
    - Pueden contener otras colecciones anidadas, formando una jerarquía con un nodo raíz “/”.
- **Documentos**
    - Unidades fundamentales de información en formato XML (o, en algunos casos, *non-XML data*).

    - Suelen asimilarse a las “filas” de una tabla relacional o a los “ficheros” en un sistema operativo.

Esta jerarquía simplifica el acceso y la organización de los datos:

- Una colección actúa como un “directorio” o análogo a la “tabla” de una BD tradicional.
- Cada documento es un “fichero” individual o análogo a la “fila” de una tabla.

Según la implementación, es posible:

- Tener **varios DOCTYPE** asociados a una misma colección (una colección puede almacenar documentos con diferentes estructuras o tipos de definiciones DTS/esquema).
- Almacenar documentos sin esquema definido.
- Añadir un *Schema* a la colección (información física de índices y lógica de relaciones).
- Guardar ficheros no-XML con un DOCTYPE especial (*non-XML*).

En definitiva, la **estrategia jerárquica de colecciones y documentos** permite que las BD XML nativas gestionen de forma flexible grandes volúmenes de información, respetando la estructura de cada fichero.

3 Instalación y configuración del SGBD de XML nativo eXist

eXistDB ha sido elegido como SGBD de XML nativo por su facilidad de uso y su amplia funcionalidad. Al estar desarrollado en Java, puede ejecutarse en cualquier sistema operativo que disponga de una **máquina virtual de Java** (Windows, Linux, etc.).

3.1 Proceso de instalación

1. **Ejecutar el instalador**
   - Aceptar las opciones por defecto.
   - Se recomienda indicar la contraseña para el usuario administrador (**admin**).
2. **Instalar como servicio (opcional, pero recomendado)**
   - Es recomendable instalarlo como servicio del sistema. Si no se hace, arrancará automáticamente, pero, como recuerda un mensaje que puede aparecer en ese momento, se pueden perder cambios realizados en la base de datos si se apaga el ordenador sin cerrar antes eXist
   - Crea opciones en el menú de inicio para iniciar/detener eXist o instalar/desinstalar el servicio.
3. **Puerto de escucha**
   - Por defecto, eXist escucha en el **puerto 8080**.
   - Tras iniciar el servicio, puede accederse al *dashboard* en: http://localhost:8080

3.2 Acceso al dashboard

Una vez arrancado el servicio, eXist dispone de un **panel de control (dashboard)** accesible desde el navegador. Algunas de sus secciones principales son:

1. **Usermanager**
   - Permite crear usuarios, cambiar contraseñas y gestionar sus propiedades.
   - Si no se definió contraseña para *admin* durante la instalación, se puede hacer aquí.
2. **Collections**
   - Para gestionar colecciones (crear, borrar) y visualizar su contenido.
   - eXist trabaja con **una sola base de datos**, organizada internamente en colecciones anidadas.
   - Cada colección puede contener documentos XML y de otros tipos.
3. **eXide**

- IDE (entorno de desarrollo) para **XQuery**, con editor de texto para los documentos XML.
- Permite navegar por la base de datos y ejecutar consultas o sentencias para modificar documentos.

4. **Java Admin Client**
   - Suele iniciarse desde la bandeja del sistema.
   - Admite operaciones de administración, como:
     - Importar ficheros y directorios desde el sistema de ficheros.
     - Crear, borrar, mover y renombrar documentos.
     - Reindexar colecciones.
     - Gestionar usuarios y sus permisos.
     - Realizar y restaurar copias de seguridad (modo mantenimiento).
5. **Package Manager**
   - Permite instalar utilidades y paquetes adicionales.
   - Se recomienda la instalación de "eXist-db Demo Apps" para disponer de ejemplos listos para usar.
6. **Shutdown**
   - Cierra eXist. Si **no** se ha arrancado como servicio, es importante cerrarlo antes de apagar el ordenador para evitar la pérdida de cambios.

4 APIs para gestión de base de datos nativas XML

Cada base de datos nativa de XML suele incluir sus propias API. Aparte de ello, para Java se han desarrollado las API estándares **XML:DB** y **XQJ**, cuyo planteamiento es similar al de JDBC para bases de datos relacionales.

- **XML:DB**. Fue la primera API estándar para bases de datos de XML. Su última revisión es de 2001.
- **XQJ**. Son las siglas de XQuery for Java. Es una API más reciente: su última revisión es de 2009.Ha sido diseñada como un proyecto JCP *(Java Community Process),* pero no está incluida en la biblioteca estándar de clases de Java.

Como su propio nombre indica, XQJ es una API para XQuery, es decir, para operaciones de consulta en documentos de XML existentes, y de modificación con diversas extensiones de XQuery. XML:DB permite utilizar XQuery y además hacer operaciones no posibles con XQuery, en particular las realizadas sobre colecciones y sobre documentos como un todo (crea-ción y borrado), y además diversas tareas administrativas y de gestión.

En el resto de esta UT se aprenderá a realizar operaciones de **creación y borrado de colecciones y documentos con XML:DB**, y de **consulta y modificación de documentos con XQJ**.

4.1 La API XML:DB

4.1.1 Descripción

La API **XML:DB** no cuenta con versiones oficiales nuevas desde 2001. Aun así, sigue siendo utilizada internamente por muchos SGBD nativos de XML, como eXist. Esta API permite gestionar aspectos como colecciones, creación y borrado de documentos, usuarios y permisos, aunque puede haber diferencias significativas en la implementación según la base de datos.

Para ilustrar su funcionamiento, se mostrará cómo realizar operaciones que no son posibles con XQuery, especialmente:

- **Operaciones sobre colecciones**.
- **Creación y borrado de documentos** dentro de colecciones.

Para usar el *driver* de XML:DB en eXist, hay que **añadir** los archivos *JAR* de la instalación de eXist a nuestro proyecto. En particular:

- exist.jar (se encuentra en el directorio raíz de la instalación).
- Algún fichero *JAR* cuyo nombre empiece por xmldb-db-api y otros presentes en lib/core.

Si al ejecutar el programa se produce una excepción *ClassNotFoundException*, será necesario identificar el archivo JAR que contiene dicha clase y añadirlo al proyecto.

4.1.2 Establecimiento de conexiones y acceso a servicios

Para establecer una conexión con un SGBD usando XML:DB, se crea una instancia de una clase que implemente la interfaz Database y se indica una **cadena de conexión** (*connection string*) específica. El formato de la cadena de conexión varía según el SGBD. En el caso de **eXist**:

- **Clase Database**: org.exist.xmldb.DatabaseImpl
- **Cadena de conexión**: xmldb:exist://host:puerto/exist/xmlrpc/db/colección

En eXist, la conexión no se abre hasta que se intenta una operación que de verdad requiera acceder a la base de datos, por lo que un posible error en los datos de conexión puede pasar inadvertido hasta ese momento. Además, si la conexión es local (localhost), el puerto indicado se ignora y se toma el de la configuración de eXist.

Una **colección** proporciona varios servicios, y cada uno de estos ofrece distintas operaciones. Por ejemplo, un programa puede abrir la base de datos, acceder a una colección y mostrar las subcolecciones y documentos que hay en ella, así como enumerar los servicios disponibles y sus versiones.

https://exist-db.org/exist/apps/doc/devguide_xmldb

Ejemplo:

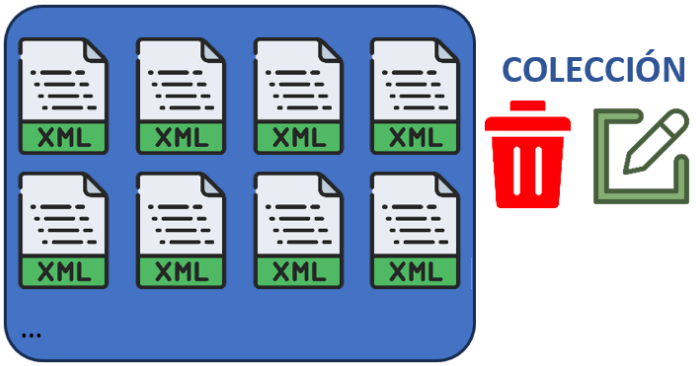
```java
import org.xmldb.api.base.*;
import org.xmldb.api.*;

public class TestExistConnection {

    public static void main(String[] args) {
        // Ajusta si es necesario, por ejemplo "org.exist.xmldb.DatabaseImpl"
        String driver = "org.exist.xmldb.DatabaseImpl";

        // URI típicamente: "xmldb:exist://[host]:[puerto]/exist/xmlrpc/db"
        // En eXist, "/db" es la colección raíz por defecto
        String URI = "xmldb:exist://localhost:8080/exist/xmlrpc/db";

        // Credenciales (ajusta a tu usuario/contraseña)
        String user = "admin";
        String pass = "admin";

        try {
            // 1. Cargar e instanciar la clase que implementa la interfaz "Database"
            Class<?> cl = Class.forName(driver);
            Database database = (Database) cl.newInstance();
            //Database database = (Database) cl.getDeclaredConstructor().newInstance();
            //Database database = (Database) cl.getDeclaredConstructor().newInstance();

            // 2. Registrar el driver en DatabaseManager
            DatabaseManager.registerDatabase(database);

            // 3. Obtener la colección raíz o la colección que desees
            Collection collection = DatabaseManager.getCollection(URI, user, pass);

            if (collection != null) {
                System.out.println("¡¡Conexión correcta con eXist!!");
                // Aquí puedes hacer más cosas, como listar subcolecciones o documentos
                collection.close();
            } else {
                System.out.println("No se pudo obtener la colección. Verifica la URL y
credenciales.");
            }
        } catch (ClassNotFoundException e) {
            System.err.println("ERROR: No se encontró la clase del driver.");
            e.printStackTrace();
        } catch (InstantiationException | IllegalAccessException e) {
            System.err.println("ERROR: No se pudo instanciar el driver.");
            e.printStackTrace();
        } catch (XMLDBException e) {
            System.err.println("ERROR: Fallo en la conexión con la base de datos eXist.");
            e.printStackTrace();
        }
    }
}
```

4.1.3 Creación y borrado de colecciones

Es posible crear y eliminar colecciones a través de la interfaz CollectionManagementService. Por ejemplo, un programa podría:

1. Obtener la colección raíz.
2. Invocar createCollection para **crear** colecciones.
3. Invocar removeCollection para **eliminarlas**.

El método para obtener la colección (obtenColeccion) sería similar al usado en el ejemplo anterior (acceso a la base de datos, conexión, etc.).

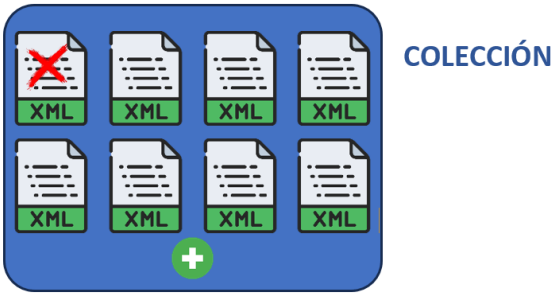


4.1.4 Creación y borrado de documentos

La creación y borrado de documentos se gestiona con las operaciones de la interfaz Resource. Un programa podría:

1. Llamar a createResource para **crear** un documento vacío en la colección.
2. Usar storeResource para **almacenarlo**.
3. Utilizar removeResource para **borrar** un documento de la colección.

Asimismo, la interfaz XMLResource dispone de métodos como setContent y getContent para asignar y recuperar el contenido del documento, además de versiones basadas en DOM y SAX (por ejemplo, setContentAsDOM o getContentAsDOM). eXist también permite almacenar documentos binarios no XML con BinaryResource en lugar de XMLResource.



4.2 La API XQJ

4.2.1 Introducción/revisión XQuery

XQuery es un lenguaje estándar del W3C creado para realizar **consultas sobre documentos XML**, del mismo modo que lo hacía previamente XPath. De hecho, **XQuery 1.0** incluye **XPath 2.0**, por lo que cualquier sentencia de XPath 2.0 es también una sentencia de XQuery 1.0 y produce los mismos resultados. Esta situación se repite con **XQuery 3.1** y **XPath 3.1**.

Se puede encontrar más información sobre XQuery en la especificación oficial del W3C:

http://www.w3.org/TR/xquery

Extensión para búsquedas de texto completo: XQuery Full Text

Dada la frecuencia de búsquedas de texto en documentos XML, donde a menudo se integra lenguaje natural (texto redactado por humanos), existe una extensión estándar del W3C para XQuery denominada **XQuery Full Text**.

Facilita la búsqueda en contenidos textuales y se describe brevemente en: http://docs.basex.org/wiki/Full-Text

CONSULTAS CON XQUERY

XQuery se puede usar sobre todos los documentos de una colección en un SGBD XML, como eXist, y se recomienda ejecutarlo y probarlo en **eXide** (el IDE de eXist). Algunas de sus características son:

- **Consultas FLWOR (for, let, where, order by, return)**
  - **FOR** vincula variables a expresiones XPath, que pueden devolver múltiples resultados.
  - **LET** asigna el resultado de una expresión a una variable (si hay varios resultados, se concatenan).
  - **WHERE** filtra los resultados generados por FOR y LET.
  - **ORDER BY** ordena los resultados.
  - **RETURN** define la expresión a evaluar para cada resultado, pudiendo generar XML y usar sentencias de XQuery entre llaves {}.

XQuery aporta muchas **funciones predefinidas** y la posibilidad de **definir nuevas**. Además, incorpora una sentencia condicional if ... then ... else, una cláusula GROUP BY y funciones de agregado como count, sum, min, max, etc.

Ejemplo de consulta XQuery:

- Tenemos una **colección pruebas** con los documentos **clientes.xml** y **productos.xml**

Clientes.xml

```
<clientes>
   <cliente DNI="78901234X">
      <apellidos>NADALES</apellidos>
      <CP>44126</CP>
   </cliente>
   <cliente DNI="89012345E">
      <apellidos>ROJAS</apellidos>
      <validez estado="borrado" timestamp="1528286082"/>
   </cliente>
   <cliente DNI="56789012B">
      <apellidos>SAMPER</apellidos>
      <CP>29730</CP>
   </cliente>
</clientes>
```

productos.xml

```
<productos>
   <producto nombre="tuerca">
      <precio>0.25</precio>
      <prov id="78901234X"/>
   </producto>
   <producto nombre="tornillo">
      <precio>0.10</precio>
      <prov id="89012345E"/>
      <prov id="78901234X"/>
   </producto>
</productos>
```

- Ejemplos de consultas:

`for $n in doc('/db/pruebas/Clientes.xml')` `return $n/clientes/cliente`	Devuelve los datos de todos los clientes. Cada uno de los resultados es un elemento de nombre `cliente` del documento.
`for $n in doc('/db/pruebas/Clientes.xml')` `return $n/clientes/cliente/apellidos/text()`	Se utiliza el paso `text()` para obtener no los elementos, sino el texto dentro.

`for $n in doc('/db/pruebas/Clientes.xml')/`
`  clientes/cliente`
`where substring($n/CP,1,2)="29"`
`return concat($n/apellidos, "-", $n/string(@DNI))`
	Se utiliza la cláusula WHERE para seleccionar clientes de Málaga. Se concatena un XPath a la especificación del documento `doc(...)`. Se utiliza la función `string()` para obtener el valor de un atributo. Se utiliza la función `concat()` para concatenar cadenas.

`for $n in doc('/db/pruebas/Clientes.xml')/`
`  clientes/cliente`
`order by number($n/CP)`
`return`
`<cli dni="{$n/string(@DNI)}" ape="{$n/apellidos}">`
`{$n/CP}`
`</cli>`
	Se utiliza la cláusula ORDER BY con la función `number()` para ordenar numérica y no alfabéticamente. Se devuelve XML, del que algunas partes se obtienen con XQuery entre llaves `{}`. Los resultados son XML, no texto, y su representación textual puede diferir de lo que aparece literalmente en la sentencia de XQuery, como es el caso de un cliente sin el elemento `CP`, cuyos datos se muestran como `<cli dni="89012345E" ape = "ROJAS"/>`.

`<clientes>`
`{`
`for $n in`
`doc('/db/pruebas/Clientes.xml')/clientes/cliente`
`return`
`<cliente>`
`<ident tipo="dni">{$n/string(@DNI)}</ident>`
`<apell>{$n/apellidos/text()}</apell>`
`</cliente>`
`}`
`</clientes>`
	Esta consulta muestra cómo con XQuery se puede incluir una sentencia FLWOR dentro de un documento de XML, y cómo XQuery permite generar documentos de XML con total flexibilidad, utilizando XPath para obtener todos los datos que sean necesarios en cualquier lugar en que sean necesarios.

`let $cl:=doc('/db/pruebas/Clientes.xml')/clientes/cliente`
`return`
`<nuestrosclientes>{$cl}</nuestrosclientes>`
	Uso de cláusula LET, no es el más habitual. El XPath devuelve varios resultados, pero se concatenan y esta consulta devuelve un único resultado.

`for $cli in doc('/db/pruebas/Clientes.xml')/clientes/cliente`
`let $id_cli:=$cli/string(@DNI)`
`return`
`<prod_cli dni="{$id_cli}" ape="{$cli/apellidos}">`
`{`
`for $prod in doc('/db/pruebas/productos.xml')/productos/producto/prov[@id=$id_cli]/`
`return`
`<prod>`
`{`
`$prod/string(@nombre)`
`}`
`</prod>`
`}`
`</prod_cli>`
	Uso de cláusula LET en una consulta sobre dos documentos. Se utiliza LET para guardar el DNI y en la cláusula RETURN se hace una nueva consulta para obtener todos los productos suministrados al cliente.

`for $n in doc('/db/pruebas/Clientes.xml')/clientes/cliente`
`order by number($n/CP)`
`return`
`element cli { attribute dni { $n/string(@DNI) },`
`data ($n/apellidos), element cp {$n/CP/text()}`
`}`
	El uso de constructores para elementos, atributos y textos de XML permite obtener un código de XQuery más escueto, aunque más alejado de la sintaxis de XML.

MODIFICACIÓN DE DATOS CON XQUERY

En eXist, las sentencias de **modificación de datos** son parte de la extensión denominada **XQuery Update Extension**. Permiten, por ejemplo:

- Insertar, eliminar o modificar nodos en un documento XML.
- Puedes probar a ejecutar estas operaciones desde eXide o desde cualquier entorno que soporte XQuery Update.

La sintaxis puede variar en otras bases de datos, pero el enfoque general y la idea de modificar datos XML directamente desde XQuery son similares.

De esta forma, **XQuery** se convierte en una herramienta completa para la **consulta** y **manipulación** de documentos XML, superando las capacidades básicas de XPath y abriendo posibilidades avanzadas de búsqueda y actualización en datos semi o poco estructurados.

- Ejemplo de modificación de datos:

`update insert`
`  <cliente DNI="67890123C">`
`  <apellidos> GAMBOA</apellidos>`
`  <CP>52351</CP>`
`</cliente>`
`into doc('/db/pruebas/Clientes.xml')/clientes`
	Inserta datos de un nuevo cliente Los datos se insertan como último elemento bajo el elemento `clientes` indicado.

`update insert`
`<cliente DNI="23456789D">`
`  <apellidos>DOLCE</apellidos>`
`  <CP>11895</CP>`
`</cliente>`
`following`
`doc('/db/pruebas/Clientes.xml')/clientes/`
`  cliente[@DNI="78901234X"]`
	Inserta datos de un nuevo cliente después de los de otro cliente especificado. Si no se especifica la posición de inserción, esta se realiza por defecto. Existe una cláusula `preceding` para realizar la inserción después de una posición determinada.

`update value doc('/db/pruebas/Clientes.`
`  xml')/clientes/cliente[@DNI="23456789D"]/`
`  apellidos`
`with "DORCE"`
	Cambia el valor en elemento `apellidos` para el cliente antes insertado.

`update replace doc('/db/pruebas/Clientes.`
`  xml')/clientes/cliente[@DNI="23456789D"]`
`with`
`<cliente DNI="12345678Z">`
`  <apellidos>ARCOS</apellidos>`
`</cliente>`
	Cambia el elemento indicado por un nuevo elemento. Nótese la diferencia de esta sentencia con la anterior. En la anterior solo se reemplazaba el texto dentro de un elemento, en esta se reemplaza el elemento entero.

`update rename doc('/db/pruebas/Clientes.`
`  xml')//validez`
`as "valid"`
	Cambia el nombre de elementos con nombre `validez` y les asigna el nombre `valid`.

`for $cli in doc('/db/pruebas/Clientes.xml')/`
`  clientes/cliente/valid[@estado="borrado"]`
`return update delete $cli`
	Borra clientes bajo los que exista un elemento `valid` con un atributo `estado` con valor `"borrado"`. Las sentencias que hacen cualquier cambio deben ejecutarse con mucha precaución. Igual que antes de ejecutar una sentencia que modifica los datos con SQL es conveniente ejecutar un SELECT con la misma cláusula WHERE para asegurarse de a qué datos va a afectar, con XQuery es conveniente ejecutar antes una consulta. En este caso, la consulta consistiría en eliminar `update delete` en la anterior sentencia.

4.2.2 Descripción de la API XQJ

XQJ es a **XQuery** (en el contexto de bases de datos XML) lo que **JDBC** es a SQL (en el contexto de bases de datos relacionales). Permite, desde programas Java, **ejecutar sentencias de XQuery** y, con sus extensiones, realizar también operaciones de modificación de datos.

- **Estructura**: la API XQJ define una serie de **interfaces** que los *drivers* (conectores) para distintas bases de datos nativas XML deben implementar.
- **Distribución**: no forma parte de la biblioteca estándar de clases de Java; por ello, es necesario descargar los ficheros JAR correspondientes.
- **Operaciones de consulta**: funcionan de forma análoga a JDBC.
- **Operaciones de modificación**: son más variadas, ya que se han propuesto diferentes extensiones de XQuery para actualizar documentos XML.

Las **Javadocs** de XQJ se pueden consultar en:

http://xqj.net/javadoc

El propio sitio http://xqj.net/exist ofrece los *drivers* XQJ para bases de datos de eXist, (existen otros diferentes para BaseX, Sedna o Marklogic). El fichero comprimido que se descarga contiene, al menos, dos JAR esenciales:

1. **xqjapi.jar**: con la API XQJ.
2. **Driver** de la base de datos elegida (por ejemplo, exist-xqj-1.0.1.jar para eXist).

Cada driver incluye una *Compliance Definition Statement* donde se especifica cómo se han implementado ciertos aspectos de XQJ (por ejemplo, el soporte de transacciones).

4.2.3 Establecimiento de conexiones con XQJ

Para crear un programa en Java que use XQJ:

1. **Añadir los ficheros JAR** al proyecto
    - El que contiene la API XQJ (por ejemplo, xqjapi.jar).
    - El driver específico para la base de datos (por ejemplo, exist-xqj-1.0.1.jar si es eXist), que implementa las interfaces de XQJ.
2. **Crear la XQDataSource**
    - Se instancia la clase que implementa la interfaz XQDataSource (por ejemplo, ExistXQDataSource para eXist).
    - Se configuran las propiedades de conexión necesarias mediante setProperty(). Los nombres de la clase y de las propiedades dependen de la implementación y se describen en la *Compliance Definition Statement* del driver.
3. **Establecer la conexión**
    - Generalmente, se hace con XQDataSource.getConnection(...) u otras variantes.
    - Si se indica *localhost* en eXist, puede que el puerto sea opcional, pues se toma el configurado por defecto.
    - Al igual que con XML:DB, **en eXist la conexión no se "abre" realmente** hasta que se ejecuta alguna operación que requiera acceso a la base de datos, por lo que no se lanzan excepciones si los datos de conexión son incorrectos hasta ese momento.

A continuación, se obtiene un objeto XQConnection, el cual permite consultar metadatos y soporte de transacciones (si lo hubiere). Si se detecta que no hay soporte para transacciones, se actuará en consecuencia (por ejemplo, se mantendrá el modo *auto-commit* activado).

4.2.4 Consultas con XQJ

La forma de realizar consultas XQuery con XQJ sigue un esquema parecido al de JDBC:

1. Se parte de una conexión XQConnection.

2. Se crea una **expresión XQ** con XQConnection.createExpression() u otro método similar.
3. Se ejecuta la consulta usando executeQuery(String query).
4. Se obtiene un objeto XQResultSequence, análogo a ResultSet en JDBC, sobre el que se itera con next().

Los resultados se pueden leer de varias formas:

- **Como un flujo de datos** (por ejemplo, XmlStreamReader mediante getItemAsStream()).
- **Como un Node** del modelo DOM con getNode().

Ejemplo básico: consultar los apellidos de todos los clientes en un documento Clientes.xml de la colección /db/pruebas. La sentencia XQuery se pasa como *String* y luego se recorre el resultado, imprimiendo cada nodo o valor obtenido.

4.2.5 Modificaciones de documentos con XQJ

Para modificar documentos XML usando XQuery Update (o extensiones similares):

1. Se crea igualmente una expresión XQ con createExpression().
2. En lugar de executeQuery(...), se usa executeCommand(...) (o método equivalente) para operaciones de actualización.
3. Se define la sentencia XQuery Update que inserta, borra o modifica nodos en un documento.
    - Por ejemplo, insert node <cliente>...</cliente> preceding //cliente[apellidos='NADALES']
    - Si la operación se ejecuta con éxito, los datos quedan actualizados en la base de datos.

Cada SGBD nativo XML puede tener ligeras diferencias en cómo implementa las extensiones de XQuery Update, pero el enfoque general es el mismo.

4.2.6 Transacciones con XQJ

El concepto de **transacción** (ACID) en XQJ funciona de modo parecido a JDBC, pero depende de:

- El **SGBD** (que lo permita).
- La **API** (XQJ, en este caso).
- El **driver** (que implemente dichas capacidades).

En bases de datos XML nativas, no siempre hay soporte de transacciones. Por ejemplo, **eXist** no las soporta, mientras que **XQJ** sí define el mecanismo. Si el driver no soporta transacciones y se llama a setAutoCommit(false), se producirá un error.

- **Auto-commit**: por defecto, está activado. Cada operación se confirma automáticamente.

- **Desactivar auto-commit**: se hace con setAutoCommit(false), permitiendo agrupar varias operaciones en una sola transacción. Entonces se deberán llamar a:
    - commit() para confirmar los cambios.
    - rollback() para deshacer cualquier operación realizada en la transacción actual.

Si ocurre una excepción (XQException) en mitad de la transacción, se debe capturar y ejecutar rollback() para mantener la coherencia de los datos. El siguiente pseudocódigo ilustra un uso básico:

```
try {
    XQConnection con = ds.getConnection();
    con.setAutoCommit(false);    // si está soportado
    // Realizar operaciones de actualización o consulta
    con.commit();
} catch (XQException e) {
    // Capturar la excepción y hacer rollback
    con.rollback();
    // Manejar el error (log, rethrow, etc.)
}
```

Si el SGBD o el driver no permiten transacciones, la llamada a setAutoCommit(false) lanzará una excepción para indicar que no se puede desactivar el modo automático.

5 Bibliografía

[1] Cortijo Bon, Carlos A. (2019). Capítulo 7 “Bases de datos de *XML*”. En editorial Síntesis (España). ISBN: 978-84-9171-356-2. *Acceso a datos.*

[2] Córcoles Tendero, Jose Eduardo. Capítulo 5 “*Bases de datos XML*”. En editorial RA-MA (España). E-ISBN: 9788499643908

[3] Camarena Estruch, Joan Gerard; Múrcia Andres, Jose A. Unidad 6: “*Bases de datos documentales. MongoDB y BBDD XML*”. En editorial McGraw Hill. ISBN 9788448626730. Acceso a datos