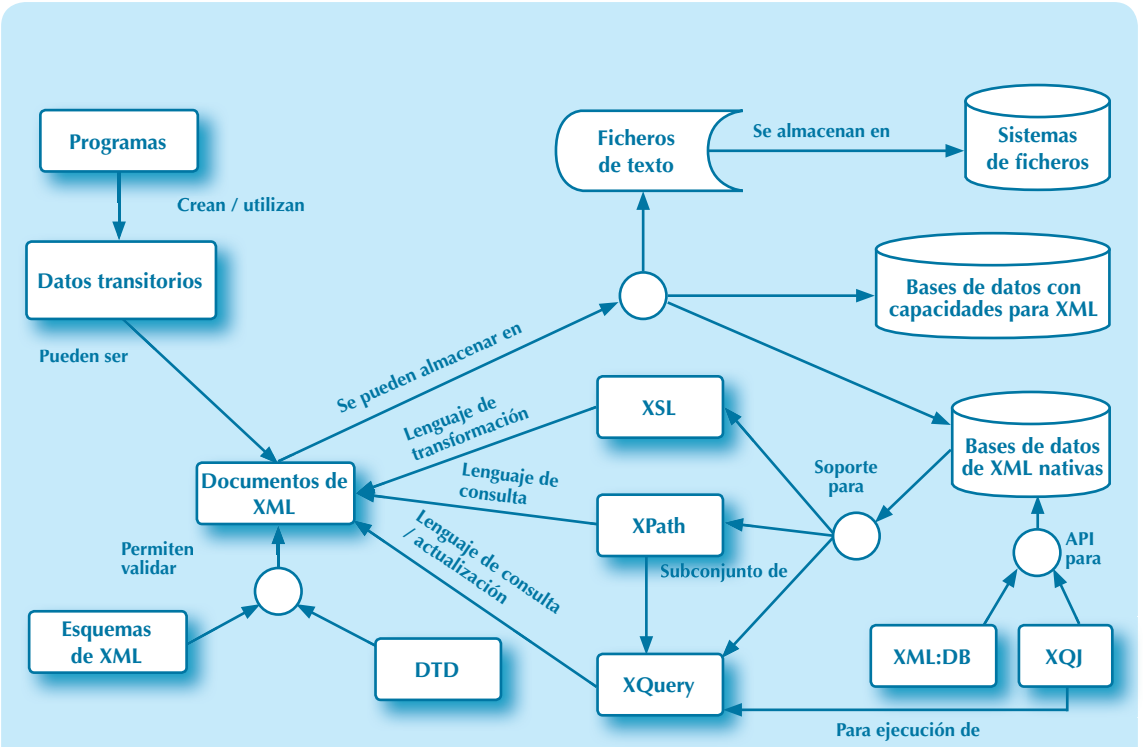
7

Bases de datos de XML

Objetivos

- ✓ Conocer los distintos tipos de aplicaciones basadas en XML: *data-centric* y *document-centric*.
- ✓ Comprender el propósito y los fundamentos de las bases de datos de XML nativas.
- ✓ Entender las diversas formas en que distintas bases de datos de XML nativas permiten organizar los documentos en colecciones.
- ✓ Aprender los lenguajes estándares XPath y XQuery para consulta y actualización de documentos.
- ✓ Utilizar la base de datos de XML nativa eXist para almacenar, consultar y modificar documentos de XML.
- ✓ Estudiar las API estándares para interacción con bases de datos de XML: XML:DB y XQJ.
- ✓ Trabajar con XML:DB para gestionar colecciones y documentos.
- ✓ Realizar operaciones de consulta y modificación de documentos con XQJ.
- ✓ Utilizar transacciones con XQJ.

Mapa conceptual



Glosario

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

XML:DB. API para bases de datos de 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 consulta para documentos de XML que permite recuperar un conjunto de nodos de su estructura jerárquica.

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

XQuery. Lenguaje de consulta para documentos de 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.

7.1. XML como soporte para almacenamiento e intercambio de datos

XML es un formato muy sencillo y flexible que permite representar cualquier tipo de información en documentos de texto con una estructura jerárquica. Su uso se ha extendido cada vez más desde su creación como estándar de W3C en 1998, y en torno a él ha ido surgiendo un abundante cuerpo de estándares. Desde su introducción, el desarrollo de nuevos servicios y aplicaciones sobre internet se ha basado cada vez más en XML. XML se utiliza a menudo como soporte para almacenamiento de datos. Sus aplicaciones son cada vez más diversas, pero se pueden distinguir a grandes rasgos dos áreas de aplicación:

1. *XML centrado en documentos (document-centric XML)*. Corresponde al uso inicial de los lenguajes de marcado antecesores de XML, como HTML. Se utilizan documentos de XML que no tienen una estructura fija y regular, producidos mediante procesos manuales por personas para su utilización en procesos manuales realizados por personas. Suelen contener textos en lenguaje natural.
2. *XML centrado en datos (data-centric XML)*. Se utilizan documentos XML con una estructura bien definida y regular. Contienen datos. Son normalmente producidos por máquinas y consumidos por máquinas mediante procesos automáticos que utilizan lenguajes estándares como XML Schema, XPath, XQuery, XSL, etc.

Estos son dos casos extremos, y puede haber planteamientos intermedios. Por ejemplo: dentro de un documento de XML centrado en documento puede existir una sección con una estructura bien definida y regular, o bien dentro de un documento de XML centrado en datos puede haber una sección donde se admita un formato más flexible.

7.2. Alternativas para el almacenamiento de documentos de XML

Existen varias posibilidades para la persistencia de documentos de XML. Cada una puede ser más o menos apropiada dependiendo del tipo de documentos que almacenar y su uso previsto.

1. *Sistemas de ficheros*. Cada documento de XML se almacena como un fichero en una jerarquía de directorios. Para el manejo de los documentos se dispone solo de las funcionalidades que para el manejo de ficheros proporciona el sistema operativo, entre las que no suele estar el control de accesos concurrentes, ni tampoco el soporte para transacciones. Cualquier tipo de consulta o de modificación que se realice sobre el contenido de los documentos debe hacerse manualmente o programarse a medida.
2. *Bases de datos con capacidades para XML (XML-enabled)*. Pueden ser de distintos tipos, como por ejemplo relacionales u orientadas a objetos. Se necesitan mecanismos de traducción entre XML y el esquema de almacenamiento de la base de datos, que no estará en general optimizado para documentos de XML, lo que puede conllevar un mayor consumo de espacio de almacenamiento y un menor rendimiento en las operaciones realizadas sobre los documentos.
3. *Bases de datos nativas de XML*. El almacenamiento se realiza en estructuras diseñadas y optimizadas para documentos de XML. Proporcionan, además, soporte para lenguajes estándares que permiten realizar diversos tipos de operaciones sobre documentos de XML: consulta (XPath, XQuery), actualización (diversas extensiones de XQuery), y

transformación (XSL). Proporcionan implementaciones de API estándares para bases de datos de XML, tales como XML:DB y XQJ.

7.3. Almacenamiento de XML en SGBD relacionales

Es de especial interés plantear las distintas posibilidades para persistencia de documentos XML en bases de datos relacionales, por varios motivos:

1. Son, con diferencia, las más empleadas en la actualidad para todo tipo de aplicaciones. En principio es recomendable utilizarlas a no ser que el uso de una base de datos de XML nativa suponga una clara ventaja.
2. Pueden perfectamente almacenar documentos de XML, por grandes que sean, en columnas de tipo CLOB (*character large object*) o BLOB (*binary large object*). Si se trata de documentos muy pequeños (pocos *kilo*bytes), podrían incluso almacenarse en columnas de tipo VARCHAR. Por tanto, son perfectamente válidas si lo único que se necesita es almacenar y recuperar documentos de XML enteros.
3. Es relativamente sencillo almacenar los datos que contienen los documentos de XML centrados en datos (*data-centric*) en esquemas relacionales, y generar los documentos a partir de ellos mediante procesos automáticos. Además, como se explica a continuación, las últimas versiones del estándar SQL incluyen características para XML que pueden facilitar mucho esta tarea.
4. El estándar SQL incluye, desde SQL:2003, SQL/XML en su parte 14. SQL/XML incluye, entre otras cosas:

   a) El tipo de datos XML para almacenar documentos de XML.
   b) Correspondencias entre tipos de datos de SQL y de XML para facilitar el almacenamiento y manipulación de XML en bases de datos con SQL.
   c) Posibilidad de utilizar XQuery dentro de consultas en SQL. Como resultado de una consulta en XQuery, se puede obtener una relación, como con una consulta en SQL. Esto permite realizar consultas de SQL que combinen y relacionen información existente en esquemas relacionales y documentos de XML.
   d) A la inversa, se pueden generar documentos de XML con el resultado de consultas en SQL. Esto permite realizar consultas en XQuery que combinen y relacionen información existente en documentos de XML y en esquemas relacionales.

5. Algunas bases de datos relacionales se pueden considerar *XML-enabled* porque implementan SQL/XML, lo que incluye soporte para XQuery, e incluso soporte para otros estándares de XML, como por ejemplo XSL.
6. Las bases de datos relacionales proporcionan, en general, un magnífico y completo soporte para transacciones, incluyendo en muchos casos transacciones distribuidas. No se puede decir tanto de las bases de datos de XML nativas, si bien es cierto que algunas bases de datos de XML nativas tienen soporte para transacciones.

La separación entre bases de datos relacionales y de XML nativas no es nítida. El estándar SQL incluye desde SQL:2003 muchas características para XML, y algunas bases de datos relacionales añaden soporte para otras tecnologías de XML como XSL, y la implementación de API estándar para bases de datos de XML. Si la funcionalidad y el rendimiento son satisfacto-

rios, lo de menos es que el almacenamiento se realice, en última instancia, sobre un esquema relacional. Todas las versiones de la base de datos de Oracle desde la 12c incluyen XML DB.

XML DB incluye un buen soporte para SQL/XML y diversas tecnologías de XML.

En conclusión, antes de optar por el uso de una base de datos de XML nativa conviene considerar el uso de una base de datos relacional.

Recurso web

www

El siguiente enlace proporciona información acerca de Oracle XML DB, incluyendo código de ejemplo y documentación:

https://www.oracle.com/database/technologies/appdev/xmldb.html

7.4. Características de las bases de datos de XML nativas

Antes que nada, hay que aclarar que, en lo sucesivo, su utilizará el término *base de datos* (se sobreentiende de XML) con dos significados: el de almacenamiento de documentos de XML y el de SGBD (sistema gestor de bases de datos). El significado exacto será claro por el contexto. *Base de datos de XML nativa* significará siempre SGBD de XML nativo.

Las bases de datos de XML nativas deben permitir el almacenamiento directo de documentos de XML en estructuras diseñadas específicamente para XML, sean del tipo que sean: basados en contenido o en documentos, o cualquier otra posibilidad. Esto no excluye que puedan almacenar documentos que no sean de XML, si bien el conjunto de operaciones que se pueda realizar sobre ellos será más limitado. Por lo demás, suelen tener las siguientes características:

1. *Colecciones*. Dentro de una base de datos de XML existente en un SGBD de XML, se pueden organizar los documentos en colecciones. La manera de organizar los documentos en colecciones varía mucho de un SGBD de XML nativo a otro.
2. *Validación*. Se pueden usar esquemas o DTD para validar los documentos almacenados.
3. *Mecanismos estándares para consulta y transformación de documentos de XML*. A saber: XPath, XQuery y XSL (todos estándares de W3C). Pueden incluir la extensión XQuery Full Text, también estándar de W3C, para búsqueda en textos de lenguaje natural, tan frecuentes en los documentos de XML.
4. *Mecanismos estándares para modificación de documentos*. Debería ser posible modificar los contenidos de un documento sin reemplazarlo globalmente por otro. Han surgido diversos lenguajes para permitir esto, en general extensiones de XQuery.
5. *Soporte para API estándares*. Las dos API más importantes para la interacción con bases de datos de XML son XML:DB y XQJ. Pero cada base de datos de XML tendrá, en general, sus propias API, y las implementaciones de las API estándares se basarán en ellas.
6. *Indexación*. El uso de índices es fundamental para agilizar las consultas sobre los contenidos de los documentos de XML. Pero su indexación es mucho más problemática que la de las tablas de una base de datos relacional. Esto es debido a la mayor complejidad de los documentos de XML, con una estructura jerárquica en vez de tabular, ya que, en

una misma colección, puede haber documentos de distintos tipos. Se necesitan mecanismos de indexación muy flexibles y adaptativos. Las bases de datos de XML nativas suelen utilizar índices estructurales de todos los documentos, en los que se incluyen los valores de todos los elementos y todos los atributos. En ellos cada documento viene identificado por un identificador único asignado automáticamente por el sistema. Debido al mayor coste computacional de actualizar los índices, algunas bases de datos no lo hacen automáticamente tras cada cambio en los datos, lo que hay que tener muy en cuenta para el desarrollo de aplicaciones y para su administración. Las distintas bases de datos permiten crear distintos tipos de índices. La indexación de bases de datos de XML nativas es un aspecto poco o nada estandarizado.

7. *Transacciones*. Las diversas bases de datos de XML nativas proporcionan un cierto grado de soporte para transacciones, pero con frecuencia no ofrecen un completo soporte para transacciones ACID, como es la norma para las bases de datos relacionales. La API XQJ incluye métodos para gestión de transacciones, pero el soporte para transacciones es opcional para una implementación de XQJ. En cualquier caso, para que se puedan utilizar desde la API, el SGBD debe soportarlas.

7.5. Gestores comerciales y libres

Existen multitud de bases de datos de XML, tanto nativas como *XML-enabled,* y tanto comerciales como libres. Hay que tener en cuenta que, aunque XML se ha venido utilizando cada vez más, las bases de datos de XML nativas siempre han tenido, y siguen teniendo, un uso muy reducido. Se han desarrollado muchas desde finales de los años noventa, comerciales y libres, pero en pocas se invierte actualmente esfuerzo de soporte y desarrollo. En cambio, las bases de datos relacionales han ido cada vez más adoptando características para XML, incluso proporcionando implementaciones bastante completas de SQL/XML, parte del estándar SQL.

Recurso web

En la siguiente dirección se puede encontrar una descripción muy completa de las características más importantes de muchos productos de bases de datos de XML. Son de especial interés los enlaces titulados "Native XML Databases" y "XML-Enabled Databases":

http://www.rpbourret.com/xml/XMLDatabaseProds.htm

A continuación se reseñan brevemente varios SGBD de XML nativos comerciales y libres.

1. *Tamino*. Es un SGBD comercial de la empresa Software AG. Surgió durante el *boom* de XML a finales de los noventa. Es un producto muy caro y con muy buen rendimiento debido a su almacenamiento de datos diseñado específicamente para XML y a la indexación que realiza de los documentos. En el momento de la redacción de este libro, en la página web de Software AG no existe documentación ni información de producto para Tamino.

2. *Marklogic*. Este SGBD comercial comenzó como un SGBD de XML nativo a principios de los 2000 para evolucionar más recientemente hacia una base de datos multimodelo, es decir, que integra varios mecanismos de almacenamiento (en este caso, XML, JSON y tripletas semánticas de RDF), y ofrece una vista lógica unificada de todos los datos. Organiza los documentos en directorios y colecciones. Los directorios forman una jerarquía. Las colecciones agrupan documentos que tienen algo en común, como por ejemplo tema o autor. Un documento está en un directorio y puede estar en muchas colecciones. Existe una versión gratuita, descargable previo registro, para uso personal y no comercial. Se puede encontrar más información de Marklogic en su página web: https://www.marklogic.com
3. *eXist*. Es un SGBD de *software* libre. En una instalación de eXist solo existe una base de datos que contiene una jerarquía de colecciones, y dentro de cada una puede haber documentos de XML y de otros tipos. Asigna a cada documento un identificador único y crea y mantiene automáticamente índices estructurales para todos ellos. Se puede encontrar más información de eXist en su página web: https://exist-db.org.
4. *BaseX*. Es un SGBD de *software* libre. En una base de datos de BaseX no se crean las colecciones explícitamente. En lugar de ello, se crean y borran implícitamente, dependiendo de la existencia de documentos en *paths* específicos. Los documentos pueden ser de tipo XML o de tipo *raw* (que en este caso significa todo lo que no sea XML). Se puede encontrar más información de BaseX en su página web: http://basex.org
5. *Sedna*. Es un SGBD de *software* libre. Está escrito en C/C++, mientras que prácticamente todos los SGBD de XML nativos están escritos en Java. Es muy eficiente y está muy optimizado en todos los aspectos, como por ejemplo su esquema de almacenamiento para XML y su API propia basada en un protocolo propio binario, y no, como es habitual, en protocolos tales como SOAP, XML-RPC o similares. Fue un magnífico SGBD y un desarrollo muy prometedor en su momento, pero, de acuerdo con la información de su página web y de sourceforge.net, no parece haber actividad ni nuevos desarrollos desde 2012. Se puede encontrar más información de Sedna en su página web: http://www.sedna.org.

7.6. Instalación y configuración del SGBD de XML nativo eXist

Se ha elegido eXist como SGBD de XML nativo para trabajar por su facilidad de uso y funcionalidad. Como casi todas las bases de datos de XML nativas, está desarrollada en Java, por lo que funciona para Windows, Linux y cualquier sistema operativo para el que esté disponible una máquina virtual de Java.

Para instalarlo basta ejecutar el programa de instalación y aceptar todas las opciones por defecto. Se debería indicar la contraseña para el usuario administrador `admin`. Si no, se puede hacer más adelante. Con ello se crean opciones en el menú de inicio para lanzar y parar eXist, y para instalarlo o desinstalarlo como servicio del sistema.

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.

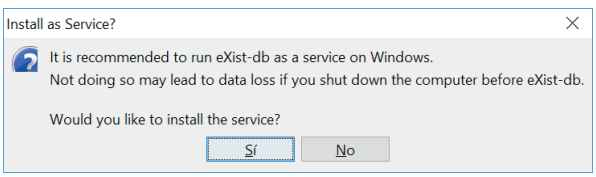


Figura 7.1
Instalación de eXist-db como servicio de Windows

El *dashboard* (panel de control) de eXist está disponible en un servidor web que escucha por defecto por el puerto 8080.

TOMA NOTA

Hay diversos programas que suelen utilizar el puerto 8080, por ejemplo Tomcat y el servidor de base de datos Oracle. Si ya está en uso, se mostrará un error y se dará la opción de visualizar los *logs,* en los que aparecerá un mensaje similar al siguiente: `ERROR (JettyStart.java [run]:369) - ERROR: Could not bind to port because Address already in use: bind`. Se puede cambiar el puerto en el fichero `tools\jetty\etc\jetty-http.xml` en el directorio de instalación de eXist, en la línea donde dice `<SystemProperty name="jetty.port" default="8080"/>`. En esta instalación se ha cambiado a 8090, como se puede ver en algunos ejemplos siguientes.

Una vez iniciado eXist, se puede acceder al *dashboard* o panel de control en la dirección http://localhost:*puerto*. En él se pueden encontrar, entre otras cosas:

1. *Usermanager*. Para crear nuevos usuarios y cambiar sus propiedades. Si no se asignó contraseña para el usuario `admin` durante la instalación, se puede hacer aquí.
2. *Collections*. Para gestionar, crear y borrar colecciones. Una instalación de eXist gestiona una sola base de datos en la que pueden definirse colecciones dentro de otras colecciones, formando una jerarquía. En cada una puede haber documentos de XML y de otros tipos.
3. *eXide*. Magnífico IDE (entorno de desarrollo integrado) para XQuery. Permite navegar por los contenidos de la base de datos y visualizar los resultados de consultas con XQuery, y también ejecutar sentencias de XQuery que modifiquen los documentos. Permite modificar los contenidos de los documentos de XML mediante un editor de texto.
4. *Java Admin Client*. También se puede lanzar desde la bandeja del sistema. Además de consultas, permite realizar diversas tareas de gestión de los contenidos de la base de datos y administrativas, como por ejemplo:

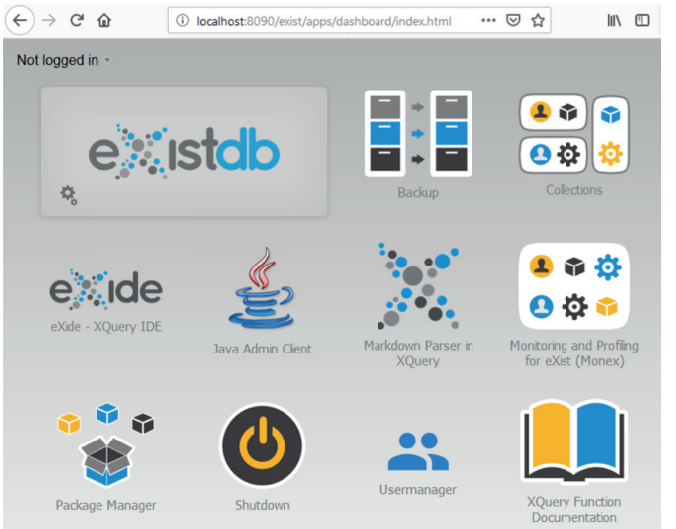


Figura 7.2
Dashboard (panel de control) de eXist

   a) Importar ficheros y directorios seleccionados desde un sistema de ficheros.
   b) Gestionar colecciones.

c) Crear, borrar, mover y renombrar documentos.
d) Reindexar colecciones.
e) Gestionar usuarios, incluyendo sus permisos sobre colecciones y documentos.
f) Realizar y restaurar copias de seguridad, en modo de mantenimiento.

5. *Package Manager*. Permite instalar muchos paquetes y utilidades adicionales. Se recomienda instalar "eXist-db Demo Apps". Con ello se instalará alguna base de datos de ejemplo que se utilizará después.
6. *Shutdown*. Para parar el SGBD eXist. Si no se ha arrancado mediante un servicio, hay que pararlo antes de apagar el ordenador. En caso contrario podrían no guardarse todos los cambios realizados, como se ve en la figura 7.1.

7.7. API para gestión de bases de datos nativas de 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 (creación y borrado), y además diversas tareas administrativas y de gestión. En el resto de este capítulo 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.

TOMA NOTA

Es posible que las implementaciones de XML:DB o XQJ para algunas bases de datos no proporcionen determinadas funcionalidades que sí están disponibles en las API propias, o que estas últimas proporcionen mejor rendimiento para algunas operaciones. En ese caso, el uso de las API propias puede ser una alternativa que considerar, dependiendo de la aplicación.

7.8. La API XML:DB

No existe ninguna nueva versión oficial de esta API desde 2001. A pesar de ello, sigue siendo ampliamente utilizada internamente por muchos SGBD de XML nativos, como es el caso de eXist. Esto, unido al hecho de que permite gestionar diversos aspectos como las colecciones, la

creación y borrado de documentos, los usuarios y sus permisos, y otras cosas que varían mucho de una base de datos a otra, hace que pueda haber diferencias significativas entre implementaciones para distintas bases de datos.

Recurso web

Se puede acceder a los Javadocs de XML:DB en la siguiente dirección: http://xmldb-org.sourceforge.net/xapi/api/index.html.

Para ilustrar el funcionamiento de esta API, se utilizará para realizar algunos tipos de operaciones sobre la base de datos que no son posibles con XQuery. En particular:

- Operaciones sobre colecciones.
- Operaciones de creación y borrado de documentos de XML dentro de colecciones.

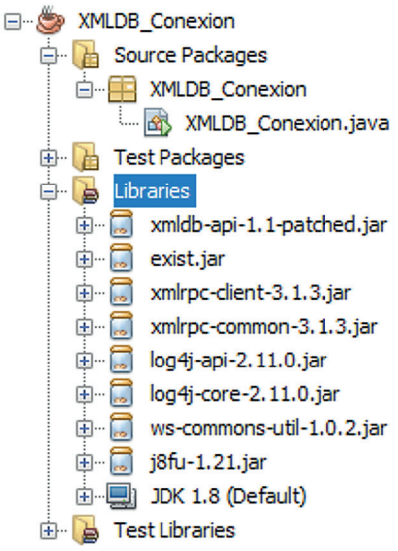


Figura 7.3
Proyecto que usa la API XML:DB

Para utilizar el *driver* de XML:DB para eXist hay que añadir al proyecto algunos ficheros jar disponibles en la instalación de eXist. Siempre hay que añadir `exist.jar` (presente en el directorio base de la instalación) y un fichero jar cuyo nombre comienza por `xmldb-db-api`, y otros presentes en el directorio `lib/core`. Según las funcionalidades que utilice el programa, podrá ser necesario añadir más ficheros jar para que se puedan cargar en tiempo de ejecución todas las clases necesarias. Si falta alguna, se producirá una excepción `ClassNotFoundException`. Entonces habrá que localizar el fichero jar que proporciona dicha clase y añadirlo al proyecto. En caso de dificultades, puede ser de ayuda algún buscador de ficheros jar que permita obtener el nombre del fichero jar a partir del nombre de la clase, como www.findjar.com

7.8.1. Establecimiento de conexiones y acceso a servicios con XML:DB

Para establecer una conexión con un SGBD con XML:DB, es necesario crear una instancia de una clase que implemente la interfaz `Database`, y proporcionar una cadena de conexión para la base de datos. El formato de la cadena de conexión varía según el SGBD. En la siguiente tabla se indica su formato para algunos SGBD de XML nativos. En ella se puede indicar el nombre de la base de datos y la colección, según el SGBD.

CUADRO 7.1
Propiedades para conexión para *drivers* de XML:DB para distintas bases de datos nativas de XML

SGBD	Clase que implementa Database	Cadena de conexión
eXist	`org.exist.xmldb.DatabaseImpl`	`xmldb:exist://`*`host`*`:`*`puerto`*`/exist/xmlrpc/db/`*`colección`*
BaseX	`org.basex.api.xmldb.BXDatabase`	`xmldb:basex://localhost:`*`puerto`*`/`*`basedatos`*
Sedna	`net.cfoster.sedna.DatabaseImpl`	`xmldb:sedna://`*`host`*`:`*`puerto`*`/`*`basedatos/colección`*

Con eXist no se intenta abrir la conexión en el mismo momento en que se solicita, sino en el momento en que se intenta alguna operación que requiere acceso efectivo a la base de datos. Cualquier error en los datos de conexión pasará inadvertido hasta entonces. Además, si la conexión es a `localhost`, se ignora el puerto indicado y se toma de la configuración de eXist.

Una colección proporciona una serie de servicios, y cada uno de ellos puede proporcionar una serie de operaciones. El siguiente programa de ejemplo abre una conexión con una base de datos de eXist, y accede a una colección determinada dentro de ella, que siempre existe en una instalación típica, y muestra las colecciones y los documentos que hay en ella. También muestra una lista con los servicios proporcionados por la colección, y la versión de cada uno. Otros SGBD pueden proporcionar un conjunto distinto de servicios. Por ejemplo, la versión 3.5 de Sedna proporciona, además de los que ofrece eXist, los servicios `TransactionService`, `SednaUpdateService`, `XQueryService`, `IndexManagementService` y `ModuleManagementService`, cuyos nombres dan idea de su utilidad. Algunos de ellos están documentados en la especificación de XML:DB, y algunos de ellos no, al ser propios del SGBD Sedna.

```
// Muestra colecciones, documentos servicios disponibles en una colección
package XMLDB_Conexion;
import org.xmldb.api.base.Collection;
import org.xmldb.api.base.Database;
import org.xmldb.api.base.Service;
import org.xmldb.api.base.XMLDBException;
import org.xmldb.api.DatabaseManager;
public class XMLDB_Conexion {
    private static Collection obtenColeccion(String nomCol) throws Exception {
        Database dbDriver;
        Collection col;
        dbDriver = (Database)
        Class.forName("org.exist.xmldb.DatabaseImpl").newInstance();
        DatabaseManager.registerDatabase(dbDriver);
        col = DatabaseManager.getCollection(
        "xmldb:exist://localhost:8090/exist/xmlrpc/db" + nomCol,
        "(usuario)", "(contraseña)");
        return col;
    }
    public static void main(String[] args) {
        Collection col = null;
```

```
        try {
            col = obtenColeccion("/apps/shared-resources");
            System.out.println("Colección actual: " + col.getName());
            int numHijas = col.getChildCollectionCount();
            System.out.println(numHijas + " colecciones hijas.");
            if (numHijas > 0) {
                String nomHijas[] = col.listChildCollections();
                for (int i = 0; i < numHijas; i++) {
                    System.out.println("\t" + nomHijas[i]);
                }
            }
            int numDocs = col.getResourceCount();
            System.out.println(numDocs + " documentos.");
            if (numDocs > 0) {
                String nomDocs[] = col.listResources();
                for (int i = 0; i < numDocs; i++) {
                    System.out.println("\t" + nomDocs[i]);
                }
            }
            Service servicios[] = col.getServices();
            System.out.println("Servicios proporcionados por colección " + col.
                getName() + ":");
            for (int i = 0; i < servicios.length; i++) {
                System.out.println("\t" + servicios[i].getName() + " - Versión:
                    " + servicios[i].getVersion());
        }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                if (col != null) {
                    col.close();
                }
            } catch (XMLDBException e) {
                e.printStackTrace();
            }
        }
    }
}
```

7.8.2. Creación y borrado de colecciones con XML:DB

El siguiente programa crea dos colecciones en el nivel más alto de la jerarquía, y acto seguido borra una de ellas. Primero, obtiene un `CollectionManagementService` a partir de la colección raíz, y después, utiliza los métodos `createCollection` y `removeCollection` de este servicio para crear y eliminar colecciones dentro de ella. El método `obtenColeccion` en este programa y en los siguientes es igual que en el anterior y, por tanto, su código se omite.

```
// Creación y borrado de colecciones

package XMLDB_CreacionColecciones;

import org.xmldb.api.base.Collection;
import org.xmldb.api.base.Database;
```

```
import org.xmldb.api.base.XMLDBException;
import org.xmldb.api.DatabaseManager;
import org.xmldb.api.modules.CollectionManagementService;

public class XMLDB_CreacionColecciones {

    private static Collection obtenColeccion(String nomCol) throws Exception {
        (...)
    }

    public static void main(String[] args) {
        Collection col = null;
        try {
            col = obtenColeccion("");
            System.out.println("Colección actual: " + col.getName());
            System.out.println(col.getChildCollectionCount()+" colecciones
                hijas antes.");
            CollectionManagementService cmServ =
            (CollectionManagementService) col.getService(
            "CollectionManagementService", "1.0"
            );
            cmServ.createCollection("prueba1");
            cmServ.createCollection("pruebas");
            cmServ.removeCollection("prueba1");
            System.out.println(col.getChildCollectionCount()+" colecciones
                hijas después.");
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
        try {
            if (col != null) {
                col.close();
            }
        } catch (XMLDBException xe) {
                xe.printStackTrace();
            }
        }
    }
}
```

7.8.3. Creación y borrado de documentos con XML:DB

El siguiente programa crea tres documentos en una colección creada por un programa anterior, y después borra uno de ellos. Los documentos se crean con `createResource` y se almacenan con `storeResource`. Para asignar y recuperar el contenido de un documento, la interfaz `XMLResource` tiene métodos `setContent` y `getContent`, y además `setContentAsDOM`, `getContentAsDOM`, `setContentAsSAX` y `getContentAsSAX`. eXist permite almacenar documentos que no son de XML (a los que denomina binarios). Para ello se utilizaría `BinaryResource` en lugar de `XMLResource`.

```
// Creación y borrado de documentos
package XMLDB_CreacionBorradoDocumentos;
import org.xmldb.api.base.*;
import org.xmldb.api.DatabaseManager;
```

```
import org.xmldb.api.modules.*;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;
import org.xml.sax.InputSource;
import org.xml.sax.ContentHandler;
import javax.xml.stream.XMLOutputFactory;
import javax.xml.stream.XMLStreamWriter;
import java.io.StringReader;
import java.io.StringWriter;
public class XMLDB_CreacionBorradoDocumentos {
    private static Collection obtenColeccion(String nomCol) throws Exception {
    (...)
    }
    public static void main(String[] args) {
        Collection col = null;
        try {
            col = obtenColeccion("/pruebas");
            XMLResource recurso;
            // Crear documento a partir de String
            recurso = (XMLResource) col.createResource("Clientes.xml",
                XMLResource.RESOURCE_TYPE);
            recurso.setContent("<clientes>\n"
                + "<cliente DNI=\"78901234X\">"
                + "<apellidos>NADALES</apellidos><CP>44126</CP></cliente>\n"
                + "<cliente DNI=\"89012345E\">"
                + "<apellidos>ROJAS</apellidos>"
                + "<validez estado=\"borrado\" timestamp=\"1528286082\"/>"
                + "</cliente>\n"
                + "<cliente DNI=\"56789012B\">\n"
                + "<apellidos>SAMPER</apellidos><CP>29730</CP></cliente>"
                + "</clientes>");
            col.storeResource(recurso);
            recurso = (XMLResource) col.createResource("Empresa_.xml",
                XMLResource.RESOURCE_TYPE);
            recurso.setContent("<empresa CIF=\"A34246801\">MegaExport"
                +"<sedes><sede>LEÓN</sede><sede>CÁCERES</sede></sedes>"
                +"</empresa>");
            col.storeResource(recurso);
            // Crear documento a partir de objeto SAX
            StringWriter out = new StringWriter();
            XMLOutputFactory xof = XMLOutputFactory.newInstance();
            XMLStreamWriter xsw;
            xsw = xof.createXMLStreamWriter(out);
            xsw.writeStartDocument();
            xsw.writeStartElement("empresa");
            xsw.writeAttribute("CIF", "A34246801");
            xsw.writeCharacters("MegaExport");
            xsw.writeStartElement("sedes");
            xsw.writeStartElement("sede");
            xsw.writeCharacters("LEÓN");
            xsw.writeEndElement();
            xsw.writeStartElement("sede");
            xsw.writeCharacters("CÁCERES");
            xsw.writeEndElement();
            xsw.writeEndElement();
            xsw.writeEndElement();
```

```
            xsw.writeEndDocument();
            xsw.flush();
            xsw.close();
            recurso = (XMLResource) col.createResource("DatosEmpresa",
                XMLResource.RESOURCE_TYPE);
            ContentHandler ch = recurso.setContentAsSAX();
            XMLReader reader = XMLReaderFactory.createXMLReader();
            reader.setContentHandler(ch);
            reader.parse(
                new InputSource(new StringReader(out.toString())));
            col.storeResource(recurso);
            col.removeResource(col.getResource("Empresa_.xml"));
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                if (col != null) {
                    col.close();
                }
            } catch (XMLDBException xe) {
                xe.printStackTrace();
            }
        }
    }
}
```

Actividad propuesta 7.1

Crea un nuevo documento, en la misma colección que el anterior programa, utilizando `setContentAsDOM`. Primero hay que crear un documento DOM, de la manera en que se vio en el capítulo anterior dedicado a XML. Puedes copiar el código de un programa de ejemplo de este capítulo para crear el documento. Si es necesario, consulta los Javadocs de la API XML:DB.

7.9. El lenguaje XQuery

XQuery es un lenguaje estándar de W3C que se creó para realizar consultas sobre documentos de XML, lo mismo que su antecesor XPath. XPath se explicó en el capítulo dedicado a XML, que incluye numerosas consultas de ejemplo en XPath. XQuery 1.0 se definió de manera que incluye XPath 2.0. Es decir, cualquier sentencia de XPath 2.0 lo es de XQuery 1.0 y devuelve los mismos resultados. Lo mismo para XQuery 3.1 respecto de XPath 3.1.

Recurso web

Se puede encontrar información de XQuery en el sitio web de W3C:

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

Pronto se planteó la necesidad de disponer de un lenguaje para realizar operaciones de modificación sobre documentos de XML. Para ello surgieron diversas propuestas, como extensiones de XQuery, que se citan a continuación en orden cronológico:

a) XUpdate. De 2000. Promovido por el grupo que desarrolló la API XML:DB.
b) XQuery Update Extension. De 2001. Promovido por Patrick Lethi.
c) XQuery Update Facility (XQUF). De 2011. Es un estándar de W3C.

Estas propuestas no solo tienen nombres muy parecidos, sino que también son similares entre sí. Si la base de datos proporciona soporte para ella, es preferible utilizar la última, porque es un estándar de W3C.

PARA SABER MÁS

Debido a la frecuencia de las operaciones de búsqueda por texto en documentos de XML, y a que los documentos de XML contienen muy a menudo lenguaje natural (es decir, texto redactado por humanos para ser leído por humanos), se desarrolló, como estándar de W3C, una extensión de XQuery para facilitar consultas sobre este tipo de textos: XQuery Full Text. Se puede encontrar una concisa descripción de sus posibilidades en la siguiente dirección:

http://docs.basex.org/wiki/Full-Text.

7.9.1. Consultas con XQuery

En este apartado se explica XQuery con unas cuantas consultas de ejemplo. Se recomienda ejecutarlas con eXide (eXist IDE o entorno de desarrollo integrado de eXist). eXide permite navegar por los contenidos de la base de datos, ejecutar consultas de XQuery y ver los resultados. Tiene características muy útiles, como verificación y autocompletado.

Una consulta de XQuery se puede ejecutar sobre todos los documentos de una colección, como se ve en la siguiente consulta, realizada sobre una colección de ejemplo, disponible si se ha instalado "eXist-db Demo Apps". Cada uno de los resultados se obtiene de un documento diferente.

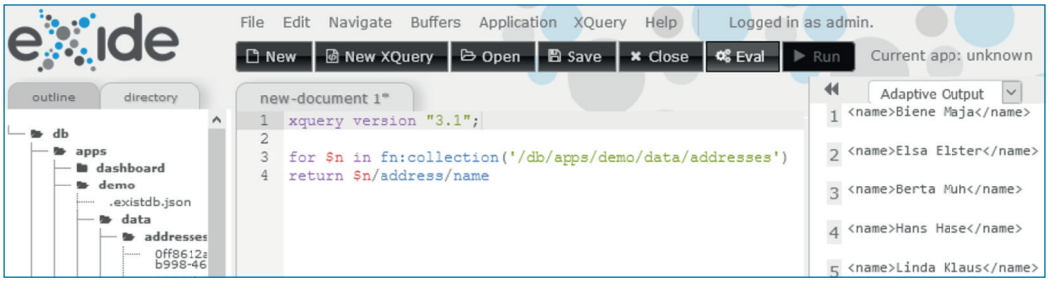


Figura 7.4
Consulta en XQuery sobre todos los documentos de una colección, ejecutada en eXide

Las siguientes consultas de ejemplo se harán en documentos de la colección `pruebas`, creada por un programa de ejemplo anterior. Se harán en general en `Clientes.xml`, pero al-

guna de ellas también en `productos.xml`, que debe crearse previamente con eXide. Se puede enlazar la información de ambos documentos utilizando el DNI.

`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>
```

Las sentencias de XQuery son de tipo FLWOR *(for, let, where, order by, return)*. La analogía con las cláusulas de una consulta de SQL *(select, from, where, group by, having, order by)* es evidente.

CUADRO 7.2
Cláusulas de una sentencia FLWOR de XQuery

FOR	Vincula variables a expresiones en XPath. Estas expresiones pueden devolver múltiples resultados, cada una de las cuales se tiene en cuenta para el resto de las cláusulas.
LET	Permite asignar a una variable el resultado de evaluar una expresión en XPath, y se suele utilizar para evitar repetirla más de una vez. Si la expresión devuelve más de un resultado, se concatenan todos para asignar el resultado a la variable.
WHERE	Especifica condiciones que permiten filtrar los resultados de las cláusulas FOR y LET.
ORDER BY	Permite ordenar los resultados de las cláusulas FOR y LET.
RETURN	Especifica una expresión que se evalúa para cada uno de los resultados proporcionados por el conjunto de cláusulas anteriores. En esta cláusula se pueden generar directamente XML y, en ese caso, se pueden incluir sentencias de XQuery entre llaves `{}`, lo que resulta un mecanismo muy potente.

Actividad propuesta 7.2

Utilizando eXide, crea en la colección `pruebas` el documento `productos.xml` con los contenidos indicados.

A continuación se explican varias sentencias de XQuery de ejemplo. Se sugiere ejecutarlas en eXide y experimentar con variaciones sobre ellas.

`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.

Por lo demás, estos ejemplos solo dan una idea de las posibilidades de XQuery. XQuery tiene muchas funciones predefinidas y se pueden definir nuevas. XQuery tiene una sentencia condicional `if ... then ... else`. Con XQuery se dispone de mucha flexibilidad para añadir más de una vez las anteriores cláusulas y no necesariamente en ese orden. Existe una cláusula GROUP BY similar a la del mismo nombre en SQL, y funciones de grupo `count, sum, min, max`, etc.

Recurso web

www

El siguiente tutorial de la empresa Altova explora todas estas posibilidades:

https://www.altova.com/training/xquery3

7.9.2. Sentencias de modificación de datos con XQuery

En este apartado se explican, con algunos ejemplos, las sentencias de modificación de datos disponibles para eXist, pertenecientes a lo que en su documentación se denomina *XQuery Update*

Extension. Estos ejemplos modifican el documento `Clientes.xml` de la colección `pruebas`, y se recomienda ejecutarlos con eXide. Otras bases de datos pueden utilizar distintas extensiones de XQuery, pero el planteamiento y la sintaxis son similares.

`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.

7.10. La API XQJ

XQJ es a XQuery y las bases de datos de XML lo que JDBC a SQL y las relacionales. Si JDBC permite ejecutar sentencias de SQL desde programas en Java, XQJ hace lo propio con XQuery

y sus extensiones. La API XQJ proporciona una serie de interfaces que los *drivers* para diferentes bases de datos deben implementar. XQJ no forma parte de la biblioteca estándar de clases de Java.

Para operaciones de consulta, el planteamiento de XQJ (y de XML:DB) es similar al de JDBC para bases de datos relacionales. Para operaciones de consulta con JDBC, se ejecuta una sentencia SELECT de SQL, y se obtiene un `ResultSet`, del que se consiguen los resultados mediante un iterador. Con XQJ se hace algo similar, pero las consultas se hacen con XQuery.

Las operaciones de actualización son más problemáticas porque, como ya se ha visto anteriormente, se han propuesto varios lenguajes para este tipo de operaciones, si bien todos son similares y se han planteado como extensiones de XQuery.

Recursos web

www

En la siguiente dirección están disponibles los Javadocs de XQJ:

http://xqj.net/javadoc

En las siguientes direcciones se pueden descargar *drivers* de XQJ para eXist, BaseX, Sedna y Marklogic. El fichero de la descarga es un fichero comprimido que contiene varios ficheros jar que hay que incluir en el proyecto. Entre ellos hay al menos uno con la API XQJ y otro con el *driver* para la base de datos que contiene clases que implementan las interfaces de XQJ:

http://xqj.net/exist
http://xqj.net/basex
http://xqj.net/sedna
http://xqj.net/marklogic

En cada una de las páginas anteriores hay a la derecha un enlace para descargar el *driver*. Hay también un enlace: Compliance Definition Statement. Como explica allí al principio, para la conformidad con el estándar XQJ, se requiere una declaración de cómo se han implementado todos los aspectos de la especificación XQJ que en ella se describen como dependientes de la implementación. Entre ellos está el soporte para transacciones.

Actividad propuesta 7.3

¿Cuáles de los *drivers* XQJ anteriores proporcionan soporte para transacciones?

7.10.1. Establecimiento de conexiones con XQJ

Para crear un programa en Java que utilice XQJ para trabajar con una base de datos, deben añadirse al proyecto los ficheros jar incluidos en la descarga del *driver* para la base de datos.

En la figura se muestra un ejemplo con la base de datos eXist. El fichero `xqjapi.jar` contiene la API XQJ. Para establecer una conexión debe, primero, crearse una instancia de la clase que implementa la interfaz `XQDataSource`. En este caso es `ExistXQDataSource`, en `exist-xqj-1.0.1.jar`, que contiene la implementación de XQJ para eXist. Después, se debe proporcionar un valor para los parámetros de conexión necesarios para la base de datos en particular, con `setProperty`. El nombre de la clase y los parámetros de conexión son aspectos dependientes de la implementación y, por tanto, están documentados en la Compliance Definition Statement del *driver*. Se indican en el cuadro 7.3 para varias bases de datos. No es siempre necesario proporcionar valor para todas las propiedades. La propiedad `description`, por ejemplo, es siempre opcional. En algunos casos, si no se indica un puerto, se puede asumir uno por defecto. Para eXist, por ejemplo, si se indica *localhost* para `serverName`, no es necesario indicar el puerto. Por último, al igual que con XML:DB, con eXist nunca se producirá ningún error ni excepción al abrir la conexión a la base de datos desde el programa, porque esta no se establece hasta que no se realiza alguna operación que requiere el acceso efectivo a la base de datos.

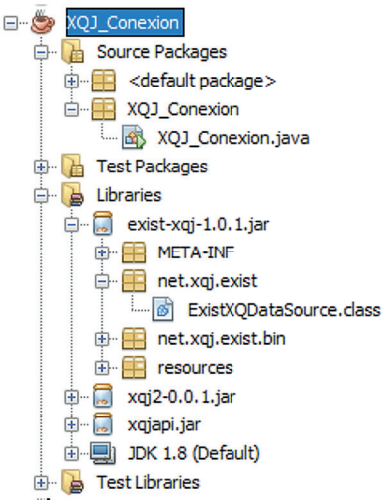


Figura 7.5
Proyecto que usa la API XOJ

CUADRO 7.3
Cadenas de conexión para *drivers* de XQJ para distintas bases de datos nativas de XML

SGBD	Clase que implementa XQDataSource	Propiedades para conexión
MarkLogic	`net.xqj.marklogic.MarkLogicXQDataSource`	`serverName, databaseName, port, user, password, description, mode`
eXist	`net.xqj.exist.ExistXQDataSource`	`serverName, port, user, password, description`
BaseX	`net.xqj.basex.BaseXXQDataSource`	`serverName, databaseName, port, user, password, description`
Sedna	`net.xqj.sedna.SednaXQDataSource`	`serverName, databaseName, port, user, password, description`

En el cuadro 7.3 se indica la clase que implementa la interfaz `XQDataSource` y las propiedades disponibles para la conexión. No es obligatorio proporcionar un valor para todas.

El siguiente programa establece una conexión con una base de datos eXist, indica si la conexión proporciona soporte para transacciones, y cierra la conexión. Para establecer la conexión, se ha creado el método `obtenConexion`, y para mostrar información específica de una excepción de tipo `XQException`, el método `muestraErrorXQuery`. Para averiguar si hay soporte para transacciones, se consultan los metadatos de la conexión, obtenidos con `getMeta-`

`Data`. Para que este programa funcione para otra base de datos, hay que asignar a `nomClaseDS` el nombre de la clase que implementa la interfaz `XQDataSource` para ella.

```
// Abre conexión con XQJ
package ConexionXQJ;

import javax.xml.xquery.XQDataSource;
import javax.xml.xquery.XQConnection;
import javax.xml.xquery.XQException;
import javax.xml.xquery.XQMetaData;
public class ConexionXQJ {

    private static String nomClaseDS = "net.xqj.exist.ExistXQDataSource";

    private static XQConnection obtenConexion() throws ClassNotFoundException,
        InstantiationException, IllegalAccessException, XQException {
        XQDataSource xqs=(XQDataSource) Class.forName(nomClaseDS).
            newInstance();
        xqs.setProperty("serverName", "localhost");
        xqs.setProperty("port", (puerto));
        xqs.setProperty("user", (usuario));
        xqs.setProperty("password", (contraseña));
        return xqs.getConnection();
    }

    private static void muestraErrorXQuery(XQException e) {
    System.err.println("XQuery ERROR mensaje: " + e.getMessage());
        System.err.println("XQuery ERROR causa: " + e.getCause());
        System.err.println("XQuery ERROR código: " + e.getVendorCode());
    }

    public static void main(String[] args) {
        XQConnection c = null;
        try {
            c = obtenConexion();
            XQMetaData xqmd = c.getMetaData();
            System.out.println("Conexión establecida como:"+xqmd.
                getUserName()+".");
            System.out.println(
            "Transacciones: "+(xqmd.isTransactionSupported() ? "Sí":
                "No")+".\n"
            );
        } catch (XQException e) {
            muestraErrorXQuery(e);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                if (c != null) {
                    c.close();
                }
            } catch (XQException xe) {
                xe.printStackTrace();
            }
        }
    }
}
```

7.10.2. Consultas con XQJ

La forma de realizar consultas con XQJ es similar a JDBC. A partir de una conexión `XQConnection` se obtiene una expresión de XQuery `XQExpression` sobre la que se ejecuta una consulta con `executeQuery(String query)`, y se obtiene un conjunto de resultados `XQResultSequence`, análogo a un `ResultSet` de JDBC. Sobre este se itera utilizando el método `next()`. Los resultados se pueden obtener en forma de `XmlStreamReader` con `getItemAsStream()`, o en forma de `Node` del modelo DOM con `getNode()`.

El siguiente programa de ejemplo recupera los apellidos de todos los clientes presentes en el documento `Clientes.xml` de la colección `/db/pruebas`.

```
// Consulta con XQJ

package XQJ_Consulta;

import javax.xml.xquery.XQDataSource;
import javax.xml.xquery.XQConnection;
import javax.xml.xquery.XQException;
import javax.xml.xquery.XQResultSequence;
import javax.xml.xquery.XQExpression;
import javax.xml.stream.XMLStreamReader;
import javax.xml.stream.XMLStreamConstants;

public class XQJ_consulta {

    private static String nomClaseDS = "net.xqj.exist.ExistXQDataSource";
    private static XQConnection obtenConexion() throws ClassNotFoundException,
        InstantiationException, IllegalAccessException, XQException {
        (...)
    }
    private static void muestraErrorXQuery(XQException e) {
        (...)
    }

    public static void main(String[] args) {
        XQConnection c = null;
        try {
            c = obtenConexion();
            String cad = "doc('/db/pruebas/Clientes.xml')/clientes/cliente/
                apellidos";
            XQExpression xqe = c.createExpression();
            XQResultSequence xqrs = xqe.executeQuery(cad);
                int i=1;
                while (xqrs.next()) {
                System.out.println("[Resultado "+(i++)+"]");
                XMLStreamReader xsr = xqrs.getItemAsStream();
                while (xsr.hasNext()) {
                    if (xsr.getEventType() == XMLStreamConstants.CHARACTERS) {
                        System.out.println(xsr.getText());
                    }
                xsr.next();
                }
            }
        } catch (XQException e) {
            muestraErrorXQuery(e);
```

```
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                if (c != null) {
                    c.close();
                }
            } catch (XQException xe) {
                xe.printStackTrace();
            }
        }
    }
}
```

Actividad propuesta 7.4

Modifica el programa anterior para obtener los resultados de la consulta como nodos del modelo DOM mediante el método `getNode()`. Escribe los resultados en forma de árbol utilizando el método `muestraNodo(Node nodo, int nivel, PrintStream ps)` de un programa de ejemplo del capítulo anterior dedicado a XML. Si es necesario, consulta los Javadocs de la API XQJ.

7.10.3. Modificaciones de documentos con XQJ

La ejecución de sentencias para modificación se hace también con una `XQExpression`, pero con el método `executeCommand`. El siguiente programa de ejemplo inserta un nuevo elemento con los datos de un nuevo cliente, en la posición anterior (cláusula `preceding`) a una determinada, indicada mediante un XPath.

```
// Operaciones de modificación con XQJ
package XQJ_Modificacion;
import javax.xml.xquery.XQDataSource;
import javax.xml.xquery.XQConnection;
import javax.xml.xquery.XQException;
import javax.xml.xquery.XQExpression;
public class XQJ_Modificacion {
    private static String nomClaseDS = "net.xqj.exist.ExistXQDataSource";
    private static XQConnection obtenConexion() throws ClassNotFoundException,
        InstantiationException, IllegalAccessException, XQException {
        (...)
    }
    private static void muestraErrorXQuery(XQException e) {
        (...)
    }
```

```
    public static void main(String[] args) {
        XQConnection c = null;
        try {
            c = obtenConexion();
            String cad = "update insert "
                + "<cliente DNI=\"09876543K\">"
                + "<apellidos>LAMIQUIZ</apellidos>"
                + "<CP>43001</CP>"
                + "</cliente>"
                + " preceding doc('/db/pruebas/Clientes.xml')/clientes/cliente/
                   apellidos[text()='SAMPER']/..";
            XQExpression xqe = c.createExpression();
            xqe.executeCommand(cad);
        } catch (XQException e) {
            muestraErrorXQuery(e);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
            if (c != null) {
                c.close();
            }
            } catch (XQException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Actividades propuestas

7.5. Escribe un programa que cambie el nombre del cliente añadido por el programa de ejemplo anterior. La sentencia utilizada debe localizar el cliente que se va a borrar mediante su DNI. Precaución: escribe una consulta en XQuery para recuperar el nombre de este cliente y después transfórmala en una sentencia que lo cambie.

7.6. Escribe un programa que borre el cliente añadido por el programa de ejemplo anterior. La sentencia utilizada debe localizar el cliente que se va a borrar mediante su DNI. Precaución: escribe una consulta en XQuery para recuperar la información de este cliente y después transfórmala en una sentencia que la borre.

7.10.4. Transacciones con XQJ

Una transacción es un conjunto de operaciones que se ejecutan como un todo, y que cumplen los requisitos ACID (atomicidad, consistencia, aislamiento y durabilidad). El concepto de transacción se explicó en detalle en el capítulo anterior dedicado a bases de datos relacionales. Para que en una aplicación se puedan utilizar transacciones, debe proporcionar soporte para ellas el

SGBD, la API y el *driver*. Ya se ha comentado que, al contrario de lo que sucede con los SGBD relacionales, solo algunos SGBD de XML nativos proporcionan soporte para transacciones, y eXist no es uno de ellos. XQJ sí permite utilizar transacciones siempre que el SGBD y el *driver* lo permitan, y además de manera muy similar a JDBC.

Al igual que en JDBC, en XQJ, por defecto, los cambios realizados por cada operación realizada sobre una conexión se confirman en la base de datos al finalizar la operación. Este comportamiento se puede cambiar con `setAutoCommit(false)` si hay soporte para transacciones, y entonces solo se confirman los cambios cuando se ejecuta `commit()`. Se pueden deshacer todos los cambios realizados en el curso de una transacción ejecutando `rollback()`. De acuerdo con la especificación XQJ, si se hace `setAutoCommit(false)` sobre una conexión para la que no hay soporte de transacciones, la implementación (léase el *driver*) debe lanzar un error. Si se produce alguna excepción durante la ejecución de una transacción, esta debe capturarse y debe ejecutarse el método `rollback` para deshacer los cambios. Todo esto se refleja en el siguiente código de programa para implementar una transacción:

```
// Transacción con XQJ

package XQJ_Transaccion;

import javax.xml.xquery.XQDataSource;
import javax.xml.xquery.XQConnection;
import javax.xml.xquery.XQException;
import javax.xml.xquery.XQExpression;

public class XQJ_Transaccion {
    // Sedna soporta transacciones.
    private static String nomClaseDS = "net.cfoster.sedna.DatabaseImpl";

    private static XQConnection obtenConexion() throws ClassNotFoundException,
        InstantiationException, IllegalAccessException, XQException {
        (...)
    }
    private static void muestraErrorXQuery(XQException e) {
        (...)
    }

    public static void main(String[] args) {
        XQConnection c = null;
        try {
            c = obtenConexion();
            c.setAutoCommit(false);
            XQExpression xqe = c.createExpression();
            // Ahora se pueden ejecutar múltiples sentencias de modificación,
            // entre las que se pueden intercalar sentencias de consulta con
            // executeQuery
            xqe.executeCommand("update... ");
            xqe.executeCommand("update... ");
            (...)
            c.commit();
        } catch (XQException e) {
            muestraErrorXQuery(e);
            try {
                c.rollback();
                System.err.println("Se hace ROLLBACK");
```

```
            } catch (XQException er) {
                System.err.println("ERROR haciendo ROLLBACK");
                muestraErrorXQuery(er);
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
            if (c != null) {
                c.close();
            }
            } catch (XQException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Resumen

- XML es un formato muy flexible para el almacenamiento e intercambio de información. Se pueden distinguir, a grandes rasgos, dos áreas de aplicación de XML: XML centrado en documentos *(document-centric XML)* y XML centrado en datos *(data-centric XML)*.
- Las bases de datos de XML nativas almacenan los documentos de XML en estructuras diseñadas y optimizadas para documentos de XML, y además proporcionan soporte para tecnologías de XML tales como XPath, XQuery y XSL.
- Existen también bases de datos no nativas de XML pero con capacidades para *XML (XML-enabled)*. Un ejemplo de ellas son las bases de datos relacionales que implementan SQL/XML, añadido al estándar SQL en SQL:2003. Pueden ser la mejor opción para aplicaciones de XML basadas en datos *(data-centric XML)*.
- Las bases de datos de XML nativas suelen organizar los documentos en colecciones. Algunas permiten definir una jerarquía de colecciones, y algunas permiten almacenar otros tipos de documentos además de documentos de XML.
- Las bases de datos de XML nativas implementan mecanismos de indexación estructural, en los que se tienen en cuenta los valores almacenados en todos los elementos y atributos.
- El principal lenguaje de consulta para bases de datos de XML es XQuery, un estándar de W3C, que es un superconjunto de XPath, otro estándar anterior de W3C. Se han propuesto varias extensiones de XQuery para permitir operaciones de modificación sobre el contenido de los documentos de XML; la última es XQUF, también un estándar de W3C.
- Las API estándares existentes para bases de datos nativas de XML son XML:DB y XQJ. XQJ está concebida para ejecutar consultas de XQuery y también sentencias de modificación de documentos de XML expresadas en alguna de las extensiones de XQuery propuestas para ello, entre ellas el estándar XQUF de W3C.
- Algunas bases de datos nativas de XML proporcionan un buen soporte para transacciones, pero este no es el caso en general, muy al contrario que con las bases de datos relacionales.

Ejercicios propuestos

Todos los documentos que se creen para estas actividades deben crearse en una colección con nombre `ejercicios_bd_XML`, a no ser que se indique otra cosa.

1. Crea un programa genérico al que se le pasen por línea de comandos el nombre de una colección, el nombre de un documento y una consulta en XQuery. El programa ejecutará la consulta y devolverá una lista de resultados. Si no se indica un nombre de documento, la consulta debe realizarse sobre todos los documentos de la colección. Los resultados se deben mostrar en formato XML. Si cada resultado es un nodo de un árbol DOM, debe mostrarse el fragmento del documento por debajo del nodo obtenido. Prueba este programa con las consultas de ejemplo de este capítulo.
2. Crea un programa genérico al que se le pasen por línea de comandos el nombre de una colección, el nombre de un documento y una sentencia de modificación de XQuery para ejecutar sobre el documento en cuestión. El programa debe crear una copia del documento original en la misma colección, en cuyo nombre debe incluirse el nombre del documento original y una marca de tiempo incluyendo la fecha y la hora. Una vez ejecutada la consulta, deben mostrarse los contenidos tanto del documento original (es decir, de la copia que se ha hecho previamente) como del documento resultante tras ejecutar la sentencia. Debe pedirse confirmación al usuario para confirmar los cambios. Si la da, debe borrarse la copia creada. Si no, debe borrarse el documento y renombrar la copia, para que todo quede como al principio. Si se produce cualquier excepción, todo debe quedar como al principio. Prueba este programa con las sentencias de modificación de datos de este capítulo.
3. Crea una clase con varios métodos para consultar datos de un cliente, crear un cliente, borrar un cliente y modificar datos de un cliente (al menos, nombre, código postal y DNI). Los datos de los clientes se deben almacenar en un documento de XML con un formato como el del documento de ejemplo utilizado durante este capítulo. A los métodos que realizan cualquier cambio sobre un cliente se les debe indicar el cliente mediante el DNI. Debe haber un método para obtener el número de clientes almacenados y otro para mostrar todos los datos de todos los clientes, que se puede limitar a escribir los contenidos del documento. El constructor de la clase debe crear el documento para almacenar los datos de los clientes, en caso de que no exista, conteniendo solo un elemento con nombre `clientes`. Debe crearse un programa de prueba en el método `main()` que realice varias operaciones de creación, borrado y modificación de clientes, y por último muestre todos los datos de todos los clientes.
4. Crea una clase con la misma interfaz que la anterior, pero que internamente funcione con un esquema de almacenamiento diferente. A saber, para cada cliente debe haber un documento de XML con sus datos, y todos ellos deben estar en una misma colección, que inicialmente estará vacía. No debería ser necesario que, tras cambiar el DNI de un cliente, sigan coincidiendo el valor del DNI almacenado y el nombre del fichero, pero, opcionalmente, se puede hacer que se cambie el nombre del documento al modificar el DNI. En su defecto, se puede crear un método estático de la clase para renombrar correctamente todos los documentos para que su nombre coincida con el del DNI almacenado. El método `main()` debe ser el mismo que para la anterior actividad.

5. Crea una aplicación que gestione los datos de los empleados de una empresa almacenados en un fichero XML. El fichero debe tener la estructura siguiente:

```
<empresa cif="…" nombre="…">
   <departamento código="…" nombre="…">
      <empleado dni="…" nombre="…"/>
         <puesto>...</puesto>
      </empleado>
      <empleado dni="…" nombre="…"/>
   ...
      <empleado dni="…" nombre="…"/>
         <puesto>...</puesto>
      </empleado>
   </departamento>
   ...
   <departamento código="…" nombre="…">
      <empleado dni="…" nombre="…">
   ...
      </empleado>
   ...
      <empleado dni="…" nombre="…">
   ...
      </empleado>
   </departamento>
</empresa>
```

En resumen, dentro de la empresa hay departamentos y dentro de cada departamento, empleados. No hace falta hacer validaciones sobre el formato de ningún dato. No puede haber dos departamentos distintos con el mismo código, ni dos empleados distintos con el mismo DNI, y ambos atributos son obligatorios. También los atributos `cif` y `nombre` del elemento `empresa`. El elemento `puesto` es opcional. Un empleado no puede estar en más de un departamento.

Debe crearse una clase `GestorEmpresa` que gestione los datos del documento, almacenado en una base de datos de eXist. El documento debe estar en una colección con nombre `empresa_y_empleados`. El constructor debe tener como parámetros el CIF y el nombre de la empresa, y debe crear la colección si no existe, y el documento si no existe, con el CIF y el nombre de la empresa. El método `main()` de la clase contendrá un programa de prueba que utilice esta clase para crear varios departamentos, añadir varios empleados en distintos departamentos y cambiar algunos empleados de departamento, utilizando los métodos que se indican más adelante. Los métodos no deben gestionar ninguna excepción, esto debe hacerlo el programa principal en el método `main()`. Esta clase debe tener métodos para:

- Crear un departamento, dado su código y nombre. Si ya existe un departamento con ese nombre, se debe mostrar un mensaje de error y devolver `false`. En otro caso, se debe devolver `true`.
- Crear un empleado, dado su DNI y nombre y el código del departamento al que pertenece. Si ya existe un empleado con ese DNI, se debe mostrar un mensaje de error y devolver `false`. También si no existe el departamento. En otro caso, se debe devolver `true`.

- Eliminar un empleado dado su DNI. Debe devolver `false` y mostrar un mensaje de error si el empleado no existe. En otro caso, debe eliminar el empleado y devolver `true`.
- Eliminar un departamento dado su código. Debe devolver `false` y mostrar un mensaje de error si el departamento no existe o si tiene algún empleado. En otro caso, debe eliminar el departamento y devolver `true`.
- Cambiar un empleado de departamento, dado el DNI del empleado y el código del nuevo departamento. Debe devolver `true` si se puede hacer la operación. En caso contrario, deben mostrarse mensajes de error y devolver `false`. A saber, si el empleado o el departamento no existen, o si el empleado ya está en el departamento.

El método `main()` podría quedar más o menos así:

```
try {
    GestorEmpresa gEmpresa = new GestorEmpresa(nomFich);
    if(!gEmpresa.creaDepartamento(codDep1,nomDep1)) return;
    if(!gEmpesa.creaEmpleado(dniEmp1,nomEmp1,codDep1)) return;
    if(!gEmpesa.creaEmpleado(dniEmp2,nomEmp2,codDep1)) return;
    if(!gEmpresa.creaDepartamento(codDep2,nomDep2)) return;
    if(!gEmpesa.creaEmpleado(dniEmp3,nomEmp3,codDep2)) return;
    System.out.println("Cambios realizados correctamente");
    }
catch(...) {
...
}
catch(...) {
...
}
```

ACTIVIDADES DE AUTOEVALUACIÓN

1. XML es un formato de documento:
   - ☐ *a)* Con una estructura fija y regular.
   - ☐ *b)* Que tiene dos variantes: una para almacenar datos y otra para textos en lenguaje natural.
   - ☐ *c)* Muy sencillo y flexible, pero con el que se pueden también crear documentos con una estructura fija y regular.
   - ☐ *d)* Con una estructura que puede ser jerárquica o tabular.

2. El almacenamiento de documentos de XML en bases de datos relacionales:
   - ☐ *a)* Es siempre una mala alternativa, al tener los documentos de XML una estructura jerárquica que no se presta al almacenamiento en estructuras tabulares.

- ☐ *b)* Es la única manera de poder disponer de transacciones en aplicaciones que trabajan con datos en formato XML.
- ☐ *c)* Solo es posible si se almacenan los documentos como un todo en campos de tipo BLOB, CLOB o VARCHAR.
- ☐ *d)* Ninguna de las respuestas anteriores es correcta.

3. Las bases de datos de XML nativas:
   - ☐ *a)* Son todas de código abierto, al ser XML un estándar de W3C.
   - ☐ *b)* Son las únicas que permiten utilizar tecnologías de XML tales como XML Schema, XPath, XQuery y XSL, sobre los documentos de XML almacenados.
   - ☐ *c)* Suelen organizar los documentos en colecciones.
   - ☐ *d)* Solo permiten almacenar documentos en formato XML.

4. Las colecciones en las bases de datos de XML nativas:
   - ☐ *a)* Son jerárquicas, es decir, puede haber colecciones dentro de colecciones.
   - ☐ *b)* Deben siempre crearse explícitamente antes de añadir documentos a ellas.
   - ☐ *c)* Deben contener siempre documentos de igual tipo para toda la colección.
   - ☐ *d)* Ninguna de las respuestas anteriores es correcta.

5. SQL/XML:
   - ☐ *a)* Es un lenguaje de consulta para bases de datos de XML inspirado en SQL.
   - ☐ *b)* Es una extensión para XML incluida en todas las versiones de Oracle desde la 12c.
   - ☐ *c)* Es una parte del estándar SQL que añade soporte para XML.
   - ☐ *d)* Es un SGBD comercial multimodelo, a la vez relacional y de XML nativo.

6. Los índices de las bases de datos de XML nativas:
   - ☐ *a)* Son más costosos de mantener que los de las bases de datos relacionales.
   - ☐ *b)* No son actualizados automáticamente en algunos SGBD de XML nativos.
   - ☐ *c)* Son estructurales, basados en los valores de todos los elementos y atributos.
   - ☐ *d)* Todas las respuestas anteriores son correctas.

7. Las transacciones en los SGBD de XML nativos:
   - ☐ *a)* No siempre están soportadas, pero si no lo están, se pueden conseguir con un *driver* de XQJ, porque el soporte para transacciones es obligatorio en XQJ.
   - ☐ *b)* No están soportadas por muchos de ellos, al contrario de lo que sucede con los SGBD relacionales.
   - ☐ *c)* Están en general soportadas, pero, como son muy costosas computacionalmente, suelen estar deshabilitadas por defecto.
   - ☐ *d)* Solo están soportadas por la API XQJ, pero no por XML:DB.

8. El lenguaje XQuery:
   - ☐ *a)* Es un subconjunto del lenguaje XPath.
   - ☐ *b)* Permite realizar consultas sobre varios documentos de XML, siempre que estén en la misma colección.
   - ☐ *c)* Obliga a incluir las cláusulas de una sentencia FLWOR en un orden fijo, al igual que sucede con las cláusulas de una sentencia SELECT de SQL.

☐ *d)* Permite la anidación de consultas FLWOR, es decir, el uso de una consulta FLWOR dentro de una cláusula de una consulta FLWOR.

9. La API XML:DB:
   - ☐ *a)* No permite realizar consultas con XQuery.
   - ☐ *b)* Está obsoleta y se utiliza cada vez menos, en especial para nuevos desarrollos.
   - ☐ *c)* No proporciona servicios para transacciones.
   - ☐ *d)* Permite realizar tareas administrativas y de gestión que XQJ no permite.

10. La API XQJ:
   - ☐ *a)* No permite realizar operaciones de modificación sobre documentos de XML.
   - ☐ *b)* Es parte de las bibliotecas estándares de Java (*Java standard libra*ries).
   - ☐ *c)* Permite obtener los resultados de una consulta como nodos de DOM.
   - ☐ *d)* Permite borrar documentos enteros de una colección.

SOLUCIONES:

1. c	**5.** c	**9.** d	
2. d	**6.** d	**10.** c	
3. c	**7.** b		
4. d	**8.** d		