

TEMA

Tema 5. Persistencia en BBDD nativas XML

Desarrollo de aplicaciones
multiplataforma

Acceso a datos

Autora: Silvia Macho



Tema 5: Persistencia en BBDD nativas XML

¿Qué aprenderás?

- Instalar y configurar el SGBD XML nativo eXist.
- Crear una aplicación Java usando eXist como elemento de persistencia.

¿Sabías que...?

- eXist es un SGBD nativo XML por lo que guarda directamente como recursos los documentos XML.



5.1. Introducción

Cualquier aplicación informática necesita de un elemento de persistencia para poder hacer permanentes los datos con los que trabaja. En su momento, utilizábamos los ficheros como elemento de persistencia, que evolucionaron a los SGBD, que es lo que utilizan las aplicaciones actuales.

Los SGBD también han ido evolucionando y adaptándose a los nuevos tipos de datos. Tanto es así, que tenemos SGBD jerárquicos, relacionales, objeto-relacionales, orientados a objetos y XML nativos. Esto hace que el equipo que desarrolla la aplicación tenga que saber qué sistema de persistencia utilizará la aplicación.

A la hora de desarrollar una aplicación, tenemos que tomar varias decisiones. Por un lado, decidir el lenguaje de programación para el desarrollo de la aplicación y por otro el sistema de persistencia que vamos a utilizar, es decir el SGBD. Como son dos elementos que tienen que trabajar conjuntamente, tenemos que asegurarnos que el lenguaje de programación seleccionado tenga una API de acceso al SGBD.



5.2. XML

XML es un formato simple basado en texto que permite representar información de forma estructurada. Actualmente es uno de los formatos más utilizado para el intercambio de información. Esto ha ocasionado la aparición de mecanismos que permitan guardar dicha información en XML, de forma que no sea necesario realizar traducciones entre diferentes formatos.

El almacenamiento de los datos en formato XML no se realiza de una única forma, sino que han aparecido diferentes estrategias, una de ellas son las BBDD XML nativas.

5.2.1. Estrategias de almacenamiento de un XML

XML es un lenguaje que nos permite tener información estructurada, ya sea para el intercambio de datos entre plataformas o para facilitar el acceso a contenido. Su rápida y amplia adaptación, ha generado la necesidad de pensar mecanismos para poder almacenar de forma ágil, grandes volúmenes de documentos XML.

Para poder decidir cuál es la mejor estrategia, debemos tener en cuenta los tipos de documentos XML que podemos tener:

- Documentos centrados en datos.

Estos documentos están pensados para el intercambio entre diferentes plataformas. Suelen ser documentos con estructuras regulares y bien definidas. Los datos que almacenan son unidades anatómicas bien definidas y en la mayoría de las ocasiones, actualizables. Su origen suele ser una BBDD y por lo tanto en muchas ocasiones no se suelen almacenar como fichero XML.



- Documentos centrados en el documento.

Son documentos con una estructura irregular. El origen y destino están en las personas y por lo tanto suelen crearse manualmente. En alguna ocasión pueden tener formato, pero éste no es estricto ni definido y por lo tanto hablamos de datos semi-estructurados.

La clasificación en documentos centrados en datos o en el documento no es siempre clara y directa y, en muchas ocasiones los datos estarán mezclados o será difícil de catalogar en un tipo concreto. Decidir entre una opción u otra la utilizaremos para decidir cuál será la mejor estrategia de almacenamiento.

A partir de aquí, lo que tenemos que hacer es saber qué tipos de almacenamiento tenemos relacionado con los documentos XML para decidir cuál es la técnica más adecuada en función de la tipología de los documentos y la gestión que queremos hacer de ellos.

Existen 3 opciones para almacenar documentos XML:

- Almacenamiento directo en un archivo.

Es la opción más directa pero la más pobre, ya que las opciones que tenemos dependen directamente del sistema operativo en el que estamos almacenando el fichero. No nos permite hacer operaciones sobre el contenido del archivo y solo se nos permite el movimiento del fichero como una unidad. Cuando trabajamos con pocos documentos XML formados por pocos datos, puede ser una opción, pero para un volumen elevado, mejor considerar otras opciones.

- Almacenamiento en una BBDD tradicional.

Cuando hablamos de BBDD tradicional, estamos haciendo referencia a BBDD relacionales, jerárquicas u orientadas a objetos. Esta opción nos obliga a tener que transformar el documento XML al modelo de datos que se corresponda con la BBDD utilizada.



- Almacenamiento en una BBDD XML nativa.

Este tipo de BBDD nos permiten almacenar directamente los documentos XML tal cual, sin ningún tipo de transformación. Están diseñadas especialmente para el almacenamiento de XML.

5.2.2. BBDD XML nativas

Una BBDD XML nativa es una BBDD diseñada especialmente para el almacenamiento de XML. Su abreviatura en castellano es BD-XML, en inglés es NXD.

Normalmente, los documentos centrados en datos los almacenamos en SGBD tradicional, ya que es más fácil hacer las transformaciones necesarias para adaptarse a la BBDD, en cambio los documentos centrados en documentos, los solemos almacenar en una BBDD XML nativa, pero esto no es una regla absoluta, siempre dependerá de cada caso y de cada aplicación.

Un SGBD XML nativo, es un sistema que debe:

- Definir un modelo lógico para un documento XML y permitir el almacenamiento y recuperación de los documentos basándose en ese modelo. Como mínimo, el modelo tiene que incluir elementos, atributos, PCDATA y el orden del documento.
- Mantener una relación transparente con el mecanismo de almacenamiento, incorporando las características ACID de cualquier SGBD.
- Incluir un número arbitrario de niveles de datos y complejidad.
- Permitir las tecnologías de consulta y transformación propias de XML: XPath, XSLT, XQL, XQuery, etc, como vehículo principal de consulta y gestión.
- Permitir la introducción de la información tanto centrada en datos, centrada en documentos y de forma mixta.



Por lo tanto, los SGBD XML nativos son la alternativa natural para las aplicaciones que trabajan con documentos XML en todos los niveles: interfaz, gestión y almacenamiento de datos. Igual que ocurre con el resto de SGBD, las BBDD XML nativas soportan transacciones, acceso de múltiples usuarios, tiene lenguaje de consultas, índices, etc. Ejemplos de estos sistemas son eXist y BaseX, entre otras.

5.2.3. eXist

eXist es una alternativa de código libre, multiplataforma, nativa XML para almacenar y recuperar documentos XML, ya que los almacena en una estructura propia sin necesidad de hacer uso de otro sistema gestor que requiera una transformación.

Al igual que la mayoría de sistemas XML nativos, estructura los documentos en colecciones. Una colección es un conjunto de documentos, de modo que forma una estructura de árbol donde cada documento pertenece a una única colección. Dentro de una colección puede almacenar documentos de cualquier tipo. Dentro de una colección, no solo podemos tener documentos XML, también podemos almacenar otro tipo de documentos de texto o binarios. A estos documentos, los llamaremos documentos no XML.

A diferencia de otros SGBD XML nativos, en eXist los documentos no tienen que tener un DTD o XML Schema asociado, por lo que solo ofrece funcionalidad para documentos XML bien formados, sin atender a si siguen o no una estructura.



El almacén central nativo de datos es el fichero dom.dbx, que es un fichero paginado donde se almacenan todos los nodos del documento siguiendo el modelo DOM de W3C. Dentro del mismo fichero también existe un árbol secundario que asocia el identificador único del nodo con su posición. En el archivo collections.dbx, se almacena la jerarquía de colecciones y relaciona esta con los documentos que contiene, se asigna un identificador único a cada documento de la colección que es almacenado junto al índice.

eXist indexa de forma automática todos los documentos utilizando índices numéricos para identificar sus nodos, que son elementos, atributos, texto y comentarios. Durante la indexación se asigna un identificador único de cada documento de la colección, el cual es almacenado también junto al índice. Para ahorrar espacio, los nombres de los nodos no son utilizados para construir el índice, en su lugar se asocia el nombre del elemento y el de los atributos con unos identificadores numéricos en una tabla de nombres.

eXist indexa por defecto todos los nodos de texto y valores de atributos dividiendo el texto en palabras. Toda esta información se almacena en el fichero words.dbx.

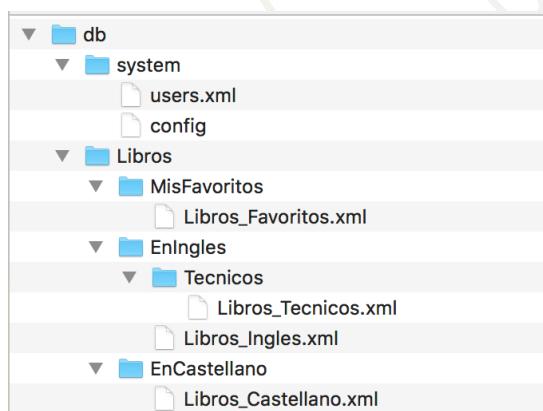
eXist proporciona diversas API de acceso desde Java, cada una de ellas la podremos utilizar en un determinado caso, por lo que al realizar la aplicación con Java y eXist, tendremos que seleccionar la API más apropiada.



5.2.4. Colecciones y documentos en eXist

De la misma forma que un documento XML sigue una estructura de árbol, los SGBD XML nativos tienen esa misma estructura para organizar los documentos que almacenan. Es un modelo parecido al sistema de archivos de un sistema operativo donde cada las carpetas parten de una raíz y pueden tener subcarpetas, y dentro de cada una de las carpetas es donde tenemos los archivos almacenados. Lo mismo hacen los SGBD XML nativos, donde las colecciones son el equivalente a las carpetas y dentro de ellas tenemos los documentos o recursos (que es como los llamaremos siempre que trabajemos con un SGBD XML nativo). Estos recursos pueden ser XML o no XML (texto o binarios).

Un ejemplo de colección en eXist es la siguiente:



Como podemos ver en la imagen, la colección raíz que contiene al resto es db. De esta colección siempre colgarán el resto, ya sea directamente o como subcolecciones.



En el ejemplo, en db tenemos dos colecciones:

- system

Contiene las colecciones y documentos necesarios para la configuración de eXist. Esta colección es la que utiliza el sistema gestor para realizar las tareas de administración, por lo tanto, no está disponible para su uso por parte de los usuarios.

- Libros

Esta colección la hemos creado específicamente con un conjunto de recursos como ejemplo para esta documentación.

Dentro de Libros, tenemos 3 colecciones:

- MisFavoritos

En esta colección tenemos un recurso XML llamado Libros_Favoritos.xml.

- EnIngles

En esta colección tenemos una colección hija y un recurso XML llamado Libros_Ingles.xml. La colección hija se llama Tecnicos y está formada por el recurso XML Libros_Tecnicos.xml

- EnCastellano

En esta colección tenemos un recurso XML llamado Libros_Castellano.xml

A partir de aquí, para poder acceder tanto a las colecciones como a los recursos que contienen, utilizaremos la misma sintaxis que con XPath. Por ejemplo, para acceder a la colección MisFavoritos, la ruta sería:

`/db/Libros/MisFavoritos`

Para acceder al recurso de esta misma colección, la ruta sería:

`/db/Libros/MisFavoritos/LibrosFavoritos.xml`

Lo mismo ocurre para acceder a cualquier elemento dentro de un recurso XML.



Por ejemplo, si el documento LibrosFavoritos.xml tiene la siguiente estructura:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Base de datos de libros favoritos pero con estructura diferente -->
<Favoritos>
    <Libro paginas="50">
        <Autor>Nikolai Gogol</Autor>
        <Titulo>El Capote</Titulo>
    </Libro>
    <Libro paginas="730">
        <Autor>Gonzalo Giner</Autor>
        <Titulo>El Sanador de Caballos</Titulo>
    </Libro>
    <Libro paginas="450">
        <Autor>Umberto Eco</Autor>
        <Titulo>El Nombre de la Rosa</Titulo>
    </Libro>
</Favoritos>
```

Para acceder a un elemento, la ruta sería la siguiente:

/db/Libros/MisFavoritos/Favoritos/Libro



5.3. Acceso a eXist desde Java

Como hemos comentado anteriormente, una aplicación está formada por la aplicación como tal creada usando un determinado lenguaje de programación y un SGBD que permita hacer los datos de la aplicación persistentes. En el caso en el que se centra esta documentación, el lenguaje de programación es Java y el SGBD es eXist. Por otro lado, para que el conjunto funcione, es necesario que el SGBD proporcione una API para el lenguaje de programación que le permita poder acceder y utilizar la BBDD. Para el caso de Java y eXist, existen diferentes APIs que lo permiten, nos centraremos en dos: XML-DB y XQJ.

5.3.1. XML-DB

XML:DB es la librería de Java de acceso a sistemas XML nativos más conocida y utilizada. La mayoría de los sistemas soportan esta librería.

El objetivo de XML-DB es la definición de una forma común de acceso a SGBD XML nativos permitiendo la consulta y modificación de contenido desde la aplicación Java. Podemos considerar XML-DB como el equivalente en SGBD XML nativos a JDBC.

Como todas las APIs de acceso a datos, XML-DB depende de lo que cada sistema gestor implemente. Su estructura depende:

- Driver: encapsula la lógica de acceso a una BBDD. Cada sistema gestor tiene que implementar este driver.
- Collection: clase que representa el concepto de colección visto en los apartados anteriores.
- Resource: clase que representa el concepto de recurso visto en los apartados anteriores. Los recursos pueden ser de dos tipos: XMLResource, BinaryResource.
- Service: implementación de una funcionalidad que extiende el núcleo de la especificación.



5.3.2. XQJ

XQJ (XQuery API for Java) es una opción más actual que XML-DB para ser el estándar de acceso a SGBD XML nativos. XQJ intenta acercarse todo lo posible del modelo de JDBC, especificando una estructura de clases Java con sintaxis similar a la de la API JDBC y que sigue una filosofía basada en un origen de datos al que podemos conectar un cliente, y a partir de una conexión, ejecutar consultas.

Las clases más significativas de XQJ que debe implementar todo SGBD que quiera ser compatible con XQJ son:

- **XQDataSource**: identifica una fuente de datos a partir de la cual crearemos las conexiones a la BBDD. Cada implementación definirá las propiedades necesarias para crear la conexión, siendo las más habituales usuario y contraseña.
- **XQConnection**: representa una sesión con la BBDD, manteniendo la información de estado, transacciones, expresiones ejecutadas y resultados. Obtenemos este objeto a partir de **XQDataSource**.
- **XQExpression**: objeto creado a partir de una conexión para la ejecución de una expresión una vez. Cuando ejecutamos esta expresión, devuelve un objeto **XQResultSequence** con los datos obtenidos como resultado. La ejecución la realizamos con el método `executeQuery`.
- **XQPreparedExpression**: objeto creado a partir de una conexión para ejecutar expresiones múltiples veces. Lo que devuelve es un **XQResultSequence** con los datos obtenidos como resultado. Igual que **XQExpression**, la ejecución la realizamos con el método `executeQuery`.
- **XQResultSetSequence**: resultado de la ejecución de una sentencia y contiene un conjunto de 0 o más **XQResultItem**.



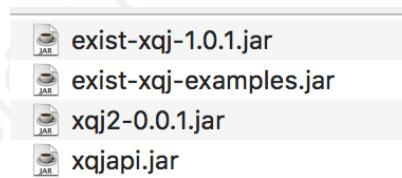
5.3.3. Uso de las APIs XML-DB y XQJ

Una diferencia importante entre XQJ y XML-DB es que el primero no se preocupa de las colecciones que puedan haber creadas en el sistema gestor. Tal y como pasa en una BBDD relacional, XQJ establece una conexión y después a través de esta conexión, accederemos desde la aplicación. En los SGBD relativos, el lenguaje que utilizamos es SQL, mientras que en sistemas XQJ es XQuery. Esto hace que XQJ ofrezca un nivel de abstracción mayor que XML-DB, ya que solo se preocupa de los datos almacenados en los recursos y su estructura dentro del recurso, pasando por alto la estructura de colecciones con la que hayamos estructurado la BBDD.

Para que una aplicación Java pueda utilizar las APIs XML-DB y XQJ, tenemos que incluir las librerías correspondientes en el proyecto de la aplicación. De esta forma, nuestra aplicación podrá utilizar las clases y métodos de la correspondiente API.

Las librerías incluidas para soportar la implementación de XML-DB y XQJ en eXist son las siguientes:

- XQJ: las librerías que tenemos que incluir en los proyectos Java son XQJapi.jar, XQJ2-0.0.1.jar, eXist-XQJ-1.0.1.jar y eXist-XQJ-examples.jar.



Estas librerías las podemos encontrar en la carpeta lib de la instalación de eXist.

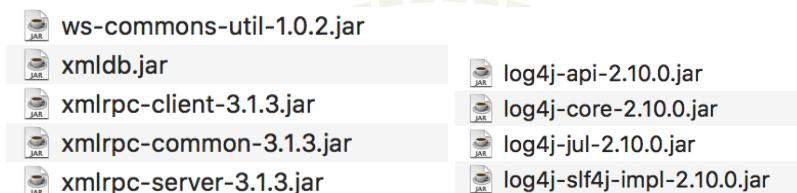


- XML-DB: las librerías necesarias las podemos encontrar en dos directorios diferentes de la instalación de eXist:

- eXist/eXist



- eXist/eXist/lib/core



5.3.4. Instalación de eXist

La instalación de eXist es muy sencilla. Descargamos la versión de su página web oficial <http://exist-db.org/>. Una vez hemos realizado la descarga, podemos iniciar el proceso de instalación. Nos pedirá la versión del JDK a utilizar, la ruta donde se instalará y el nombre de usuario/contraseña del usuario administrador con el que accederemos desde el cliente para realizar la administración. Por defecto el usuario es admin y la contraseña también, pero son datos que podemos modificar.



Cuando ya tenemos eXist instalado, podemos poner en marcha el servicio de eXist, para poder acceder desde el cliente para realizar su administración. El cliente por defecto que se instala es eXist Client Shell, programado en Java, que permite el acceso para gestionar las colecciones y recursos que tengamos en el sistema. Previo al acceso nos pedirá usuario y contraseña.



The screenshot shows the 'Cliente de Administración eXist' window. At the top, there's a menu bar with Fichero, Herramientas, Conexión, Opciones, and Ayuda. Below the menu is a toolbar with icons for file operations like Open, Save, and Delete. The main area is a table titled 'Recurso' with columns for Recurso, Fecha, Propietario, Grupo, and Permisos. The table lists several items: Libros (2018-03-31, admin, dba, crwxxxxrwx), apps (2018-03-31, SYSTEM, dba, crwxr-xr-x), biblioteca (2018-04-16, admin, dba, crwxxr-xr-x), sample (2018-04-25, admin, dba, crwxxr-xr-x), and system (2018-03-31, SYSTEM, dba, crwxr-xr-x). Below the table is a command-line interface window with the text: 'type help or ? for help.' and 'exist:/db>'. At the bottom, it says 'Cliente de Administración de eXist conectado - admin@xmlDb:exist://localhost:8080'.

Otra forma de acceder a eXist para realizar su administración es desde un navegador. Para ello, tenemos que poner la IP de la máquina donde hemos instalado el servicio de eXist o localhost en el caso que sea la máquina local e indicar el puerto 8080.

The screenshot shows the 'Dashboard' page of eXist. It features a header with a user dropdown (admin) and a search bar ('localhost:8080/exist/apps/dashboard/index.html'). Below the header are several sections: 'existdb' (Backup, Collections), 'eXide - XQuery IDE', 'Java Admin Client', 'Markdown Parser in XQuery', 'Monitoring and Profiling for eXist (MoneX)', 'Package Manager', 'Shutdown', 'Usermanager', and 'XQuery Function Documentation'.



5.4. Creación de la aplicación Java

Una vez tenemos instalado el SGBD eXist, su servicio funcionando, ya podemos empezar a crear la aplicación en Java. No debemos olvidar las librerías necesarias para que el conjunto funcione correctamente.

Como ocurre con cualquier aplicación que trabaja con un sistema de persistencia, hay una serie de acciones que la aplicación realizará relacionadas solo con la propia persistencia. En este caso, al disponer de dos librerías diferentes, iremos especificando en cada caso que acción estamos realizando y con qué librería, ya que cada una de ellas tiene sus propias clases y métodos.

Estas acciones son:

- Creación y cierre de conexiones
- Gestión de recursos
- Gestión de colecciones
- Modificación de recursos
- Ejecución de consultas
- Gestión de excepciones

5.4.1. Creación y cierre de conexiones

El primer paso en toda aplicación que quiere utilizar un SGBD como elemento de persistencia es la creación de una conexión con la BBDD. Los pasos en el caso de Java y eXist son los mismos independientemente de la librería que estemos utilizando:

1. Carga del driver que nos permite conectar al SGBD.
2. Paso del usuario y contraseña con el que vamos a acceder a la BBDD.
3. Paso de la colección donde están los datos que queremos gestionar (inserción, consulta, modificación y borrado).



La diferencia entre las dos librerías, XQL y XML-DB está en las clases y métodos que utilizaremos en cada paso, el nivel de abstracción que ofrecen respecto a la estructura de colecciones y recursos, y la forma en la que implementan el proceso durante la conexión.

5.4.1.1. Conexiones con XML-DB

Los pasos comentados en el punto anterior, los realizamos de la siguiente forma con XML-DB:

1. Expresaremos el driver de la siguiente forma:

```
driver = "org.exist.xmldb.DatabaseImpl";
```

2. Cargaremos el driver en memoria:

```
c1 = Class.forName(driver);
```

3. Crearemos un objeto de la BBDD y lo guardaremos asociado a la clase Database:

```
database = (Database) c1.newInstance();
```

4. El objeto que representa a la BBDD, es necesario que lo registremos:

```
DatabaseManager.registerDatabase(database);
```

5. Una vez tenemos la BBDD registrada, ya podemos acceder a la colección que contiene los recursos a gestionar. La ruta de la conexión a la que queremos acceder, la especificaremos a través de una cadena de texto formada por la ruta de acceso al servicio de la BBDD y la ruta de acceso a la colección.

Un ejemplo para el acceso al servicio de la BBDD, estando esta en local:

```
URI = "xmldb:exist://localhost:8080/exist/xmlrpc";
```

Un ejemplo para el acceso a una colección:

```
collection = "/db/Libros/EnIngles";
```



Un ejemplo para especificar el usuario y contraseña con el que vamos a acceder:

```
usuario = "admin";  
usuarioPwd = "admin";
```

Ahora solo nos queda unirlo todo y crear el objeto que representa la colección:

```
col = DatabaseManager.getCollection(URI + collection,  
usuario, usuarioPwd);
```

En este punto ya tendremos creado el acceso a la colección. En el siguiente ejemplo, tenemos el código completo:

```
protected static String driver = "org.exist.xmldb.DatabaseImpl";  
public static String URI = "xmldb:exist://localhost:8080/exist/xmlrpc";  
private Database database;  
private Collection col;  
private String usuario;  
private String usuarioPwd;  
private String collection;  
private String nombre_recurso;  
public LibrosXML() throws ExpcionGestorBD {  
    this.database = null;  
    this.usuario = "admin";  
    this.usuarioPwd = "";  
    collection = "/db/Libros/EnIngles";  
    nombre_recurso = "Libros_Ingles.xml";  
    //System.out.println("He conectado ...");  
}  
  
public Collection conectarBD() throws ExpcionGestorBD {  
    try {  
        System.out.println("Intento Conectar...");  
        Class cl = Class.forName(driver);  
        System.out.println("Conecta el driver...");  
        //Se crea un objeto Database  
        database = (Database) cl.newInstance();  
        DatabaseManager.registerDatabase(database);  
        System.out.println("Ahora obtiene la colección " + URI + collection);  
        //Ahora se obtiene la colección (URI + collection)  
        //con el usuario y password que tiene acceso a ella.  
        col = DatabaseManager.getCollection(URI + collection, usuario, usuarioPwd);  
        if (col.getResourceCount() == 0) {  
            //si la colección no tiene recursos no podrá devolver ninguno  
            System.out.println("La colección no tiene recursos..."  
                + "No puede devolver ninguno [FIN]");  
            return null;  
    }  
}
```



El ejemplo, obtiene en el objeto col acceso a la colección /db/Libros/EnIngles. Dentro de esta colección podemos tener más colecciones o recursos. Con el método getResourceCount podemos saber cuántos recursos tiene y a partir de ahí hacer la gestión que corresponda a la aplicación.

5.4.1.2. Conexiones con XQJ

Con XQJ la conexión a la BBDD es mucho más sencilla, se parece mucho a cómo la realizamos con JDBC. Los pasos son los siguientes:

1. Creación de una fuente de datos (datasource):

```
ExistXQDataSource xqs = new ExistXQDataSource();
```

2. Las propiedades de la conexión que tenemos que establecer son varias, pero las imprescindibles son nombre del servidor y puerto de acceso.

```
xqs.setProperty("serverName", "localhost");  
xqs.setProperty("port", "8080");
```

3. Indicamos el usuario y la contraseña para acceder:

```
XQConnection conn = xqs.getConnection("admin", "admin");
```

El objeto de la clase XQConnection representa la conexión a la BBDD y es el que tiene los métodos necesarios para hacer toda la gestión sobre la BBDD.

```
XQConnection conn;  
XQDataSource xqs;  
//Se crea el DataSource (origen de datos)  
xqs = new ExistXQDataSource();  
//Se establecen las propiedades de la conexión  
xqs.setProperty("serverName", "localhost");  
xqs.setProperty("port", "8080");  
//Se obtiene la conexión  
conn = xqs.getConnection("admin", "password");  
//Se cierra la conexión  
conn.close();
```



En la última línea de código podemos ver que la conexión creada se cierra al finalizar la aplicación. No debemos olvidar este cierre para no dejar conexiones abiertas, tanto en XQJ como en XML-DB.

5.4.2. Gestión de recursos

De las dos APIs que estamos comentando en esta documentación, XQJ y XML-DB, la primera (XQJ) ofrece un nivel de abstracción mayor, centrándose únicamente en realizar consultas sobre la BBDD con XQuery. Además, no tiene en cuenta la estructura de colecciones y recursos creada en el sistema, por lo que no ofrece la posibilidad de crear ni eliminar colecciones y recursos. Por lo tanto, a la hora de hacer la gestión de recursos en eXist, tendremos que hacerlo a través de la API XML-DB.

Los SGBD XML nativos, como es eXist, se estructuran en colecciones y dentro de ellas guardamos los documentos o recursos que pueden ser XML o no. En el caso de eXist, los recursos los tenemos que validar respecto a un esquema (XML Schema o DTD), por lo que no tienen que tener ninguno de estos esquemas asociados y los podemos guardar en la BBDD sin más. XML-DB llama a los documentos XML o no XML recursos, que están representados por la clase Resource o alguna de sus clases hija.



5.4.2.1. Creación de recursos

Como hemos comentado anteriormente, la estructura con la que trabajamos en eXist está basada en colecciones, dentro de las cuales tenemos los recursos, de tal forma que es una estructura de árbol similar a la estructura de un sistema de archivos de un sistema operativo.

En una colección, podemos añadir recursos, ya sean XML o no. Para ello, utilizaremos las siguientes clases y métodos:

- Collection: esta clase representa una colección de recursos (Resources) existente en eXist.
- storeResource: este método guarda en la colección un recurso. Este recurso lo pasamos como parámetro de entrada al método.
- listResources: este método devuelve un array de tipo String con todos los identificadores (ids) de los recursos que tiene la colección.
- getResourceCount: este método devuelve el número de recursos existentes en la colección.
- createResource: este método crea en la colección un nuevo recurso con id y tipo que le pasamos como parámetro.



El siguiente ejemplo muestra la creación de un nuevo recurso en una colección:

```
public void asignarRecursoBD(Collection contexto, File archivo)
    throws ExcepcionGestorBD {
    try {
        // Crea un recurso vacío
        Resource nuevoRecurso = contexto.createResource(archivo.getName(),
                "XMLResource");
        // Asigna contenido del archivo al nuevo recurso vacío
        nuevoRecurso.setContent(archivo);
        // Almacena el recurso en la colección
        contexto.storeResource(nuevoRecurso);
    } catch (XMLDBException e) {
        throw new ExcepcionGestorBD("error XMLDB :" + e.getMessage());
    }
}
```

En el ejemplo podemos ver que lo primero que hacemos es crear un nuevo objeto Resource. Para hacerlo, utilizamos la conexión a la que estamos conectados (en este caso representada por el objeto contexto) utilizando el método createResource. A este método le pasamos un objeto File, que representa al archivo que queremos subir como recurso. De este objeto File, le pasamos el nombre utilizando el método getName de la clase File. En el segundo parámetro de createResource le pasamos el String “XMLResource” para indicar que el tipo del recurso que vamos a crear es de tipo XML.

Lo siguiente que hacemos en el ejemplo es asignar el contenido al nuevo recurso creado. Es decir, en el paso explicado anteriormente, lo único que hacemos es crear un nuevo recurso en la colección indicando el nombre que tendrá y el tipo, pero no su contenido. Ahora con el método setContent sobre el nuevo recurso creado, asignaremos contenido a este nuevo recurso. El parámetro que le pasamos a este método es el objeto File del que queremos obtener el contenido para el recurso.

Lo último que nos queda por hacer es asignar este nuevo recurso a la colección. Para ello utilizamos el método storeResource. Este método lo llamamos sobre la colección a la que queremos asignar el recurso y le pasamos como parámetro el nuevo recurso que hemos creado en los pasos anteriores.



Todo el código lo tenemos dentro de un bloque try-catch para poder controlar cualquier tipo de excepción de tipo XML-DB que se pueda producir durante la ejecución de la aplicación.

5.4.2.2. Acceso a recursos

Para poder acceder a un recurso existente en una colección y poder realizar alguna acción sobre su contenido, utilizaremos las siguientes clases y métodos:

- Resource: esta clase es una clase contenedora de los datos guardados en las BBDD.
- getContent: método de la clase Resource que nos permite recuperar el contenido de un recurso.
- getId: método de la clase Resource que nos permite recuperar el identificador interno y único del recurso. En el caso que el recurso sea anónimo, devolverá null. El tipo de datos que devuelve es de tipo String.
- getParentCollection: método de la clase Resource que nos permite recuperar un objeto que representa a la colección donde está ubicado el recurso. El tipo de datos que devuelve es de tipo Collection.
- getResourceType: método de la clase Resource que devuelve el tipo del recurso. El tipo de dato que devuelve es String indicando si el recurso es un documento XML (XMLResource) o un tipo no XML (BinaryResource).
- setContent: método de la clase Resource que nos permite asociar un valor como contenido a un recurso.
- XMLResource: clase hija de la clase Resource. Proporciona acceso a recursos solo del tipo XML que tengamos guardados en la BBDD. A los objetos de esta clase podemos acceder utilizando las APIs DOM y SAX.



- `getContentAsDOM` y `getContentAsSAX`: estos dos métodos son de la clase `XMLResource` y nos permiten recuperar el recurso como un nodo DOM o con un `ContentHandler` para SAX, respectivamente.
- `setContentAsDOM` y `setContentAsSAX`: estos dos métodos son de la clase `XMLResource` y nos permiten asignar contenido a los recursos utilizando un nodo DOM como fuente o utilizando un `ContentHandler` SAX, respectivamente.

El siguiente ejemplo muestra cómo acceder a un recurso de una colección y mostrar su contenido por pantalla:

```
Collection col = DatabaseManager.getCollection(URI + collection, usuario, usuarioPwd);
Resource res = null;
res = (Resource) col.getResource(nombre_recurso);
System.out.println("De la colección saca el recurso que tiene la "
+ "variable nombre_recurso:" + nombre_recurso);
XMLResource xmlres = (XMLResource) res;
System.out.println("La salida es:\n" + xmlres.getContent());
System.out.println("El tipo es:" + xmlres.getResourceType());
```

Lo primero que hacemos en el ejemplo es obtener la colección que contiene el recurso al que queremos acceder. Para obtenerla, utilizamos el mismo método `getCollection` comentado en apartados anteriores, al que le pasamos todos los datos necesarios.

Sobre este objeto que representa la colección, llamamos al método `getResource`, al que le pasamos el nombre del recurso al que queremos acceder. Este método devuelve un objeto que casteamos a `Resource` para que pueda representar correctamente al recurso.

El siguiente paso, es volver a castear el recurso al tipo concreto al que pertenece, en este caso `XMLResource`. En este ejemplo hacemos el cast directamente, pero para evitar el riesgo de una excepción de tipo `ClassCastException`, sería mucho más correcto, poner ese trozo de código dentro de una estructura `if-else` que comprobara el tipo real del recurso y así hacer el cast a la clase que corresponda.

Lo último que hacemos en el ejemplo es llamar a los métodos `getContent` y `getResourceType` y mostrar su resultado por pantalla. El primer método lo que hará será mostrar todo el contenido del recurso y el segundo un texto indicando el tipo de recurso.



5.4.2.3. Eliminación de recursos

El proceso de eliminar un recurso es el más sencillo de todos. Las clases y métodos que utilizaremos son los siguientes:

- Las clases Collection y Resource: estas clases representan a la colección a la que nos queremos conectar y al recurso que queremos eliminar, respectivamente.
- removeResource: este método de la clase Collection nos permite eliminar un recurso de una colección. El parámetro que le pasamos es un objeto de la clase Resource que representa al recurso que queremos eliminar.
- getResource: este método de la clase Collection nos permite recuperar un recurso a partir de su id. El parámetro que le pasamos es el valor que nos permitirá localizar dicho recurso.

El siguiente ejemplo muestra cómo eliminar un recurso de una colección:

```
public void borrarRecurso(Collection contexto, File archivo)
    throws ExcepcionGestorBD {
    try {
        Resource res = (Resource) contexto.getResource(archivo.getName());
        contexto.removeResource(res);
    } catch (XMLDBException e) {
        throw new ExcepcionGestorBD("error XMLDB :" + e.getMessage());
    }
}
```

A la función del ejemplo le pasamos dos parámetros. El primero representa la colección donde está el recurso que queremos eliminar. El segundo es un objeto de la clase File que representa al recurso que queremos eliminar.

Lo primero que hace la función es obtener el recurso que queremos eliminar. Para ello, utilizamos el método getResource pasándole como parámetro el nombre del archivo que se corresponde con el recurso. Este método nos devuelve un objeto que casteamos a Resource.



Lo último que queda por hacer es llamar al método `removeResource` sobre la colección. Como parámetro le pasamos el objeto `Resource` que representa el recurso que queremos eliminar.

Todo el código lo tenemos dentro de un bloque `try-catch` para poder controlar cualquier tipo de excepción de tipo XML-DB que se pueda producir durante la ejecución de la aplicación.

5.4.3. Gestión de colecciones

Al igual que ocurre con la gestión de recursos que hemos tratado en el apartado anterior, la gestión de colecciones solo la podemos hacer con la API XML-DB, por lo que todas las clases y métodos que comentaremos en los ejemplos serán de esta API.

5.4.3.1. Creación de colecciones

Para la creación de colecciones, las clases y métodos que utilizaremos son:

- `Collection`: esta clase representa la nueva colección que crearemos.
- `CollectionManagementService`: esta clase hereda de `Service` y nos permite la gestión básica de colecciones en la BBDD.
- `getService`: método de la clase `Collection` que devuelve un objeto que representa al Servicio (objeto de la clase `Service`). Como parámetros de entrada le pasamos el nombre del servicio y la versión. En el caso que no sea posible crear el servicio, nos devolverá `null`.
- `createCollection`: método de la clase `CollectionManagementService` que crea la colección en la BBDD. Como parámetro de entrada le pasaremos el nombre que queremos darle a la colección.



El siguiente ejemplo muestra cómo crear una nueva colección hija de otra colección ya existente en el sistema:

```
public Collection anadirColeccion(Collection contexto,
    String newColec) throws ExcepcionGestorBD {
    Collection newCollection = null;
    try {
        //Se crea un nuevo servicio desde el contexto
        CollectionManagementService mgtService
            = (CollectionManagementService) contexto.getService(
                "CollectionManagementService", "1.0");
        //Se crea una nueva colección con el nombre newColec codificado UTF8.
        //La colección nueva se devuelve en newCollection(Collection)
        newCollection = mgtService.createCollection(
            new String(UTF8.encode(newColec)));
    } catch (XMLDBException e) {
        throw new ExcepcionGestorBD("Error añadiendo colección: " + e.getMessage());
    }
    return newCollection;
}
```

El ejemplo está formado por una función que recibe dos parámetros de entrada. El primero es un objeto de tipo Collection que representa la colección madre donde vamos a crear la nueva. El segundo es un valor de tipo String con el nombre que vamos a darle a la nueva colección.

Lo primero que hace la función es crear un objeto de la clase CollectionManagementService. Para ello, llamamos al método getService sobre la colección madre. Como parámetros le pasamos el tipo de servicio que queremos crear “CollectionManagementService” y la versión, que en este caso es la 1.0. Ambos parámetros son de tipo String. Este método devuelve un objeto que casteamos directamente a CollectionManagementService.

Sobre este nuevo objeto creado, ya podemos crear la nueva colección. Para ello, llamamos al método createCollection sobre el objeto y le pasamos el nombre que le queremos dar a la colección. Este valor no lo pasamos directamente, lo pasamos codificado en UTF8 para no tener problemas con caracteres que no puedan ser válidos para el sistema de codificación que utilice la BBDD. El objeto devuelto ya es la nueva colección creada.

Igual que pasaba con la gestión de recursos, todo el código está dentro de un bloque try-catch para poder realizar la gestión de excepciones.



5.4.3.2. Eliminación de colecciones

El proceso de eliminar una colección es mucho más sencillo que el de crearla. En este caso volvemos a partir de la colección madre que contiene la colección que queremos eliminar. También crearemos un CollectionManagementService. La diferencia está en que en este caso, el método que utilizaremos para eliminar será removeCollection, al que le pasaremos el nombre de la colección que queremos eliminar.

Aquí tenemos un ejemplo de eliminación de una colección:

```
public void borrarColección(Collection contexto,
    String antColecc) throws ExcepcionGestorBD {
    try {
        CollectionManagementService mgtService
            = (CollectionManagementService) contexto.getService(
                "CollectionManagementService", "1.0");
        mgtService.removeCollection(antColecc);
    } catch (XMLDBException e) {
        throw new ExcepcionGestorBD(
            "Error eliminando colección: " + e.getMessage());
    }
}
```

De nuevo el ejemplo es una función. Como parámetros de entrada, la función recibe un objeto de la clase Collection que representa a la colección madre que contiene a la colección que queremos borrar y un String que es el nombre de la función a eliminar.

Lo primero que hace es crear un objeto CollectionManagementService a partir de la colección madre. Los parámetros que le pasamos son los mismos que para el caso de crear la colección. También hacemos el cast final para tener el objeto que representa al servicio.

Sobre este objeto servicio, llamamos al método removeCollection pasando el nombre de la colección a eliminar como parámetro. En este caso no tenemos que formatear el nombre a UTF8 como pasa al crear las colecciones.

Una vez más, todo el código está en un bloque try-catch para hacer una correcta gestión de excepciones.



5.4.4. Modificación del contenido recursos

Cuando utilizamos una BBDD como elemento de persistencia, las únicas acciones que realizamos sobre ella no es la gestión de su estructura, que en el caso de una BBDD XML nativa esta estructura está formada por las colecciones y recursos. Una BBDD XML nativa la solemos utilizar principalmente para gestionar el contenido de los documentos XML, es decir de los recursos almacenados en la BBDD. Esta gestión consiste en la inserción, modificación y eliminación de elementos dentro de los recursos.

Las acciones de modificación del contenido de los recursos de nuevo solo las podemos hacer con la API XML-DB, así que solo podremos utilizar clases y métodos de esta API.

Para realizar todas estas acciones de gestión del contenido de los recursos, hay diferentes alternativas. El problema está en que no todas ellas son aplicables en todos los casos, así que hay que evaluar cada caso, es decir cada SGBD XML nativo y el lenguaje de programación, para determinar qué caso se podrá implementar.

Actualmente las opciones de las que disponemos son:

1. Uso de DOM o SAX.

Es la posibilidad más básica pero menos recomendable. Para usar este sistema, primero tenemos que recuperar el recurso XML, después hacer las modificaciones necesarias, para finalmente volver a guardar el recurso en el sistema. El problema de esta opción es que puede que los recursos con los que tengamos que trabajar tengan tamaños grandes o distribuidos entre diferentes colecciones del sistema. Además, esta opción es poco flexible ya que tendremos que crear una aplicación para cada consulta.



2. XUpdate.

Es un lenguaje declarativo orientado a actualizar contenido XML. Tiene sintaxis XML y es compatible con el estándar W3C. Su principal característica es que está soportado por la API XML-DB, y aunque no todas las implementaciones de esta API que hacen los SGBD XML nativos no lo incluyen, eXist sí lo hace.

3. XQuery Update Facility.

Es un lenguaje basado en XQuery. La principal carencia de XQuery es que es un lenguaje que no tiene sentencias para la modificación de datos. XQuery Update Facility es una recomendación y no un estándar final, cosa que hace que no todas las implementaciones de las API XML-DB y XQJ la contemplen. Este es el caso de eXist, que no proporciona una implementación para esta alternativa.

4. Lenguajes declarativos propios de los SGBD XML nativos.

Cada SGBD XML nativo puede proporcionar un propio lenguaje declarativo para la modificación de datos, haciendo que las posibilidades en el mercado sean muy diversas y haciendo más difícil la existencia de un estándar.

En el caso de eXist, ofrece XQuery Update Extension, que es una extensión de la opción anterior añadiendo las opciones de modificación de contenido de un documento XML. Aunque el nombre pueda dar a pensar que es un estándar, no lo es, simplemente es una implementación de eXist para que a través de su API XML-DB podamos realizar la modificación del contenido de los recursos existentes en la BBDD.



5.4.4.1. XQuery Update Extension

Todas las sentencias de XQuery Update Extension empiezan con la palabra `UPDATE` seguida de la instrucción a realizar (`insert`, `replace/value`, `delete` o `rename`).

- `insert`

```
update insert origen (into|following|preceding) destino
```

Con esta sentencia añadimos el contenido marcado en origen en destino. Destino tiene que ser una expresión que indique un nodo del documento XML. La forma de especificar origen y destino es usando XPath.

El lugar donde insertamos la información lo especificamos de la siguiente forma:

- `into`: el contenido lo añadimos como último hijo de los nodos especificados.
- `following`: el contenido lo añadimos inmediatamente después de los nodos especificados.
- `preceding`: el contenido lo añadimos inmediatamente antes de los nodos especificados.

Un ejemplo es el siguiente:

```
update insert <nacionalidad> Rusa </nacionalidad>
into /Libros/Libro[Autor='Nicolai Gogol']
```

Añade el elemento `<nacionalidad>` a los libros cuyo autor es Nikolai Gogol.

Otro ejemplo:

```
update insert <nacionalidad> Española
</nacionalidad> preceding /Libros/Libro[Titulo="El
Sanador de Caballos"]/Autor
```

Añade el elemento `<nacionalidad>Española</nacionalidad>` antes del elemento `<autor>` para los libros con el título “El Sanador de Caballos”.



- **replace/value**

Para modificar los datos de un elemento tenemos dos opciones.

Podemos modificar el elemento con el valor o hacerlo solo indicando el valor.

```
update replace destino with nuevo_valor  
update value destino with nuevo_valor
```

Con estas sentencias reemplazamos el destino con el nuevo_valor que se define como un nodo XML en la primera opción o como valor directamente en la segunda. Destino debe devolver un único ítem.

Si destino es un elemento, entonces nuevo_valor tiene que ser también un elemento. Si destino es un nodo de texto o atributo, actualizaremos su valor con la concatenación de todos los valores de nuevo_valor.

Un ejemplo es el siguiente:

```
update replace /Libros/Libro[ Titulo="El Sanador de  
Caballo s"]/Autor with < Autor>Gonzalito S. Giner  
</ Autor>
```

Otro ejemplo:

```
update value /Libros/Libro[ Titulo="El Sanador de  
Caballo s"]/Autor with "Gonzalito S. Giner"
```

Ambos ejemplos hacen lo mismo, modifican el nombre del elemento <autor> del libro con el título “El Sanador de Caballos”.



En el primer ejemplo el nombre del elemento no cambia, continua siendo <autor>. Con esta sentencia podríamos cambiar también el nombre del elemento:

```
update replace /Libros/Libro[Titulo="El Sanador de Caballos"]/Autor with <Traductor>Gonzalito S. Giner <Traductor>
```

- **delete**

```
update delete expresión
```

Eliminamos los nodos (elemento o atributo) que indica la expresión.

Un ejemplo:

```
update delete /Libros/Libro[Titulo='El Resplandor']/Traductor
```

Elimina del documento el elemento <traductor> de los libros con título “El Resplandor”.

- **rename**

```
update rename destino as nuevo_nombre
```

Con esta sentencia cambiamos el nombre de elementos o atributos. El nodo especificado en destino, que puede ser un elemento o atributo, es reemplazado con el nuevo_nombre.

Ejemplo:

```
update rename /Libros/Libro/Autor as "Escritor"
```

Este ejemplo cambia el nombre de todos los elementos <autor> por <escritor>



Ahora que ya conocemos la sintaxis de todas las sentencias XQuery Update Extension que nos permiten hacer modificaciones en los recursos XML almacenados en eXist, el siguiente paso es ver las clases y métodos que tenemos en la API XML-DB para poder ejecutar dichas sentencias sobre la BBDD:

- Collection: esta clase representa la colección a la que hemos hecho la conexión en eXist y que contiene los recursos de los que vamos a modificar su contenido.
- getService: método de la clase Collection que ya hemos visto en apartados anteriores. Este método crea un servicio, en este caso del tipo XQueryService a partir de los parámetros nombre y versión.
- XQueryService: esta clase hereda de la clase Service y nos proporciona mecanismos para consultar las colecciones con XQuery.
- compile: este método de la clase XQueryService compila una sentencia XQuery buscando errores de sintaxis. Este método devuelve un objeto de la clase CompliedExpression que será lo que ejecutaremos sobre la BBDD. Como parámetro le pasamos la sentencia XQuery a compilar en formato String
- execute: este método de la clase XQueryService ejecuta una expresión de tipo CompliedExpression y devuelve un objeto ResultSet con el resultado de la ejecución.



En el siguiente ejemplo vemos cómo ejecutar el contenido de un recurso XML en eXist:

```
public void ModificarBD() throws ExcepcionGestorBD {
    Collection col;
    ResourceSet result = null;
    String consulta = "update replace /Libros/Libro/segundo_autor with "
        + "<segundo_autor>Linkia FP</segundo_autor>";
    /*String consulta = "update replace /Libros/Libro[Titulo=\"Norwegian Wood\"]"
        "/segundo_autor with <segundo_autor>ABC ABC</segundo_autor>";*/
    System.out.println("Intento modificar... ");
    try {
        col = DatabaseManager.getCollection(URI + collection, usuario, usuarioPwd);
        XQueryService service = (XQueryService) col.getService("XQueryService",
            "1.0");
        service.setProperty(OutputKeys.INDENT, "yes");
        service.setProperty(OutputKeys.ENCODING, "UTF-8");
        CompiledExpression compiled = service.compile(consulta);
        result = service.execute(compiled);

    } catch (XMLDBException e) {
        throw new ExcepcionGestorBD("Error ejecutando query: " + e.getMessage());
    }
}
```

La sentencia que queremos ejecutar la declaramos en la variable de tipo String consulta.

Lo primero que hace la función es conectarse a una determinada colección en la BBDD. Esta colección es la que contiene el recurso del que queremos modificar el contenido. Esta conexión la tendremos en el objeto col.

Después creamos el objeto XQueryService llamando el método getService sobre la colección creada anteriormente. A este método le pasamos el nombre del servicio que queremos crear “XQueryService” y la versión, que en este caso es la 1.0. Ambos parámetros son de tipo String. Este método devuelve un objeto que casteamos a XQueryService que es el tipo de servicio que vamos a utilizar.

A este servicio, le establecemos varias propiedades, todas ellas a través del método setProperty. Este método recibe dos parámetros, el primero es el nombre de la propiedad y el segundo el valor que le queremos dar. En el ejemplo hemos establecido dos propiedades. La primera es para indentar el documento de la forma en que indentamos normalmente los documentos XML. La segunda es para codificar el contenido en UTF-8 para no tener problemas con los caracteres no válidos en el sistema de codificación que utilice el SGBD.



A partir de aquí ya podemos compilar la sentencia. Para ello hemos usado el método compile sobre el servicio, pasándole la consulta. Esto nos genera un objeto de la clase CompiledExpression.

Solo nos queda ejecutar esa sentencia. Sobre el objeto que representa al servicio llamamos al método execute pasándole el objeto resultado de compilar la sentencia. El resultado lo guardamos en un objeto ResultSet.

5.4.4.2. XUpdate

XUpdate es actualmente la alternativa más estándar para modificar contenido de documentos XML almacenados en SGBD XML nativos. XUpdate utiliza la propia sintaxis XML para crear las sentencias de modificación de documentos XML. Todas las sentencias empiezan con el elemento `<xupdate:modifications>`. Éste debe contener un atributo `versión` para indicar la versión (obligatorio). Además, puede contener otro atributo para definir el espacio de nombres (opcional).

Dentro de este elemento, añadiremos un elemento para indicar la acción de modificación que queremos realizar, también utilizando un formato XML. El nodo contexto será el nodo que le pasemos como referencia a la sentencia. La ruta para acceder a los elementos y atributos los indicaremos con XPath.

- `xupdate:insert-before`

Inserta un nodo hermano precediendo al nodo de contexto.

```
<xupdate:modifications version='1.0' xmlns:xupdate="http://www.xmldb.org/xupdate">
    <xupdate:insert-before select="/Libros/Libro[Titulo='El Sanador de Caballos']/Autor">
        <xupdate:element name='nacionalidad'>Española</xupdate:element>
    </xupdate:insert-before>
</xupdate:modifications>
```

En el ejemplo, insertamos el elemento `<nacionalidad>` antes del elemento `<autor>` para todos aquellos `<libro>` cuyo `<titulo>` sea “El Sanador de Caballos”.



- **xupdate:insert-after**

Inserta un nodo hermano a continuación del nodo de contexto.

```
<xupdate:modifications version='1.0' xmlns:xupdate="http://www.xmldb.org/xupdate">
    <xupdate:insert-after select="/Libros/Libro/Autor[.='Nikolai Gogol']">
        <xupdate:element name='nacionalidad'>Rusa</xupdate:element>
    </xupdate:insert-after>
</xupdate:modifications>
```

En el ejemplo, insertamos el elemento `<nacionalidad>` después del elemento `<autor>` para todos aquellos `<libro>` cuyo `<autor>` sea “Nikolai Gogol”.

- **xupdate:append**

Inserta un nodo como hijo del nodo de contexto.

```
<xupdate:modifications version='1.0' xmlns:xupdate="http://www.xmldb.org/xupdate">
    <xupdate:append select="/Libros" child="last()">
        <xupdate:element name='Libro'>
            <Autor>AAA</Autor>
            <Titulo>TTT</Titulo>
        </xupdate:element>
    </xupdate:append>
</xupdate:modifications>
```

En el ejemplo, seleccionamos el último nodo `<Libros>` y añadimos un nuevo nodo hijo de tipo `<libro>`.

Si en lugar de un nuevo elemento hijo lo que queremos hacer es añadir un nuevo atributo, utilizaremos la etiqueta `<update:attribute>`.

- **xupdate:update**

Actualiza el contenido de los nodos seleccionados.

```
<xupdate:modifications version='1.0' xmlns:xupdate="http://www.xmldb.org/xupdate">
    <xupdate:update select="/Libros/Libro[Titulo='El Sanador de Caballos']/Autor">
        Gonzalo S. Giner
    </xupdate:update>
</xupdate:modifications>
```

En el ejemplo modificamos el nombre del autor del libro con el título “El Sanador de Caballos” por el de “Gonzalo S. Giner”.



- **xupdate:remove**

Elimina los nodos seleccionados.

```
<xupdate:modifications version='1.0' xmlns:xupdate="http://www.xmldb.org/xupdate">
  <xupdate:remove select="/Libros/Libro[Titulo='TTT']">
    </xupdate:remove>
  </xupdate:modifications>
```

En el ejemplo eliminamos todos los nodo `<libro>` cuyo `<titulo>` sea “TTT”.

- **xupdate:rename**

Permite cambiar el nombre de un elemento o atributo.

```
<xupdate:modifications version='1.0' xmlns:xupdate="http://www.xmldb.org/xupdate">
  <xupdate:rename select="/Libros/Libro/Autor">
    Escritor
  </xupdate:rename>
</xupdate:modifications>
```

En el ejemplo cambiamos el nombre del elemento `<autor>` por `<escritor>` en todos los elementos `<libro>`.

Ahora que ya conocemos la sintaxis de todas las sentencias XUpdate que nos permiten hacer modificaciones en los recursos XML almacenados en eXist, el siguiente paso es ver las clases y métodos que tenemos en la API XML-DB para poder ejecutar dichas sentencias sobre la BBDD:

- Collection: esta clase representa la colección a la que hemos hecho la conexión en eXist y que contiene los recursos de los que vamos a modificar su contenido.
- getService: método de la clase Collection que ya hemos visto en apartados anteriores. Este método crea un servicio, en este caso del tipo XUpdateQueryService a partir de los parámetros nombre y versión.
- XUpdateQueryService: esta clase hereda de Service adaptada para ejecutar entencias XUpdate.



- `setCollection`: método de la clase `XUpdateQueryService`, aunque en realidad es un método heredado de `Service`. Asigna los atributos de la colección al servicio, preparándolo así para ejecutar la sentencia `XUpdate`. Como parámetro de entrada le pasamos la colección.
- `update`: este método de la clase `XUpdateQueryService` que ejecuta la sentencia `XUpdate` sobre el servicio obtenido previamente con el método `getService`. Como parámetro de entrada le pasamos un valor de tipo `String` que representa la sentencia a ejecutar.

En el siguiente ejemplo vemos la ejecución de una sentencia `XUpdate`:

```
public boolean ejecutarInsertAfter() {
    boolean result = false;
    try {
        String query = "<xupdate:modifications version='1.0' "
            + "xmlns:xupdate=\"http://www.xmldb.org/xupdate\">"
            + "<xupdate:insert-after select=\"/Libros/Libro/Autor[.=\'Nikolai Gogol\']\">"
            + "<xupdate:element name='nacionalidad'>Rusa</xupdate:element>"
            + "</xupdate:insert-after>"
            + "</xupdate:modifications>";
        this.ejecutarXUpdate(query, "/db/Libros/EnCastellano");
        result = true;
    } catch (ExpcionGestorBD e) {
        System.out.println("Error en carga inicial\n" + e.getMessage());
    }
    return result;
}

public void ejecutarXUpdate(String consulta, String contexto) throws ExpcionGestorBD {
    Collection col;
    try {
        if (contexto == null) {
            col = DatabaseManager.getCollection(URI + "/db");
            System.out.print(URI + "/db");
        } else {
            col = DatabaseManager.getCollection(URI + contexto);
        }
        XUpdateQueryService serviceXUpdate = (XUpdateQueryService) col.getService("XUpdateQueryService", "1.0");
        serviceXUpdate.setCollection(col);

        serviceXUpdate.update(consulta);
    } catch (XMLDBException e) {
        throw new ExpcionGestorBD("Error ejecutando query: "
            + e.getMessage());
    }
}
```

El ejemplo está formado por dos funciones.

La primera función simplemente crea la sentencia `XUpdate` que queremos ejecutar y llama a la segunda función.



La segunda función ejecuta la sentencia XUpdate. Recibe dos parámetros de entrada de tipo String, el primero es la sentencia a ejecutar y el segundo el nombre de la colección donde están los recursos de los que queremos modificar su contenido.

Lo primero que hacemos es crear una estructura if-else para comprobar si el parámetro con el nombre de la colección está a null o no. Si no lo está, realizamos la conexión a la colección correspondiente. Si está a null, realizamos la conexión a la colección raíz (db).

Para crear esta conexión utilizamos el método getCollection de DatabaseManager. Ahora ya podemos crear el servicio, que al igual que en apartados anteriores, utilizamos el método getService sobre la colección. En este caso el nombre del servicio es “XUpdateQueryService” y la versión 1.0. El objeto que devuelve este método lo casteamos a XUpdateService.

El siguiente paso es asignar la colección a este servicio. Para ello utilizamos el método setCollection sobre el servicio pasándole como parámetro la colección a la que nos hemos conectados en el punto inicial.

Ahora solo queda ejecutar la consulta. Eso lo hacemos con el método update ejecutado sobre el servicio y con la sentencia a ejecutar como parámetro.



5.4.5. Ejecución de consultas

La última acción que nos queda por realizar sobre un SGBD es la consulta de la información que almacena. En el caso de los SGBD XML nativos el lenguaje que utilizaremos para realizar las consultas es XQuery. Es una extensión de XPath 2.0 y es el estándar propuesto por la W3C para realizar consultas sobre recursos XML, siendo el equivalente natural de SQL pero adaptado para datos XML.

En el caso de eXist y Java, tanto XML-BD como XQJ soportan la ejecución de consultas con XQuery, así que podremos usar cualquiera de las dos APIs en las aplicaciones. Las sentencias de consulta serán las mismas, solo cambiaremos el conjunto de clases y métodos que utilizará la aplicación que serán unas u otros en función de la API utilizada.

Una sentencia FLWOR (For Let Where Order Return) nos permite la unión de variables sobre un conjunto de nodos y la iteración sobre el resultado. Tiene la siguiente estructura:

- FOR

El objetivo de la cláusula FOR es obtener una secuencia de valores o nodos del documento XML sobre el que se ejecuta. El formato de la sentencia FOR está formado por:

```
for 'variable' in 'secuencia enlazada'  
    variable: en ella guardamos los nodos  
    secuencia: nodos que vamos a tratar
```

Con esta cláusula enlazamos la secuencia de valores o nodos a la variable. Para cada ocurrencia que se consiga en la secuencia enlazada, obtendremos un valor o nodo. Podemos especificar varias variables en la misma sentencia, si lo hacemos obtendremos un valor o nodo con el producto cartesiano de todas ellos, algo similar a lo que ocurre cuando incluimos más de una tabla en una cláusula FROM DE SQL.



El siguiente ejemplo devuelve todos los autores poniéndolos dentro de etiquetas <MisAutores>.

```
for $a in //Libros/Libro/Autor  
return <MisAutores>{$a}</MisAutores>
```

Primero la consulta recupera todos los nodos //Libros/Libro/Autor que se identifican en la variable \$a. Después cada valor o nodo guardados en \$a lo ubicamos entre las etiquetas <MisAutores>.

```
<MisAutores>  
    <Autor>Nicolai Gogol</Autor>  
</MisAutores>  
<MisAutores>  
    <Autor>Gonzalo Giner</Autor>  
</MisAutores>  
<MisAutores>  
    <Autor>Umberto Eco</Autor>  
</MisAutores>
```



El siguiente ejemplo usa dos variables para mostrar la unión del resultado.

```
for $a in //Libros/Libro/Autor,  
$b in //Libros/Libro/Titulo  
return <Informacion>{$a,$b}</Informacion>
```

El resultado sería el siguiente:

```
<Informacion>  
  <Autor>Nicolai Gogol</Autor>  
  <Titulo>El Capote</Titulo>  
</Informacion>  
<Informacion>  
  <Autor>Nicolai Gogol</Autor>  
  <Titulo>El Sanador de Caballos</Titulo>  
</Informacion>  
<Informacion>  
  <Autor>Nicolai Gogol</Autor>  
  <Titulo>El Nombre de la Rosa</Titulo>  
</Informacion>  
<Informacion>  
  <Autor>Gonzalo Giner</Autor>  
  <Titulo>El Capote</Titulo>  
</Informacion>  
<Informacion>  
  <Autor>Gonzalo Giner</Autor>  
  <Titulo>El Sanador de Caballos</Titulo>  
</Informacion>  
<Informacion>  
  <Autor>Gonzalo Giner</Autor>  
  <Titulo>El Nombre de la Rosa</Titulo>  
</Informacion>  
<Informacion>  
  <Autor>Umberto Eco</Autor>  
  <Titulo>l Capote</Titulo>  
</Informacion>  
<Informacion>  
  <Autor>Umberto Eco</Autor>  
  <Titulo>l Sanador de Caballos</Titulo>  
</Informacion>  
<Informacion>  
  <Autor>Umberto Eco</Autor>  
  <Titulo>El Nombre de la Rosa</Titulo>  
</Informacion>
```



También podemos trabajar con la posición que ocupan los valores o nodos.

```
for $a at $p in //Libros/Libro
return <MisLibros posición="${p}">
{$a/Autor}
</MisLibros>
```

El ejemplo, devuelve los autores, poniendo como atributo del elemento <MisLibros> un atributo posición con la posición que ocupa el elemento. La estructura \$a at \$p, asigna a \$p la posición de cada uno de los valores o nodos.

```
<MisAutores posición="1">
    <Autor>Nicolai Gogol</Autor>
</MisAutores>
<MisAutores posición="2">
    <Autor>Gonzalo Giner</Autor>
</MisAutores>
<MisAutores posición="3">
    <Autor>Umberto Eco</Autor>
</MisAutores>
```

- LET

El objetivo de la sentencia LET es el mismo que el de la sentencia FOR, obtener una secuencia de valores o nodos, pero la forma de hacerlo es diferente.

```
let 'variable' := 'secuencia enlazada'
```

En este caso la variable y la secuencia enlazada las unimos con el símbolo :=

La diferencia don FOR es que obtenemos el valor o nodos en una única etiqueta.



El siguiente ejemplo devuelve todos los autores poniéndolos dentro la etiqueta <MisAutores>

```
let $a := //Libros/Libro/Autor
return <MisAutores>{$a}</MisAutores>
```

En este caso, todos los valores o nodos estarán englobados dentro de la misma etiqueta.

```
<MisAutores>
  <Autor>Nicolai Gogol</Autor>
  <Autor>Gonzalo Giner</Autor>
  <Autor>Umberto Eco</Autor>
</MisAutores>
```

- WHERE

La cláusula WHERE es una cláusula opcional que nos sirve para seleccionar determinados valores o nodos generados como resultado por las sentencias FOR y LET.

```
WHERE 'expresion'
```

El valor de expresión lo evaluamos como un booleano para cada valor o nodo. Si el resultado es true, el valor o nodo lo tratamos. Si el false, lo descartamos.

Para obtener un booleano, lo hacemos igual que con XPath2.0.

El siguiente ejemplo devuelve todos los <Titulo> de los libros cuyo <Autor> tenga entre sus primeros 7 caracteres la cadena “NIKOLAI”.

Para ello en \$a recuperamos todos los elementos <Titulo> (3 en total) y sobre cada uno de ellos aplicamos la condición del WHERE (que en este caso solo 1 lo cumplirá). Para crear la condición hemos utilizado la función substring que retorna una subcadena determinada entre 2 posiciones (en el ejemplo 1 y 7) y upper-case que pasa todos los caracteres que forman una determinada cadena a mayúsculas. Si el resultado de ejecutar las dos funciones es una cadena igual a “NIKOLAI”, entonces el valor o nodo lo incluimos como resultado y le aplicamos la cláusula RESULT para generar el resultado.



```
FOR $a in //Libros/Libro/Autor
WHERE upper-case(substring($a, 1, 7))='NIKOLAI'
return      <LibrosNikolai>{$a/ancestor::Libro/Titulo}
</LibrosNikolai>
```

También utilizamos `ancestor::` para hacer referencia al nodo padre de `$a` y así poder acceder al `<titulo>` del libro.

```
<LibrosNikolai>
|   <Titulo>El Capote</Titulo>
</LibrosNikolai >
```

- ORDER BY

Utilizamos la sentencia ORDER BY para ordenar el resultado. Evaluamos esta sentencia antes del RETURN y después del WHERE. La forma de ordenar la secuencia de valores o nodos viene dada por ascendente (ascending) o descendiente (descending).

Como no todos los documentos XML tienen que seguir una estructura, es posible que haya valores o nodos que cumplan con la condición del WHERE pero no tengan ningún ítem que determine el orden a seguir. En este caso XQuery tenemos dos opciones: los valores o nodos vacíos los marcamos con mayor prioridad o menor en la salida.

El siguiente ejemplo devuelve los autores de los libros cuya posición es mayor que 1 pero menor que 4. El resultado lo ordenamos de forma descendiente.

```
for $a at $p in //Libros/Libro
where $p>1 and $p<4
order by $a descending
return <MisLibros posicion="{{$p}}>
{$a/Autor} </MisLibros>
```



El resultado sería el siguiente:

```
<MisLibros posicion="3">
|   <Autor>Umberto Eco</Autor>
|</MisLibros>
<MisLibros posicion="2">
|   <Autor>Gonzalo Giner</Autor>
|</MisLibros>
```

- RETURN

Evaluamos la sentencia RETURN una vez para cada valor o nodo de la secuencia resultado obtenida por las sentencias FOR, LET y WHERE, y en el orden marcado por ORDER BY.

Lo bueno de esta sentencia es que nos permite crear nuevos elementos, dando la opción a generar una estructura diferente a la que tienes los datos sobre los que hacemos la consulta.

El siguiente ejemplo crea una nueva etiqueta <MisLibros> formada por 2 atributos, el primero con la posición y el segundo con el valor del nodo <titulo>. La condición WHERE marca que solo queremos los valores del nodo que ocupa la primera posición.

```
for $a at $p in //Libros/Libro
where $p=1
order by $a descending
return element MisLibros {
    attribute posicion {$p},
    attribute titulo {$a/Titulo}
}
```

El resultado de esta sentencia sería:

```
<MisLibros posicion="1" titilo="El Capote"/>
```



Ahora que ya conocemos la sintaxis de las sentencias FLWOR, tenemos que ver de qué clases y métodos disponemos para poder ejecutarlas desde una aplicación Java. A diferencia de los casos anteriores (los de modificación de la estructura y contenido) donde solo podíamos usar la API XML-DB, cuando hablamos de realizar consultas desde una aplicación Java a eXist, podemos utilizar cualquiera de las dos APIs: XML-DB y XQJ.

5.4.5.1. API XML-DB

Para la ejecución de las consultas con XQuery, la API XML-DB utiliza el mismo conjunto de clases y métodos que para el resto de sentencias de modificación del contenido de los recursos. Esto hace que sea una muy buena opción a la hora de crear aplicaciones con Java y eXist, ya que con el mismo procedimiento, la aplicación puede ejecutar cualquier tipo de sentencia sobre la BBDD, ya sea de tipo consulta o de modificación.

Las clases que vamos a utilizar son las siguientes: Collection, XQueryService, ResourceSet, Resourcelerator.

De la clase Collection utilizaremos el método getService para obtener el servicio, igual que lo hemos utilizado en los apartados anteriores.

De la clase XQueryService utilizaremos los métodos compile y execute. El primero compila la sentencia a ejecutar devolviendo un objeto CompiledExpression. El segundo ejecuta una CompiledExpression y devuelve un ResultSet.

La clase ResultSet representa el conjunto de valores resultado de la ejecución de la consulta. El método que más utilizaremos será el getIterator, que devuelve un objeto de tipo Resourcelerator que nos va a permitir poder movernos por el resultado.



De la clase ResourceIterator los métodos más destacados son hasMoreResources y nextResource. El primero devuelve un booleano indicando si hay más recursos en el resultado por gestionar. El segundo devuelve el siguiente recurso existente en la colección de recursos resultado.

```
private XQueryService prepararConsulta(String colección) throws XMLDBException {
    Collection col = DatabaseManager.getCollection(uri + colección, user, pass);
    XQueryService servicio = (XQueryService) col.getService("XQueryService", "1.0");
    servicio.setProperty(OutputKeys.INDENT, "yes");
    servicio.setProperty(OutputKeys.ENCODING, "UTF-8");
    return servicio;
}

// Función interna para ejecutar consultas XQuery
private ResourceSet ejecutarConsultaXQuery(String colección, String consulta) throws XMLDBException {
    XQueryService servicio = prepararConsulta(colección);
    ResourceSet resultado = servicio.query(consulta);
    return resultado;
}
```

En el ejemplo tenemos dos funciones que ejecutan una sentencia XQuery con la API XML-DB y el resultado lo devuelve en una colección ResultSet.

La función prepararConsulta recibe como parámetro de entrada el nombre de la colección a la que nos vamos a conectar y que contiene los recursos sobre los que haremos las consultas. La función primero se conecta a dicha colección y después crea el servicio, en este caso de tipo XQueryService, tal y como hemos hecho en casos anteriores. Después establecemos dos propiedades para el servicio, que son la indentación y la codificación. Para finalizar, la función retorna el servicio creado.



La función ejecutarConsultaXQuery recibe como parámetro dos parámetros de tipo String, el primero es el nombre de la colección a la que nos vamos a conectar y el segundo la consulta a ejecutar. Los primero que hace la función es llamar a la función anterior para obtener el servicio. Después ejecuta la consulta utilizando el método query sobre el servicio, pasándole la consulta como parámetro de entrada. La ejecución de este método retorna una colección ResourceSet formada por todos los recursos que cumplen con la consulta ejecutada. Finalmente, la función retorna este objeto ResourceSet.

Estas dos funciones están creadas de forma genérica para que las podamos utilizar en aquellos casos en los que necesitemos ejecutar una consulta XQuery, como por ejemplo:

```
private boolean existeLibro(Libro l) throws XMLDBException {  
    String consulta = "for $t in //Libros/Libro/Titulo where $t='" + l.getTitulo() + "' return $t";  
    ResourceSet resultado = ejecutarConsultaXQuery(colecLibros, consulta);  
    return resultado.getSize() > 0;  
}
```

En este ejemplo, creamos la consulta XQuery en una variable de tipo String que después le pasaremos a la función. El resultado que guardamos en ResourceSet en este caso lo utilizamos para comprobar si hay algún elemento que cumpla con la consulta ejecutada. Para ello usamos el método getSize que nos dice el número de objetos que hay en la colección. En este caso como el objetivo de la función es ver si existe un tipo de elemento en los recursos de la BBDD, el uso de este método nos permitirá devolver un booleano indicando si hay recursos o no.



5.4.5.2. API XQJ

Con la API XQJ podemos hacer consultas a recursos XML almacenados en eXist. Esta API tiene un nivel de abstracción mayor y por lo tanto no tiene en cuenta la estructura de la BBDD, es decir las colecciones que pueda contener.

Antes de poder hacer una consulta, tenemos que conectarnos a la BBDD, tal y como hemos hecho con XQJ en apartados anteriores.

Las clases y métodos que utilizaremos para la ejecución de consultas con XQJ son los siguientes:

- **XQExpression**

Esta clase tiene las funcionalidades necesarias para definir expresiones que ejecutaremos con `executeQuery` o `executeCommand`. Para crear las expresiones que ejecutaremos, utilizaremos la clase `XQConnection`. El método `executeQuery` recibe como parámetro de tipo `String` la consulta que queremos ejecutar y devuelve el resultado en un objeto de tipo `XQResultSequence`.

- **XQResultSequence**

Es una clase que hereda de `XQSequence` que contiene una serie de ítems para que los recorramos. Tiene métodos como `count`, que devuelve el número de ítems que tiene el resultado, `close` que libera los recursos, `first` que obtiene el primer ítem de la secuencia, `getItem` que devuelve el ítem actual de tipo `XQItem` y `getPosition` que devuelve un número entero que indica la posición en la que está ubicado el cursor.



En el siguiente ejemplo vemos como conectarnos a una BBDD eXist con XQJ y realizar una consulta:

```
private void conectarBD() throws XQException{
    //Se crea el DataSource (origen de datos)
    xqs = new ExistXQDataSource();
    //Se establecen las propiedades de la conexión
    xqs.setProperty("serverName", "localhost");
    xqs.setProperty("port", "8080");
    //Se obtiene la conexión
    conn = xqs.getConnection("admin","");
}

public void cerrarBD() throws XQException{
    conn.close();
}

public XQResultSequence ejecutarQuery(String textoConsulta) throws XQException{
    // Crea un objeto expresión (reusable) XQuery Expression
    XQExpression expr = conn.createExpression();
    // Ejecuta la XQuery expression
    XQResultSequence result = expr.executeQuery(textoConsulta);
    return result;
}
```

El ejemplo está dividido en varias funciones que conjuntamente realizan el objetivo buscado.

La función conectarBD realiza la conexión a la BBDD. Para ello primero crea un objeto XQDataSource. Para ello utiliza el constructor por defecto de la clase ExistXQDataSource. Después establece las propiedades de servidor y puerto para este DataSource. A partir de este objeto, creamos la conexión a la BBDD pasando usuario y contraseña.

La función close simplemente libera los recursos al finalizar la aplicación.

La función ejecutarQuery recibe como parámetro un String con la consulta a ejecutar. A partir de la conexión creada en la primera función, crea una nueva expresión utilizando el método createExpression. Este método devuelve un objeto de tipo XQExpression. Con este objeto podemos ejecutar la consulta, llamando al método executeQuery y pasándole como parámetro el String que representa la consulta. El resultado es un objeto de tipo XQResultSequence con todos y cada uno de los recursos que cumplen con la consulta.



Un ejemplo de uso de estas funciones lo tenemos aquí:

```
XQResultSequence result;
result = gestorDB.ejecutarQuery("for $a in /Libros/Libro return ($a/Autor,$a/Titulo)");

// Iteración
if (!result.isClosed()) {
    while (result.next()) {
        // Imprime cada uno de los elementos encontrados
        System.out.println("\n" + result.getItemAsString(null));
    }
} else {
    System.out.println("El Result está cerrado...hay un problema que detectar");
}
```

A la función ejecutarQuery le pasamos la consulta a ejecutar y nos devuelve un XQResultSequence con los recursos resultado. Usamos los métodos de la clase XQResultSequence para iterar por cada recurso, para ello utilizamos el método next, para saber si hay más resultados que procesar o no. Para cada recurso procesado, mostramos por pantalla su contenido.

5.4.6. Gestión de excepciones

Todos los ejemplos explicados en esta documentación van dentro de un bloque try-catch. Esto se debe a que los métodos de las clases que estamos utilizando as las APIs XML-DB y XQJ son candidatos a generar excepciones y por lo tanto tenemos que crear el código teniéndolo en cuenta.

En la API XML-DB tenemos la clase XMLDBException que captura todos los errores que se pueden producir al tratar con SGBD XML nativos mediante esta API. Esta clase contiene una serie de códigos de error que vienen marcados por cada SGBD que la implementa y la clase ErrorCode. En el momento que el error producido sea propio del sistema gestor, la clase ErrorCode tiene un error genérico errorCode.VENDOR_ERROR.



Algunos de estos errores son:

- **COLLECTION_CLOSED**
Se activa si se ha llamado al método close en Collection.
- **INVALID_DATABASE, INVALID_COLLECTION, INVALID_RESOURCE, INVALID_URI**
Se activan para indicar que la BBDD, colección, recurso o URI son inválidos.
- **NO SUCH DATABASE, NO SUCH COLLECTION, NO SUCH RESOURCE y NO SUCH SERVICE**
Se activan si la BBDD, la colección, el recurso o la URI no pueden ser localizados.
- Otros errores como: **NOT_IMPLEMENTED, VENDOR_ERROR, UNKNOWN_ERROR, UNKNOWN_RESOURCE_TYPE, VENDOR_ERROR, PERMISSION_DENIED, WRONG_CONTENT_TYPE**

En la API XQJ tenemos la clase **XQException** que captura todos los errores que se producen las trabajar con BBDD utilizando esta API. En esta clase, tenemos una String que informa del error que se ha producido. Esta cadena la tenemos disponible a través del método `getMessage`. La causa del error la podemos obtener con el método `getCause`. Los errores propios del fabricante del SGBD los obtenemos con el método `getVendorCode`. En el caso que se produzcan varios errores a la vez, se concatenan todos en un objeto **XQException**.

Además de esta clase, la API XQJ tiene la clase **XQQueryException** que nos proporciona más información sobre la consulta XQuery ejecutada.



Recursos y Enlaces

- [Java](#)



- [API Java 11](#)



- [NetBeans](#)



- [eXist](#)



Conceptos clave

- **ACID:** se corresponden con las siglas Atomicidad, Consistencia, Aislamiento, Durabilidad. Son características que debe tener cualquier SGBD para que sea considerado como tal.
- **XML-DB:** API de Java que permite crear aplicaciones que trabajen con un SGBD XML nativo, que pueden ejecutar cualquier tipo de sentencia.
- **XQJ:** API de Java que permite crear aplicaciones que trabajen con un SGBD XML nativo, que pueden ejecutar solo consultas.



Test de autoevaluación

1. ¿Cuál es el nombre de la API que permite la modificación del contenido de los recursos en eXist?
 - a. XML-DB
 - b. XML-BD
 - c. XQJ
 - d. XML-XQJ
2. ¿Qué método de la API XML-DB nos permite ejecutar una expresión XUpdate?
 - a. execute
 - b. compile
 - c. update
 - d. executeQuery
3. ¿Cuál de las siguientes no es una opción a la hora de crear una sentencia XQuery Update Extension?
 - a. update update
 - b. update delete
 - c. update replace
 - d. update insert



Ponlo en práctica

Actividad 1

Crea una conexión a una colección de una BBDD utilizando la API XML-DB.





SOLUCIONARIOS

Test de autoevaluación

1. ¿Cuál es el nombre de la API que permite la modificación del contenido de los recursos en eXist?
 - a. **XML-DB**
 - b. XML-BD
 - c. XQJ
 - d. XML-XQJ

2. ¿Qué método de la API XML-DB nos permite ejecutar una expresión XUpdate?
 - a. execute
 - b. compile
 - c. **update**
 - d. executeQuery

3. ¿Cuál de las siguientes no es una opción a la hora de crear una sentencia XQuery Update Extension?
 - a. **update update**
 - b. update delete
 - c. update replace
 - d. update insert



Ponlo en práctica

Actividad 1

Crea una conexión a una colección de una BBDD utilizando la API XML-DB.

Solución:

```
protected static String driver = "org.existxmldb.DatabaseImpl";
public static String URI = "xmldb:exist://localhost:8080/exist/xmlrpc";
private Database database;
private Collection col;
private String usuario;
private String usuarioPwd;
private String collection;
private String nombre_recurso;

public LibrosXML() throws ExcepcionGestorBD {
    this.database = null;
    this.usuario = "admin";
    this.usuarioPwd = "";
    collection = "/db/Libros/EnIngles";
    nombre_recurso = "Libros_Ingles.xml";
    //System.out.println("He conectado ...");
}

public Collection conectarBD() throws ExcepcionGestorBD {
    try {
        System.out.println("Intento Conectar...");
        Class cl = Class.forName(driver);
        System.out.println("Conecta el driver...");
        //Se crea un objeto Database
        database = (Database) cl.newInstance();
        DatabaseManager.registerDatabase(database);
        System.out.println("Ahora obtiene la colección " + URI + collection);
        //Ahora se obtiene la colección (URI + collection)
        //con el usuario y password que tiene acceso a ella.
        col = DatabaseManager.getCollection(URI + collection, usuario, usuarioPwd);
        if (col.getResourceCount() == 0) {
            //si la colección no tiene recursos no podrá devolver ninguno
            System.out.println("La colección no tiene recursos..."
                + "No puede devolver ninguno [FIN]");
            return null;
        }
    } catch (Exception e) {
        System.out.println("Error al conectar a la base de datos: " + e.getMessage());
        throw new ExcepcionGestorBD("Error al conectar a la base de datos");
    }
}
```