



CFGs | INFORMÁTICA

Acceso a datos

Joan Gerard Camarena Estruch
José Alfredo Murcia Andrés



Acceso a datos

No está permitida la reproducción total o parcial de este libro, ni su tratamiento informático, ni la transmisión de ninguna forma o por cualquier medio, ya sea electrónico, mecánico, por fotocopia, por registro u otros métodos, sin el permiso previo y por escrito de los titulares del *copyright*.

Diríjase a CEDRO (Centro Español de Derechos Reprográficos, www.cedro.org) si necesita fotocopiar o escanear algún fragmento de esta obra.

Nota: Este libro se atiene al artículo 32 del derecho de cita de la Ley de Propiedad Intelectual de 1996 (R. D. Leg. 1/1996 de 12 de abril).

Derechos reservados © 2022, respecto a la primera edición en español, por:

McGraw Hill/Interamericana de España, S. L.

Basauri, 17 Edificio A, 1ª planta

28023 Aravaca (Madrid)

ISBN digital: 9788448626730

© Joan Gerard Camarena Estruch y José A. Murcia Andrés

Corrección técnica: José Pascual Rocher Camps

Equipo editorial: Cristina García Sánchez y Óscar Alonso Álvarez

Corrección de estilo y ortotipografía: Henrique Torreiro Sebastián

Ilustración: Pablo Vázquez Rodríguez, Archivo McGraw-Hill, 123rf

Fotografías: Archivo McGraw-Hill, 123rf



Unidad 1.

Acceso a ficheros



1. Sistema de ficheros

Desde los albores de la informática, la información se ha almacenado de manera permanente en ficheros. Aunque hoy en día los mayores volúmenes de datos se almacenan en bases de datos, se siguen almacenando tanto fotos como vídeos o una ingente cantidad de formatos en ficheros, que pueden ser utilizados como mecanismo de intercambio de información. Veremos a continuación cómo podemos procesar estos archivos, almacenados en nuestro sistema.

1.1. Gestión y organización de sistema de ficheros

Los sistemas operativos gestionan los dispositivos de almacenamiento de manera casi transparente para el usuario. Sin entrar en detalles de las tecnologías de almacenamiento subyacentes (tipos de dispositivos: discos duros, SSD, tarjetas externas) o del formato del sistema de ficheros (NTFS, ext3, NFS, APFS, etc.), estos nos ofrecen una abstracción donde solamente nos hemos de preocupar de dos conceptos: los ficheros y los directorios. Entendemos los ficheros como los contenedores de información, mientras que los directorios son los organizadores de los ficheros, que pueden contener ficheros y otros directorios.



Debemos tener en cuenta el tipo de información que contendrá el fichero, y podemos clasificarlos en ficheros de texto o ficheros binarios. Los ficheros de texto son aquellos que contienen información en una codificación interpretable por los editores de texto básicos, como **vi**, **nano** o **emacs**, e incluso interpretable por los **navegadores web**. Los ficheros binarios son aquellos cuya información se guarda codificada, representando y almacenando información de cualquier naturaleza, como imágenes, vídeo, datos, etc. Aun así, la información es accesible, aunque en texto plano puede guardarse oculta, mediante algoritmos de encriptación, como, por ejemplo, los ficheros de **password** de Apache o los ficheros de certificados.

1.2. Acceso al sistema de ficheros

Desde Java se facilita más aún la gestión de todo esto, mediante la clase `File`, cuya utilidad es que representa un enlace a un elemento genérico del sistema de archivos.

Los constructores principales que posee son:

<code>File(File parent, String child)</code>	Crea el acceso a un elemento (child) a partir de un objeto <code>File</code> existente (parent), que representa a un directorio. El constructor está sobrecargado, y el primer elemento puede ser simplemente un String
<code>File(String pathname)</code>	Crea el acceso a un elemento mediante una ruta absoluta, o bien un elemento dentro de la ruta de ejecución.



1.3. Rutas

Cuando accedemos a los sistemas de ficheros, tenemos que plantearnos en qué sistema está ejecutándose nuestro programa. En todos los sistemas operativos actuales existe la gestión en directorios y ficheros, pero debemos tener en cuenta el **separator**, referido al carácter que separa un elemento contenido dentro de otro cuando especificamos las rutas de carpetas. En sistemas basados en Linux es la barra (**Slash**, /), mientras que en sistemas Microsoft es la contrabarra (**Backslash**, \).

Para hacer nuestros programas portables deberemos utilizar la constante **File.separator**, que nos devuelve dicho elemento.

En sistemas Microsoft, en caso de tener que escribir alguna ruta, deberemos escapar la **contrabarra**, ya que tiene un significado especial dentro de los **Strings**, cosa que no ocurre con la barra:



EJEMPLO

```
File f=new File("/home/alumno/texto.md") // sistemas Linux
File f=new File("C:\\Usuarios\\alumno\\Escritorio\\texto.md") //
sistemas Windows
```

En los sistemas operativos también aparece el concepto de «**separador de rutas**» o **path separator**, carácter que se utiliza para separar una lista de **path**. Habitualmente, es el carácter «:», pero, por portabilidad, puede obtenerse mediante la constante **File.pathSeparator**.

IMPORTANTE

Las rutas absolutas están referidas respecto a la raíz del sistema operativo, mientras que las relativas lo hacen desde el punto de ejecución del programa, desde donde se ejecuta este.

Una vez se ha obtenido la referencia al elemento del sistema de ficheros, debemos comprobar si dicho elemento es un fichero o un directorio, y lo más importante, si existe o no. Para esas comprobaciones disponemos de los siguientes métodos de la clase **File**:

boolean exists()	Devuelve true si el objeto al que apuntamos existe.
boolean isFile() boolean isDirectory()	Devuelven true en el caso de que la referencia sea un fichero o un directorio, respectivamente. Estas funciones son excluyentes.

Para el caso de que queramos manipular elementos, disponemos de:

boolean createNewFile()	Crea un fichero nuevo, solo en caso de que no exista.
boolean mkdir() boolean mkdirs()	Crea un directorio nuevo, solo en caso de que no exista. Mediante mkdirs() crearemos también los directorios padre del actual, en caso de que no existan.
boolean renameTo(File)	Permite renombrar el elemento actual a otro.
boolean delete()	Elimina el elemento.

IMPORTANTE

Cuando vayamos a trabajar con el contenido de los ficheros, no necesitaremos crearlo previamente, ya que los métodos de apertura de fichero, en caso de no existir, lo crean automáticamente.

1.4. Propiedades de los elementos

Para poder consultar ciertas propiedades de los ficheros, presentamos algunos de los métodos de utilidad que posee la clase `File`.

<code>boolean canRead()</code> <code>boolean canWrite()</code> <code>boolean canExecute()</code>	Como su nombre indica, informan de los permisos de usuario para las operaciones de lectura, escritura y ejecución de cada fichero.
<code>long length()</code>	Devuelve el tamaño en bytes del fichero. Para los directorios, esta información no es representativa.
<code>String[] list()</code>	Devuelve un vector de String con los elementos que contiene el directorio actual.

En el siguiente programa se muestra el uso de cada una de las opciones anteriores.



EJEMPLO

```
public static void main(String[] args) {  
    String ruta = args[0];  
    File f = new File(ruta);  
  
    if (f.exists()) {  
        if (f.isFile()) {  
            System.out.println("El tamaño es de " + f.length());  
            System.out.println("Puede ejecutarse: " + f.canExecute());  
            System.out.println("Puede leerse: " + f.canRead());  
            System.out.println("Puede escribirse: " + f.canWrite());  
        } else {  
            String[] losArchivos = f.list();  
            System.out.println("El directorio " + ruta + " contiene:");  
            for (String archivo : losArchivos) {  
                System.out.println("\t" + archivo);  
            }  
        }  
    } else {  
        System.out.println("El fichero o ruta no existe");  
    }  
}
```

Veamos la lógica del ejemplo anterior, en el que comprobamos la existencia del fichero:

- En caso de no existir, se informa.
- En caso de existir, se verifica de qué tipo es:
 - Si es un fichero, se muestran algunos de sus atributos.
 - Si es un directorio, se muestran los elementos que contiene, mediante un listado simple.

Con todo esto, ya podemos movernos por la estructura de ficheros del sistema en el que estemos y empezar a acceder a los contenidos.



1. Ficheros de texto

Una vez estudiados los conceptos de los sistemas de ficheros, vamos a estudiar ya sus contenidos. El primer cometido de los ficheros fue almacenar la información como caracteres de texto.

Las clases que posee Java para la gestión de ficheros de texto son las siguientes:

Clase	Utilidad
FileWriter/FileReader	Nos permite gestionar ficheros de texto, aunque solo mediante lecturas/escrituras carácter a carácter o en bloques de longitud constante.
BufferedWriter BufferedReader	Creados a partir de FileWriter/FileReader, permiten el tratamiento de ficheros de texto más cercano al ser humano, ya que el tratamiento de sus datos puede realizarse línea a línea. Sigue incorporando los mecanismos de proceso carácter a carácter de la clase a partir de la cual se crea.

IMPORTANTE

Es muy importante tener en cuenta que crear un **objeto File** con el acceso a un elemento no crea un fichero (de momento), aunque nos permitirá hacerlo a partir de él.

1.1. Procesado carácter a carácter

Los objetos **FileWriter/FileReader** suelen ser adecuados para tratamientos generales de ficheros, pero, debido a la dificultad de saber cuánto ocupa la información *a priori*, no se usan demasiado. Debes de fijarte en que, por ejemplo, no podemos responder a la pregunta «¿Qué longitud tienen las frases o líneas?».

Estas clases poseen como métodos generales:

int read()	El primero de ellos lee un solo carácter, aunque devuelto en un entero, por lo que necesitaremos hacer una conversión de tipo a char (cástring). Este método devuelve un -1 cuando no puede leerse.
int read char[], offset, max)	El segundo de ellos lee un bloque de caracteres de una longitud máxima determinada, a partir de una posición en el array, devolviendo la cantidad de datos leídos. Este método devuelve un -1 cuando no puede leerse.
void write(int c)	Escribe el carácter pasado como argumento.
void write (char[])	Escribe el conjunto de caracteres pasados dentro del vector.

El siguiente ejemplo imprime por pantalla un fichero, leyendo una letra y mostrándola a continuación:



EJEMPLO

```
File f = new File(ruta);          //suponemos que existe el fichero
FileReader fr = new FileReader(f);
int c;
while ((c = fr.read()) != -1) {
    char letra = (char) c;
    System.out.print(letra);
}
```

1.2. Procesado línea a línea

El principal inconveniente para estos casos es la ausencia del concepto de línea. Este concepto se introduce en las clases **BufferedWriter** y **BufferedReader**. Mediante estas clases, podemos leer un texto hasta encontrar el separador de la línea, que viene representado mediante la constante «\n».

IMPORTANTE

El carácter \n puede denotarse como el LF (*line feed*); en ocasiones se combina también con CR (*carriage return*) por compatibilidad con protocolos antiguos, representando los dos movimientos que se introducían en la máquina de escribir para avanzar y volver al principio de la línea.

Para leer archivos línea a línea, deberemos, como antes, crear la referencia a un fichero y crear el objeto. Leeremos posteriormente con el método **String readLine()**, guardando el contenido de dicha lectura en una variable **String**. Este método devolverá **null** cuando no se ha podido efectuar la lectura porque se ha llegado al final del fichero.

IMPORTANTE

Hay que tener en cuenta que, en los ficheros de texto, entre línea y línea se almacena el carácter «\n». Al leer mediante líneas, dicho «\n» desaparece; es decir, leemos las líneas, y los «\n», como son los separadores, se pierden. Si queremos mantenerlos en ficheros de salida, deberemos añadirlos de manera explícita.

Para la escritura por bloques o líneas, necesitaremos el objeto de tipo **BufferedWriter**. El método que permite la escritura es **void write(String s)**, que envía al fichero el texto pasado. Ten en cuenta que este método no añade el \n, por lo que debemos solucionarlo con una de estas dos opciones:

```
String texto="Esto es una línea";  
fw.write(texto);  
fw.newLine();
```

```
String texto="Esto es una línea";  
fw.write(texto+"\n");
```

1.3. Archivos CSV

Los archivos **CSV** (*comma separated values*) son ficheros de texto plano que contienen información respecto a una entidad. Se estructuran en líneas, donde cada línea representa a una ocurrencia de una entidad. Cada una de estas líneas contiene una serie de valores separados por comas, que especifican las propiedades o características de cada ocurrencia.

Sea por ejemplo el extracto de un archivo **alumnos.csv**, donde se indican para una serie de alumnos su nombre, su edad, el CFGS que cursa y su nota de acceso.

```
Angela Veron,20,DAW,7.1223  
Silvia Climent,24,DAM,5.5  
Jordi Costa,22,DAW,9
```

Deberemos, en este punto, fijarnos en los siguientes detalles:

- Las líneas no tienen la misma longitud, pero eso no supondrá ningún problema porque vienen delimitadas por el \n.
- Deben existir todos los campos; si hay alguno vacío, deben aparecer las comas que lo delimitan.
- Si, por algún motivo, el carácter coma (,) puede aparecer dentro del texto (como, por ejemplo, «Pérez Angulo, José Ramón»), entonces el fichero CSV debe utilizar como separador otro carácter, como el «;» o el «#», o incluso separarlo por el carácter especial tabulación (\t).

Como resumen, el algoritmo para procesar dicho fichero deberá:

- Abrir el fichero.
- Recorrer hasta el final.
 - Leer cada línea y procesarla.
- Calcular la información de salida.
- Generar el fichero de salida, si procede.

1. Ficheros binarios. Ficheros de objetos

Como se ha comentado en contenidos de esta unidad, existen infinidad de datos que necesitan guardarse en formato binario. Esto tiene algunas ventajas:

1. Es una medida de protección, ya que los ficheros de texto, como ya sabemos, son visibles desde cualquier entorno.
2. Las operaciones de lectura y escritura son secuenciales, es decir, debemos recorrer todos los caracteres antes de llegar a uno deseado. Esta limitación se elimina con los archivos binarios.

Por tanto, estamos ante la posibilidad de guardar datos de cualquier naturaleza, pero podemos ir mucho más allá. Los objetos tienen una constitución estructurada y están formados por atributos, y en última instancia son estos los que deberán ser guardados.

IMPORTANTE

Imaginemos que poseemos una clase denominada **Alumno**, que posee un nombre, una edad y un peso, de tipos texto, entero y real. Para guardar un objeto de tipo alumno, podemos ir accediendo mediante los métodos correspondientes (**getters**) a los distintos atributos y guardándolos uno a uno en los ficheros. Como veremos, existen formas de realizar el guardado automático de todos los atributos (el objeto entero) mediante un mecanismo denominado Serialización.

1.1. Clases Java para los ficheros binarios

En el apartado anterior, vimos algunas clases de la librería de Java para el proceso de ficheros de texto. Dichas librerías, como intuirás, no son válidas para estos formatos de fichero. Para ese cometido, Java nos ofrece las siguientes clases:

FileInputStream FileOutputStream	<p>Son clases generales de acceso a ficheros binarios. Estas clases se crean, al igual que los ficheros de texto, de un objeto File. Permiten leer o escribir bytes ad hoc mediante sus métodos genéricos read() y write(), por lo que no son del todo intuitivos.</p> <p>Para leer de un fichero hasta llegar a su fin, necesitaremos evaluar lo que devuelve read, ya que la constante «-1» indica que hemos llegado al final del fichero.</p>
DataInputStream DataOutputStream	<p>Son clases generadas a partir de las correspondientes anteriores. La gran ventaja es la incorporación de los métodos readTIPO() y writeTIPO(), siendo tipo cualquiera de los tipos básicos. El flujo de la información viaja en un solo sentido: o leemos o escribimos.</p> <p>Ejemplos de estas funciones son readInt(), readDouble(), readUTF() y las correspondientes de writeInt(), writeUTF().</p>
RandomAccessFile	<p>Esta clase engloba las dos anteriores, ya que permite tanto leer como escribir los tipos adecuados. Además, tiene métodos de posicionamiento en una determinada posición mediante la función seek(int pos). También la función skipBytes(int cantidad) permite avanzar (saltarse) la cantidad determinada de bytes.</p>
ObjectInputStream ObjectOutputStream	<p>Esta clase permite, además de los métodos de DataInputStream y DataOutputStream, leer y escribir objetos directamente. Se requiere la serialización de los objetos, implementando la interfaz Serializable.</p>

GUARDAR TEXTO EN EL FORMATO BINARIO

Respecto a la escritura de cadenas de texto (y su guardado en ficheros binarios) debemos tener en cuenta lo siguiente):

- **writeString(String texto)** → almacena el **String** en el fichero.
- **writeUTF(String texto)** → almacena el **String** en el fichero, pero, al ser en formato UTF, guarda en dos bytes que preceden al texto la longitud de la cadena.

¿Para qué sirve esto? Muy sencillo: para saber la longitud del texto, en previsión de saber la cantidad de texto cuando hagamos la operación de lectura.



EJEMPLO

Escribimos en un fichero el número 10 (4 bytes), el texto HOLA (4 bytes) y el número 20 (4 bytes).

```
writeInt(10);  
writeString("HOLA");  
writeInt(20);
```

En el fichero encontraremos

0	1	2	3	4	5	6	7	8	9	10	11
10				H	O	L	A	20			

Al leer, primero leemos en **int** sin complicaciones (4 bytes), pero el problema es que no podemos leer el texto, ya que no hay ninguna marca que indique dónde termina el texto.

La solución, mediante **writeUTF()**, queda como sigue:

0	1	2	3	4	5	6	7	8	9	10	11	12	13
10				4		H	O	L	A	20			

Como podemos observar, ahora el primer **readInt()** leerá el 10, y el siguiente **readUTF()** procesa dos bytes, para leer el 4, y entonces sabe que tiene que leer el texto HOLA (los cuatro siguientes bytes), pudiendo así leer el siguiente número que queda al final.

1.2. Flujos de datos

Entendemos por flujos de datos los transvases de datos que se realizan entre un origen y un destino. Si la información que viaja no precisa ser interpretada, el mecanismo adecuado para enviar y recibir esos flujos de datos son los **FileInputStream** y **FileOutputStream**. Habitualmente, en estos flujos (**streams**) la información se envía sin interpretar como flujos de bits.

1.3. Decoradores

En caso de querer aplicar una transformación a dichos flujos de datos, necesitamos una capa de «maquillaje» que nos transforme dichos bits a datos. Esas clases decoradoras son las que se crean a partir de las anteriores y que contiene los métodos específicos.

A. ESCRITURA/LECTURA DE OBJETOS

Mediante las últimas clases decoradas, **ObjectInputStream** y **ObjectOutputStream**, podemos leer y escribir objetos directamente. Los métodos que añaden para la realización de su cometido son:

writeObject (Object o)	<p>Método de ObjectOutputStream</p> <p>Permite la escritura de un objeto cualquiera dentro de la jerarquía de objetos de Java. Para posibilitar esto, los objetos que queramos guardar deben implementar la interfaz Serializable. La serialización es el hecho de convertir un objeto en un flujo de bits; en nuestro caso, para guardar dichos bits en un fichero. También la necesitaremos para enviarlos por sockets de comunicaciones o por cualquier stream.</p> <p>Java guardará en la cabecera de los ficheros una metainformación extraída de la clase de los objetos que contiene (básicamente, los tipos de sus atributos).</p>
Object readObject()	<p>Método de ObjectInputStream</p> <p>Realiza el proceso de «deserialización» para convertir los bits del fichero en los objetos del programa. Java conoce la descripción del objeto que se va a guardar, ya que se ha guardado en la cabecera del fichero.</p> <p>Como el resultado de la carga del objeto es un Object, necesitaremos la realización de una conversión (cásting) al tipo de objeto adecuado. Si el objeto que cargamos no puede transformarse en el que indicamos en el cásting, aparecerá la excepción ClassNotFoundException.</p>



EJEMPLO

Supongamos el siguiente ejemplo:

```
public class ClaseA {
    private String texto;
}

public class ClaseB implements Serializable {
    private ClaseA elA;
    private int num;
}
```

Si guardamos un objetos del tipo B, nos encontraremos la sorpresa de que aparece una **java.io.NotSerializableException**; esto es debido a que, aunque B está serializado, al intentar guardar el objeto de tipo B, como contiene a un objeto de tipo A, este no está serializado, lo que provoca el error anterior.

B. ESCRITURA/LECTURA DE COLECCIONES

En la mayoría de los programas, no será habitual disponer de un solo objeto, sino de colecciones de estos. Por ello, podemos proceder de varias maneras para su proceso. Supongamos que tenemos una colección de n objetos, y queremos guardarlos en un fichero. Se nos presentan dos posibilidades:

Repetir n veces (1 por elemento) Guardar 1 elemento For(Alumno a: losAlumnos) writeObject(a)	Guardar la colección writeObject(losAlumnos)
Generamos un fichero con n objetos, de manera individual.	Generamos un archivo con un solo objeto, que contiene una colección de objetos individuales.

Los dos ejemplos anteriores son simplemente eso, dos posibilidades; no es ni mejor ni peor uno respecto del otro. Dependiendo de los requerimientos del programa, podemos hacerlo de una manera o de otra.

1. Almacenamiento con documentos XML

Cuando queremos guardar datos que puedan ser leídos por diferentes aplicaciones y plataformas, lo más adecuado es hacer uso de formatos estándares de almacenamiento, que múltiples aplicaciones puedan entender. Un caso muy concreto son los lenguajes de marcas, y el más conocido es el estándar **XML** (*extensible markup language*).

Con los documentos **XML** estructuramos la información intercalando marcas o *hashtags* entre la información. Estos *hashtags* tienen un principio y un final, y pueden anidarse dentro de otros, así como contener información textual. Como la información será textual, no tenemos el problema de la diferente representación de datos, puesto que cualquier dato, sea del tipo que sea, se pasará a texto. Para evitar también el problema de los diferentes sistemas de codificación de texto, **XML** permite incluir en la cabecera del documento la codificación que se ha utilizado para guardarlo.

La manera de almacenar la información en **XML** de forma jerárquica se asemeja mucho a la manera en que lo hacen los objetos en una aplicación, de modo que estos pueden traducirse de una forma relativamente cómoda a un documento **XML**.

1.1. Analizadores XML

Un *parser* o analizador XML es una clase que permite analizar un fichero XML y extraer la información de él, relacionándola según su posición en la jerarquía.

Los analizadores, según su forma de funcionar, pueden ser:

Analizadores secuenciales o sintácticos	Van extrayendo el contenido según se van descubriendo los <i>hashtags</i> de apertura y cierre. Son muy rápidos, pero tienen el problema de que hay que leer todo el documento para acceder a una parte concreta. En Java existe el analizador SAX (<i>Simple API for XML</i>) como analizador secuencial.
Analizadores jerárquicos	Suelen ser los más utilizados, ya que guardan todos los datos del documento XML en memoria, en forma de estructura jerarquizada (DOM o modelo de objetos del documento); son los preferidos para aplicaciones que tengan que leer los datos de forma más continua.

1.2. El modelo de objetos del documento (DOM)

El **DOM** (*document object model*) es la estructura especificada por el W3C donde se almacena la información de los documentos XML. El **DOM** ha sido ligado sobre todo al mundo web, con HTML y Javascript como principales impulsores. En Java, el **DOM** se implementa haciendo uso de interfaces.

La interfaz principal del DOM en Java es **Document**, y representa todo el documento XML. Como se trata de una interfaz, esta podrá implementarse en varias clases.

Además de **Document**, el W3C también define la clase abstracta **DocumentBuilder**, que permite crear el DOM a partir del XML. Además, se especifica la clase **DocumentBuilderFactory**, que nos permite fabricar **DocumentBuilders**, puesto que, al ser abstracta, no se puede instanciar directamente.

La manera de recuperar el DOM de un XML sería:

```
public Document ObreXML(String nombreArchivoXML) throws
IOException, SAXException, ParserConfigurationException,
FileNotFoundException {

    // Creamos una instancia de DocumentBuilderFactory

    DocumentBuilderFactory dbFactory =
    DocumentBuilderFactory.newInstance();

    // Con la instancia de DocumentBuilderFactory creamos un
    DocumentBuilder

    DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();

    //Con el método "parse" de DocumentBuilder obtenemos el documento
```

```
Document doc = dBuilder.parse(new File(nombreArchivoXML));  
return doc;  
}
```

Por otro lado, la clase **DocumentBuilder** nos permite también crear un DOM nuevo, con el método **newDocument()**. Esto nos servirá después para almacenar los documentos XML. Lo primero que tendremos que hacer es crear un DOM nuevo con **newDocument()**, ir añadiendo los elementos y después almacenarlo. En apartados posteriores, veremos cómo hacer todo esto. De momento, vamos a centrarnos en la interpretación y lectura del DOM.

1.3. Clases y métodos del DOM

Hasta ahora hemos visto como abrir y **parsear** un documento XML. Veamos cómo acceder a su contenido. El DOM tiene una estructura jerárquica, formada por nodos. Los diferentes tipos de nodos que nos podemos encontrar son:

- **Document**, que es el nodo principal y representa todo el XML.
- **Element**, que representa los diferentes *hashtags* (incluida la raíz).
- **TextElement**, que representa el contenido de un *hashtag* de texto.
- **Attribute**, que representa los atributos.

Todas estas interfaces derivan de la interfaz **Node**, por la que heredarán sus atributos y métodos, y además, aportarán atributos y métodos propios.

A. LECTURA DE DOCUMENTOS XML

Supongamos que disponemos del siguiente XML (recortado):

```
<curso>  
  <modulo>  
    <nombre>Acceso a Datos</nombre>  
    <horas>6</horas>  
    <nota>8.45</nota>  
  </modulo>  
  <modulo>  
    <nombre>Programación de servicios y procesos</nombre>  
    <horas>3</horas>  
    <nota>9.0</nota>  
  </modulo>
```

El siguiente bloque de código:

```
// Siendo raíz el DOM obtenido con la función anterior  
// 1 Obtenemos una lista de nodos. Devuelve un NodeList con todas las  
// etiquetas módulo  
NodeList modulos = raiz.getElementsByTagName("módulo");  
// Para cada nodo (módulo), lo recorremos. De cada Node, hacemos un  
// cásting a Element  
for (int i = 0; i < modulos.getLength(); i++) {  
  Element el = (Element) modulos.item(i);  
  // 2 Muestra el nombre del nodo  
  System.out.println(el.getNodeName() + " " + (i + 1));  
  // 3. Mostramos el valor de sus etiquetas
```

```
System.out.println("Nombre: " + el.getElementsByTagName("nombre").  
item(0)  
    .getFirstChild().getNodeValue());  
System.out.println("Horas: " + el.getElementsByTagName("horas").  
item(0)  
    .getFirstChild().getNodeValue());  
System.out.println("Nota: " + el.getElementsByTagName("nota").  
item(0)  
    .getTextContent());  
System.out.println();  
}
```

En el paso 3 requiere una explicación aparte. A partir de un **Element**, recuperamos aquel hijo que tenga como **TagName (etiqueta)** "nombre". Como solo es una y devuelve una lista, recuperamos el primer elemento (**item(0)**). A partir de él, recuperamos su primer hijo y el valor del nodo. Estas dos operaciones anteriores podemos fusionarlas, en caso de saber que es un nodo de texto con **getTextContent()**.

En caso de que un nodo posea un atributo, podemos recuperarlo del siguiente modo: **node.getAttribute("nombreAtributo")**.

B. ESCRITURA DE DOCUMENTOS XML

Vamos ahora a la parte de escritura de los documentos XML. Lo veremos mediante ejemplos:

```
Document document  
=DocumentBuilderFactory.newInstance().newDocumentBuilder()  
    .newDocument();  
document.setXmlVersion("1.0");  
Element root = document.createElement("curso");  
  
root.setAttribute("nivel", "2");  
root.setAttribute("ciclo", "DAM");  
document.appendChild(root);
```

Este bloque crea un **Document**, y le añade una raíz «curso» con dos pares de atributos, «nivel» y «ciclo». Ahora deberemos ir añadiendo más nodos, pero los añadiremos al **root** (añadir módulos al curso) en vez de al **Document**.

```
//creamos un elemento módulo y lo añadimos a la raíz  
Element modulo = document.createElement("módulo");  
root.appendChild(modulo);  
  
//añadimos las características del Elemento. Tantas como queramos  
Element nombre = document.createElement("nombre");  
nombre.appendChild(document.createTextNode("Acceso a datos"));  
nombre.setAttribute("curso", "2");  
modulo.appendChild(nombre);  
  
// añadir el resto de elementos
```

Java nos ofrece la utilidad **Transformer** para convertir información entre diferentes formatos jerárquicos, como, por ejemplo, el objeto **Document** que contiene el DOM de nuestro XML a un fichero de texto XML. La clase **Transformer** trabaja con dos tipos de adaptadores. Los adaptadores son clases que hacen compatibles jerarquías diferentes. Estos adaptadores son **Source** y **Result**. Las clases que implementan estos adaptadores se encargarán de hacer compatibles los diferentes tipos de contenedores al que requiera la clase **Transformer**. Así pues, y para clarificar, disponemos de las clases **DOMSource**, **SAXSource** o **StreamSource**, que son adaptadores del contenedor de la fuente de información para **DOM**, **SAX** o **Stream**; y de **DOMResult**, **SAXResult** y **StreamResult** como adaptadores equivalentes al contenedor destino.

Para nuestro caso, como tenemos un DOM y lo queremos convertir a **Stream (fichero)**, necesitaremos un **DomSource** y un **StreamResult**. Vemos el código necesario para hacer esto:

```
Transformer trans =  
TransformerFactory.newInstance().newTransformer();  
DOMSource source = new DOMSource(document); // el DOM creado  
previamente  
StreamResult result = new StreamResult(new  
FileOutputStream(nombreFichero+".xml"));
```

Y finalmente, solo nos queda la transformación de un elemento a otro, el que automáticamente generará el fichero XML de salida en el sistema de ficheros:

```
trans.transform(source, result);
```

Propiedad de McGraw-Hill®

1. Almacenamiento con archivos JSON

JSON es otro formato de texto ligero para el intercambio de datos. Las siglas **JSON** provienen de **JavaScript object notation** (notación de objetos de JavaScript), y se trata de un subconjunto de la notación literal de objetos de este lenguaje, que se ha adoptado junto con XML como uno de los grandes estándares de intercambio y almacenamiento de datos.

1.1. El formato JSON

La sintaxis básica, junto con los tipos de datos, que podemos representar en JSON es:

- **Números:** tanto enteros como decimales.
- **Cadenas:** expresadas entre comillas y con la posibilidad de incluir secuencias de escape.
- **Booleanos:** para representar los valores *true* y *false*.
- **Null:** para representar el valor nulo.
- **Array:** para representar listas de cero o más valores, de cualquier tipo, entre corchetes y separados por comas.
- **Objetos:** como colecciones de pares <clave>:<valor>, separados por comas y entre llaves, y de cualquier tipo de valor.



EJEMPLO

Ejemplo de fichero JSON como respuesta a una API rest sobre películas de *Star Wars*.

```
{
  "name": "Luke Skywalker",
  "hair_color": "blond",
  "birth_year": "19BBY",
  "gender": "male",
  "films": [
    "https://swapi.dev/api/films/1/",
    "https://swapi.dev/api/films/2/",
    "https://swapi.dev/api/films/3/",
    "https://swapi.dev/api/films/6/"
  ],
  ...
}
```


1.2. JSON y Java. Librería org.JSON

Existe un gran abanico de librerías Java para la manipulación de documentos **JSON** (**GSON**, **Jackson**, **JSON.simple...**). Nosotros vamos a utilizar la librería **org.json**, que podemos encontrar y añadir a nuestro proyecto en el repositorio de Maven.

La librería ofrece un conjunto de clases para procesar documentos JSON para Java, además de conversores entre JSON y XML. De entre las clases que ofrece, podríamos destacar:

org.json.JSONObject	<p>Almacena un objeto JSON con pares atributo valor, con los tipos vistos anteriormente: Boolean, JSONArray, Number, String y JSONObject.NULL. Podemos añadir y eliminar atributos de este objeto mediante los métodos:</p> <ul style="list-style-type: none"> • void put(String key, Object Valor) → añade un objeto identificado por una clave. Podemos utilizar métodos más concretos para los tipos, ya que estos métodos están sobrecargados. • JSONObject get(String key) → retorna el objeto identificado por dicha clave. • TIPO getTIPO(String key) → retorna el tipo básico identificado por esta clave, donde TIPO es cualquiera de los tipos básicos (String, int, boolean, etc.) <p>En caso de acceder a una clave inexistente, saltará la excepción JSONException.</p>
org.json.JSONTokener	Procesa una cadena JSON; utilizado internamente por JSONObject .
org.json.JSONArray	<p>Almacena una serie de valores representados por un array JSON. Dispone de mecanismos para:</p> <ul style="list-style-type: none"> • put (JSONObject) → almacena en un JSONArray el objeto indicado. • Recorrido del array: <ul style="list-style-type: none"> • int length() → devuelve la dimensión. • JSONObject get(int index) → devuelve el elemento de la posición indicada.
org.json.JSONWriter	Ofrece una forma de producir texto JSON. Dispone de un método append(String) que añade más texto a un objeto JSON de tipo texto, además de los métodos key(String) y value(String) para añadir claves y valores a una cadena JSON.

1.3. Creación de un objeto JSON a partir de un objeto Java

En este caso es muy simple, ya que directamente crearemos un **JSONObject** vacío, y mediante **get** iremos añadiendo cada uno de los atributos del objeto, según el tipo básico. Tomando como referencia la clase Módulo de los apartados anteriores, le añadiremos el siguiente método que nos permitirá obtener una representación del módulo como JSON:

```
public JSONObject getAsJSON() {
    JSONObject modulo = new JSONObject();

    modulo.put("nombre", this.nombre);
    modulo.put("horas", this.horas);
    modulo.put("nota", this.nota);

    // en caso de existir la posibilidad de un valor nulo, deberíamos gestionarlo:
    // modulo.put("atributo", this.atributo!=null?this.
    atributo:JSONObject.NULL);
    // si no es nulo, su valor, en caso contrario, JSONObject.NULL

    return modulo;
};
```

Ahora, suponiendo que tenemos un array de `Modulo`, podemos generar un objeto «curso» que contendrá a todos los `Modulo`:

```
// Raíz del documento "curso"..
JSONObject curso=new JSONObject();

// Que contendrá una lista de módulos:
JSONArray jsarray = new JSONArray();

// Rellenamos el array con los JSONObject, uno por cada Modulo
// llamando a la anterior función anterior Curso es una colección de
módulos
for(Modulo m:this.Curso) {
    JSONObject modulo_json=m.getAsJSON();
    // añadimos el objeto al vector
    jsarray.put(modulo_json);
}

// El array será una entrada en el JSONObject Raíz, por lo que debemos
añadirlo
curso.put("curso", jsarray);
```

Ahora solo nos queda el objeto raíz, guardarlo en un fichero de texto JSON. Para ello:

```
try {
    FileWriter file = new FileWriter("curso.json");
    file.write(curso.toString(4)); // 4 espacios de indentación en
el fichero
    file.close();
} catch (IOException e) {
    System.out.println("Error en la creación del fichero json");
}
```

Como podemos observar, el método interno `toString` de la clase `JSONObject` devuelve esa representación en formato texto. Como parámetro, la cantidad de espacios de indentación.

1.4. Lectura de ficheros JSON

La lectura de ficheros JSON es muy sencilla, ya que el propio constructor de `JSONObject` permite crearlos a partir de un `String`. Si conseguimos que dicho `String` contenga el fichero de texto, ya lo tenemos conseguido, por tanto:

```
try {
    // Con FileReader leemos carácter a carácter y lo guardamos en un
String
    FileReader file = new FileReader("fichero.JSON");
    String myJson="";

    int i;
    while ((i=file.read()) != -1)
        myJson=myJson+((char) i);
    file.close();

    // Con el constructor de JSONObject, el JSON lo generamos a partir
del String:
    JSONObject modulos=new JSONObject(myJson);

} catch (Exception e)
{
    System.out.println("Error cargando el fichero");
}
```

Ahora, lo que necesitamos es recoger el contenido a partir de dicho objeto. Hay que recordar que está formado por un array de **JSONObjects**.

```
JSONArray losModulos= modulos.getJSONArray("curs");
// recorremos el JSONArray
for (int i = 0; i < losModulos.length(); i++) {
    // recuperamos módulos individuales con get
    JSONObject modulo=(JSONObject) losModulos.get(i);
    // Estructura {"nombre": "PSP", "horas": 3, "nota": 7.75 }
    // A los valores de cada módulo accedemos con get:
    String nombre = modulo.get("nombre");
    int horas = modulo.getInt("horas");
    double nota = modulo.getDouble("nota");

    // creamos el módulo
    Modulo m=new Modulo(nombre,horas,nota)
}
```

Propiedad de McGraw-Hill®



Unidad 2.

Conectores a SGBD



1. El desfase objeto-relacional

Los modelos conceptuales nos ayudan a modelar una realidad compleja, y se basan en un proceso de abstracción de la realidad. Cada modelo tiene una forma de plasmar esta realidad, pero todas ellas son más próximas a la mentalidad humana que a la memoria de un ordenador.

Cuando modelamos una base de datos, hacemos uso del modelo conceptual entidad-relación y, posteriormente, llevamos a cabo un proceso de paso a tablas y normalización de este modelo, para tener un **modelo relacional de datos**.






En el caso de la **programación orientada a objetos**, intentamos representar la realidad mediante objetos y las relaciones entre ellos. Se trata de otro tipo de modelo conceptual, pero que pretende representar la misma realidad que el relacional.

Así pues, tenemos dos aproximaciones **diferentes** para representar la realidad de un problema: el modelo relacional de la base de datos y el modelo orientado a objetos de nuestras aplicaciones.



1.1. Representación de la información con el modelo relacional

El modelo relacional se basa en tablas y en la relación entre ellas:

	Cada tabla tiene tantas columnas como atributos queremos representar, y tantas filas como registros o elementos de ese tipo contenga.
	Las tablas tienen una clave principal , que identifica cada uno de los registros, y puede estar formada por uno o varios atributos.
	La relación entre tablas se representa mediante claves externas , que consisten en incluir en una tabla la clave principal de otra tabla, como «referencia» a esta.
	<p>Cuando se elimina un registro de una tabla cuya clave primaria está referenciada por otra, hay que asegurarse de que mantenemos la integridad referencial de la base de datos. Entonces, ante este borrado de una clave primaria, podemos:</p> <ul style="list-style-type: none"> – No permitir el borrado (NO ACTION). – Realizar el borrado en cascada, borrando también todos los registros que hicieron referencia a la clave primaria del registro borrado (CASCADE). – Establecer en nulos (SET NULL), de forma que la clave externa que referenciaba a la clave primaria de la otra tabla toma el valor de NULO.
	<p>Los diferentes campos de las tablas pueden tener también ciertas restricciones asociadas, como pueden ser:</p> <ul style="list-style-type: none"> – Restricción de valor no nulo, de forma que el campo no puede ser nulo en ningún caso. – Restricción de unicidad sobre uno o varios campos, de forma que el valor tiene que ser único en toda la tabla. Las claves primarias poseen ambas propiedades: valor no nulo y unicidad. – Restricción de dominio, o, lo que es lo mismo, pueden tener un conjunto de valores posibles predeterminado.

1.2. Representación de la información con el modelo orientado a objetos

Al igual que el modelo entidad-relación, el modelo orientado a objetos es un modelo de datos conceptual, pero que centra la importancia en el modelado de objetos.

Un objeto puede representar cualquier elemento conceptual: **entidades, procesos, acciones...** Un objeto no representa únicamente las características o propiedades, sino que se centra también en los procesos que estos sufren. En términos del modelo orientado a objetos, decimos que un objeto son datos más operaciones o comportamiento.

- Un objeto es una entidad con ciertas **propiedades** y determinado **comportamiento**.
- En términos de **POO**, las propiedades se conocen como **atributos**, y el conjunto de valores de estas determinan el **estado** del objeto en un determinado momento.
- El comportamiento viene determinado por una serie de funciones y procedimientos que denominamos **métodos** y que modifican el estado del objeto.
- Un objeto tendrá además un **nombre** por el que se identifica.
- Una **clase** es una abstracción de un conjunto de objetos, y un objeto tiene que pertenecer necesariamente a alguna clase.
- Las **clases definen los atributos y métodos** que poseerán los objetos de esta clase.
- Un objeto se entiende como una instancia de una clase.

1.3. Modelo relacional frente a modelo OO

Conceptualmente, el modelo orientado a objeto es un modelo dinámico, que se centra en los objetos y en los procesos que estos sufren, pero que no tiene en cuenta, de partida, su **persistencia**. Tenemos que conseguir, pues, poder guardar los estados de los objetos de forma permanente, y cargarlos cuando los necesitemos en la aplicación, así como mantener la **consistencia** entre estos datos almacenados y los objetos que las representan en la aplicación.

Una forma de ofrecer esta persistencia a los objetos sería hacer uso de un SGBD relacional, pero nos encontramos con algunas complicaciones. La primera, desde el punto de vista conceptual, es que **el modelo entidad-relación se centra en los datos, mientras que el modelo orientado a objetos se centra en los objetos**, entendidos como agrupaciones de datos, y las **operaciones** que se realizan sobre ellos.

Otra diferencia, bastante importante, es la vinculación de elementos entre uno y otro modelo. Por un lado, el modelo relacional añade información extra a las tablas en forma de clave externa, mientras que, en el modelo orientado a objetos, no necesitamos estos datos externos, sino que la vinculación entre objetos se hace a través de referencias entre ellos. Un objeto, por ejemplo, tampoco necesitará una clave primaria, puesto que el objeto se identifica por sí mismo.



Las tablas en el modelo relacional tienen una clave primaria, para identificar los objetos, y claves externas, para expresar las relaciones, mientras que en el modelo orientado a objetos estas desaparecen, expresando las relaciones entre objetos mediante referencias. Además, la forma de expresar estas relaciones también es diferente.

Por otro lado, a la hora de manipular los datos, hay que tener en cuenta que el modelo relacional dispone de lenguajes (SQL principalmente) pensados solo para esta finalidad, mientras que en un lenguaje orientado a objetos se trabaja de forma diferente, y por eso habrá que incorporar mecanismos que permiten realizar desde el lenguaje de programación estas consultas. Además, cuando obtenemos los resultados de la consulta, nos encontramos también con otro problema, y es la **conversión de resultados**. Una consulta a una base de

datos siempre devuelve un resultado en **forma de tabla**, por lo que habrá que transformar estas en estados de los objetos de la aplicación.

Otras características que habrá que tener en cuenta son las propias de la **POO** que no son modeladas con el **modelo E/R**:

Encapsulación	El diseño orientado a objetos permite ocultar el interior de los objetos, declarando los atributos como privados y accediendo a ellos a través de los métodos públicos. Este acceso a través de las tablas es mucho más complejo de conseguir.
Herencia y polimorfismo	La herencia aún puede simularse ligeramente mediante las especializaciones de las entidades. El polimorfismo no puede modelarse.
Tipos estructurados	Los sistemas gestores de bases de datos relacionales obligan a declarar tipos atómicos, y no pueden generar tablas a partir de otras previamente construidas.

Todas estas diferencias suponen lo que se conoce como **desfase objeto-relacional**, y que nos obligará a hacer determinadas conversiones entre objetos y tablas cuando queramos guardar la información en un SGBDR.

En este apartado y los siguientes, veremos cómo superar este desfase desde diferentes aproximaciones.

Propiedad de McGraw-Hill®

1. Gestión de conectores

1.1. Protocolos de acceso a bases de datos. JDBC

Cuando hablamos de protocolos de acceso a bases de datos, nos encontramos con dos normas principales de conexión:

- **ODBC (*open data base connectivity*)**: se trata de una **API (*application programming interface*)** desarrollada por Microsoft para sistemas Windows, que permite añadir diferentes conectores a varias bases de datos relacionales basadas en SQL, de forma sencilla y transparente. Mediante ODBC, las aplicaciones pueden abrir conexiones con la base de datos, enviar consultas y actualizaciones y gestionar los resultados.
- **JDBC (*java data base connectivity*)**: define una API multiplataforma, que podemos utilizar en nuestros programas en Java para la conexión a los SGBDR.

Por ser multiplataforma y por estar orientada a Java, trabajaremos con esta última. Mediante JDBC:

- Se ofrece una **API**, encapsulada en clases, que garantiza uniformidad en la forma en que las aplicaciones se conectan a la base de datos, independientemente del SGBDR subyacente.
- Necesitaremos un controlador por cada SGBDR al que nos queramos controlar.

Java no dispone de ninguna biblioteca específica ODBC, pero, para no perder el potencial de estas conexiones, sí se incorporaron unos drivers especiales que actúan de adaptadores entre JDBC y ODBC, de forma que se permite, mediante este puente, conectar cualquier aplicación Java con cualquier conexión ODBC. Actualmente, casi todos los SGBD disponen de drivers JDBC. En caso de no disponer, podemos recurrir a este último.

1.2. Arquitectura de JDBC

La biblioteca estándar de JDBC ofrece un conjunto de interfaces sin implementación. De esta implementación se encargarán los controladores o drivers de cada SGBD en cuestión. Las aplicaciones, para acceder a la base de datos, tendrán que utilizar las interfaces de JDBC, de forma que para la aplicación sea totalmente transparente la implementación que realiza cada SGBD.

Como vemos, las aplicaciones Java acceden a los diferentes métodos que especifica el API en forma de interfaces, pero son los controladores quienes acceden a la base de datos.

Hay que decir que las aplicaciones pueden utilizar varios controladores JDBC de forma simultánea, y acceder, por lo tanto, a varias bases de datos.

La aplicación especificará un controlador JDBC mediante una URL (localizador universal de recursos) al gestor de drivers, y este es quien se encarga de establecer de forma correcta las conexiones con las bases de datos a través de los controladores.

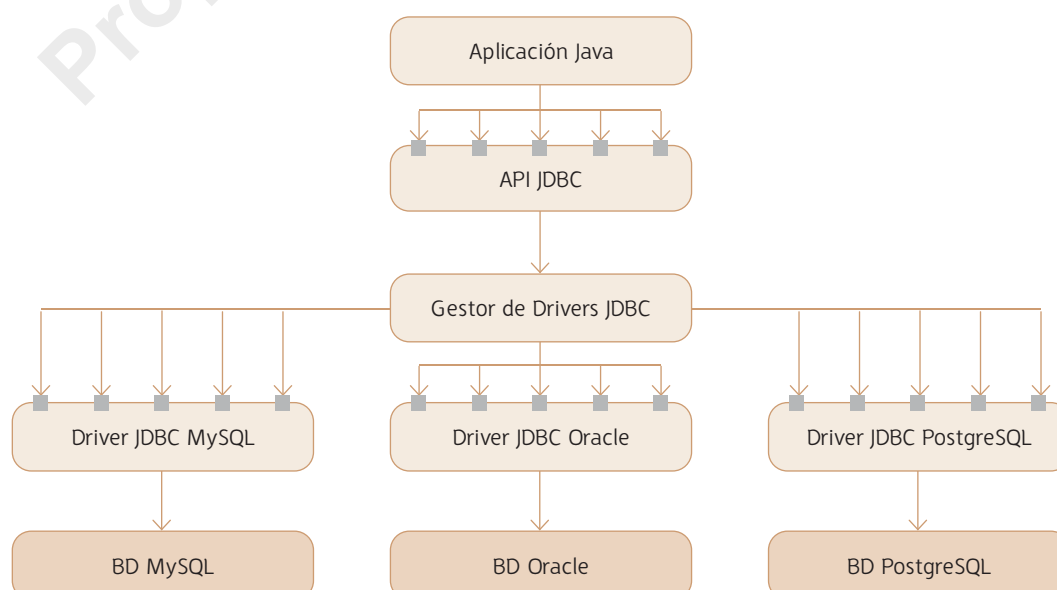


Tabla 1.1. Arquitectura de JDBC.

IMPORTANTE

Tal y como está construido el mecanismo de conexión, de manera modular, el cambio de driver y conector no afecta a la lógica del programa, ya que la API JDBC es la misma, independientemente del SGBDR.

Los controladores pueden ser de diferentes tipos:

Tipo I o controladores puente
Caracterizados por hacer uso de tecnología externa a JDBC y actuar de adaptador entre JDBC y la tecnología concreta utilizada. Un ejemplo es el puente JDBC-ODBC.
Tipo II o controladores con API parcialmente nativa o controladores nativos
Están formados por una parte Java y por otra que hace uso de bibliotecas del sistema operativo. Su uso se debe a algunos SGBD que incorporan conectores propietarios que no siguen ningún estándar (suelen ser anteriores a ODBC/JDBC).
Tipo III o controladores Java vía protocolo de red
Son controladores desarrollados en Java que traducen las llamadas JDBC a un protocolo de red contra un servidor intermedio. Se trata de un sistema muy flexible, puesto que los cambios en la implementación de la base de datos no afectan las aplicaciones.
Tipo IV o Java puros 100 %
También denominados «de protocolo nativo»; se trata de controladores escritos totalmente en Java. Las peticiones al SGBD se hacen a través del protocolo de red que utiliza el propio SGBD, por lo que no se necesita código nativo al cliente ni un servidor intermediario. Es la alternativa que ha acabado imponiéndose, dado que no requiere ningún tipo de instalación.

En el esquema anterior, fijémonos, además, en que son los fabricantes los que deben proporcionar el driver, que es quien adapta las peticiones de JDBC a su propia BD.

1.3. Bases de datos embebidas

Hoy en día existen lo que se denominan bases de datos embebidas. A efectos prácticos, son bases de datos que eliminan el servidor de la ecuación, por lo que no necesitan el sistema gestor. Son lo que se conoce como bases de datos en un único fichero, que almacenan tanto la estructura de la información como la propia información.

Dicho fichero se almacena en la propia estructura del proyecto, lo que provoca que el acceso sea muy rápido y eficaz, pero perdemos la potencia de toda la gestión que nos ofrecen los servidores. El mayor uso de estas bases de datos está apareciendo en los dispositivos móviles, por la propia encapsulación de las aplicaciones.

De cara al programador, dado que al trabajar con JDBC nos oculta que la BD está embebida, en el proyecto no cambiará nada.



EJEMPLOS DE BASES DE DATOS EMBEBIDAS

SQLite	Es un pequeño motor de SQL relacional escrito en su totalidad en C. Permite bases de datos autocontenidas en ficheros multiplataforma en forma de contenedores. Permite su uso tanto en aplicaciones de escritorio como en tecnologías móviles.
H2	Es un motor de bases de datos relacionales escrito en Java. Añade la posibilidad de la gestión de tablas basadas en memoria (existen durante la ejecución solamente) o basadas en disco. También ofrece la opción de modo servidor, añadido recientemente.
ObjectDB	Base de datos que permite ser embebida (o no, según se desee). Es totalmente orientada a objetos, por lo que todas las construcciones se harán desde el código, sin nada de SQL. Se estudiará en la unidad de bases de datos orientadas a objetos.

1. Conexión a la base de datos

1.1. Establecimiento de la conexión

Para crear una conexión a la base de datos, una vez cargado el driver, deberemos especificar dos conceptos básicos, junto con algunas opciones más. Estos mismos datos son los que nos solicita cualquier cliente para conectarse al servidor:

	HOST Debemos indicar la dirección IP o el nombre de la máquina donde se está ejecutando el servidor.
	PUERTO Habitualmente MySQL escucha por el puerto 3306, y Postgres por el 5432, aunque dicho puerto puede modificarse por decisiones de seguridad o por tener varios servidores ejecutándose a la vez.

Toda esta información la recogeremos en lo que se denomina «cadena de conexión», que será un **string** con el contenido en un cierto formato. Esta cadena de conexión tendrá un formato **URL** donde el protocolo será **JDBC** en vez del clásico **HTTP** o **FTP**. Después del protocolo **JDBC**, se indica el driver (en nuestro caso, **MySQL**) y, a continuación, el resto de los cuatro conceptos indicados.

En Java, la clase necesaria para gestionar el driver es **java.sql.DriverManager**. Los drivers los intenta cargar del sistema al leer la propiedad **jdbc.drivers**, pero podemos indicar que está cargado mediante la instrucción:

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

La clase que centralizará todas las operaciones con la base de datos es **java.sql.Connection**, y la debemos obtener del **DriverManager** con cualquiera de los tres métodos estáticos que tiene:

static Connection getConnection(String url) Retorna una conexión, si es posible, a la base de datos cuyos parámetros están especificados en la URL.
static Connection getConnection(String url, Properties info) Retorna una conexión, si es posible, a la base de datos; algunos parámetros están especificados en la URL y otros en un objeto de propiedades (Properties).
static Connection getConnection(String url, String user, String password) Retorna una conexión, si es posible, a la base de datos cuyos parámetros están especificados en la URL. Los datos de usuario y contraseña se suministran en dos parámetros adicionales

EJEMPLO

Un ejemplo básico de conexión sería:

```
String connectionUrl = "jdbc:mysql://localhost:3308";  
Connection conn = DriverManager.getConnection(connectionUrl);
```

Aquí vemos que vamos a utilizar el conector **JDBC** para el driver **MySQL**. El **SGBD** estará escuchando en nuestro **localhost**, en el puerto 3308. Como podemos intuir, esta conexión fallará, debido a que no nos hemos autenticado, y, hoy en día, aparte de las bases de datos embebidas, en prácticamente todos los **SGBD** debemos hacerlo. Una posibilidad será utilizar el tercer método que ofrece:

```
String connectionUrl = "jdbc:mysql://localhost:3308";  
Connection conn =  
    DriverManager.getConnection(connectionUrl, "root", "toor");
```

De este modo, ya nos validamos con el usuario **root** y la contraseña **toor**.

1.2. Parámetros de la conexión

En la cadena de conexión, de manera obligatoria tenemos que indicar el host y el puerto, como hemos visto. Podemos además determinar ya una base de datos concreta marcándola como activa. Para realizarlo, debemos añadir al terminar la URL de conexión una barra más el nombre de la base de datos. Así, quedaría:

```
String connectionUrl = "jdbc:mysql://localhost:3308/BDJuegos";  
Connection conn =  
    DriverManager.getConnection(connectionUrl, "root", "toor");
```

Permite conectarse al servidor de la máquina local en el puerto 3308 y marcar como activa la base de datos denominada **BDJuegos**.

IMPORTANTE

Este mecanismo de conexión no garantiza ni comprueba que la base de datos exista. Solo dará error en caso de algún error en cuanto al puerto, servidor o datos de credenciales de los usuarios.

MÁS PROPIEDADES DURANTE LA CONEXIÓN

Para poder indicar más propiedades a la conexión o parámetros adicionales, deben indicarse de la misma manera que se indican los parámetros con las peticiones **GET** y **PUT**, siguiendo el protocolo HTTP. Dichos parámetros se indican inicialmente con el símbolo «?», seguido de una lista de pares **atributo=valor**.



EJEMPLO

Un primer ejemplo de una URL de conexión completa sería:

```
String connectionUrl =  
    "jdbc:mysql://localhost:3308/BDJuegos?user=root&password=toor";  
Connection conn = DriverManager.getConnection(connectionUrl);
```

Como puede observarse, en el método **DriverManager.getConnection** no se pasan más atributos, ya que dicha información está en la cadena de conexión. Con más atributos aún:

```
String connectionUrl =  
    "jdbc:mysql://localhost:3308/BDJuegos?user=root&password=toor&useUnicode=true  
    &characterEncoding=UTF-8";
```

A modo de resumen, aquí podemos consultar la sintaxis completa de la URL de conexión:

```
jdbc:mysql://[host] [,failoverhost...]  
[:port]/[database]  
[?propertyName1 [=propertyValue1]  
[&propertyName2 [=propertyValue2]  
[&propertyName3 [=propertyValue3] ...
```

Como conclusión, podemos parametrizar dicha cadena de conexión, para evitar que esté *hard-coded*:

```
/*  
    Suponemos que las variables siguientes están obtenidas del fichero  
    de configuración o introducidas por el usuario directamente  
*/  
String usuario= "root";  
String passwd= "toor";
```

```
String dbName= "BDJuegos";

String connectionUrl = "jdbc:mysql://localhost:3308/" + dbName
+"?user=" + usuario + "&password="+ passwd
+"&useUnicode=true&characterEncoding=UTF-8";

Connection conn = DriverManager.getConnection(connectionUrl);
```

IMPORTANTE

Con el término *hard-coded* queremos hacer referencia al hecho que se produce cuando la información aparece *ad hoc* dentro de nuestro programa, almacenada en variables, en vez de en ficheros de configuración o parámetros que se indican al ejecutar el programa. Es una mala praxis que hay que evitar, ya que reduce el mantenimiento de los programas.

1.3. Organizar y centralizar la conexión

Nuestra aplicación va a conectarse a una base de datos (o a más de una). A dicha base de datos podemos hacerle muchas peticiones, y, si estamos implementando una aplicación **multihilo**, este número de peticiones puede incrementarse mucho. Por ese motivo, debemos tener controlado dónde y cuándo se crean y se cierran las conexiones. Una buena idea es crear una clase que encapsule todos estos procesos. El esqueleto de dicha clase sería el siguiente:

```
public class ConnexionBD {

    private Connection laConnexion = null;

    // variables de acceso a la base de datos

    private void connect() {
        // realizar la conexión. Ojo: método privado. Invocable desde
dentro
    }

    // cierra la conexión, si está abierta
    public void disconnect() {
        if (laConnexion != null) {
            laConnexion.close();
        }
    }

    // Si no se ha creado, la creo. En cualquier caso, la devuelvo
    public Connection getConexion() {
        if (laConnexion == null) {
            this.connect();
        }
        return this.laConnexion;
    }
}
```

Como puede comprobarse en este ejemplo, solo podemos realizar el acceso con los métodos públicos, lo que protege el objeto conexión. Además, aplicamos el **patrón de diseño Singleton**, con lo cual solo existirá una única instancia de la conexión.

IMPORTANTE

En el Enlace 3, en la sección de enlaces de interés, puedes encontrar más información sobre patrones de diseño.

Propiedad de McGraw-Hill®

1. Metainformación de la base de datos

1.1. El objeto ResultSet

Como ya hemos dicho, vamos a recuperar información, e incluso metainformación. Antes de entrar en detalles de cómo hacerlo, vamos a sentar las bases de cómo procesar la información recuperada. Para ello debemos entender cómo nos proporciona el SGBD la información. La respuesta es con una salida en formato tabulado: **ResultSet**.

Cursor	→	1	2	3
<code>.next() --</code>				
<code>true</code>				

IMPORTANTE

Un **ResultSet** es una tabla resultado de una consulta (habitualmente **Select** o similar), con tantas columnas como la selección realizada (entre el **Select** y el **From**), y tantas filas como registros hayan satisfecho el **From** (en caso de haberlo).

Para procesar el resultado de dicho **ResultSet**, haremos uso del método `next()`, que tiene dos funciones:

- Devuelve **true** si quedan resultados por procesar.
- Adelanta el cursor hasta la siguiente fila.

IMPORTANTE

El cursor se sitúa antes de la primera fila.

Recuperaremos los elementos de las columnas mediante métodos `getXXX(int pos)`, siendo:

- **xxx** → el tipo de datos que queremos recuperar: int, double, String, etc.
- **Pos** → el número de la columna, empezando la primera por 1.

Opcionalmente, si conocemos el nombre de la columna (en la tabla, o el asignado durante la selección de información), podemos indicarlo mediante un **String**.



EJEMPLO

Ejemplos de recuperación de datos dentro de un **ResultSet**:

<code>getInt(1)</code>	Recupera el entero que ocupa la columna 1.
<code>getString("nombre")</code>	Recupera el String de la columna etiquetada como «nombre».
<code>getObject(3)</code>	Recupera un objeto (útil) cuando no sabemos el tipo que contiene la columna 3.

Cuando queremos acceder a los métodos `getXXX()` debemos ser cuidadosos, ya que pueden aparecer varios errores; afortunadamente, la mayoría están contemplados por la excepción de Java **SQLException**. Esta excepción puede aparecer en algunos de los siguientes casos:

- Cuando el índice de la columna no es correcto (fuera de rango de las columnas del **ResultSet**).

- Cuando la etiqueta de la columna es incorrecta.
- Cuando intentamos acceder a un **ResultSet** que ha sido ya cerrado.
- Algún error indeterminado del servidor de la base de datos.

Así pues, aunque se verá más adelante, el proceso de trabajar con **ResultSet** será:

```
Connection laConnexion = DriverManager.getConnection(connectionUrl,
config);

// recuperamos datos de alguna manera

ResultSet rst= ....

// mientras queden datos
while (rst.next()){

    // procesado de una fila
    // accederemos a cada una de las columnas

}
```

IMPORTANTE

Dentro del bucle **while** no deberemos realizar otro **next()**, ya que, en caso de hacerlo, nos saltaremos una fila. Recuerda:

- Antes de acceder a ninguna columna, debemos hacer un **next()**. Al principio, el cursor está antes de la primera fila.
- **next()** avanza y devuelve **true** si está situado en una fila después de avanzar.
- Al final del bucle, el cursor está situado al final de la última fila (sin acceso a información).

En la sección de enlaces, concretamente en el Enlace 1, dispones de amplia información sobre los métodos y propiedades de la clase **ResultSet**.

1.2. Metadatos de la base de datos

Para consultar información de la base de datos, Java dispone de la interfaz **DatabaseMetaData**, obtenida a partir de la conexión que tenemos establecida. Así pues, accederemos a ella:

```
Connection laConnexion = DriverManager.getConnection(connectionUrl,
config);

DatabaseMetaData dbmd=laConnexion.getMetaData();
```

Este objeto contiene los métodos siguientes:

Método	Devuelve...
<code>String getDatabaseProductName()</code>	El nombre del SGBD.
<code>String getDriverName()</code>	El nombre del driver.
<code>String getURL()</code>	La URL de conexión.
<code>String getUsername()</code>	El nombre de usuario.
<code>ResultSet getTables(String catalogo, String esquema, String NombreTabla, String[] tipo)</code>	<p>Un ResultSet con las tablas del catálogo indicado. En catálogo indicamos el nombre de la base de datos de la que queremos recuperar las tablas. esquema puede dejarse con valor null. El resto de campos es para filtrados por nombre o por tipo (tabla, vista, etc.).</p> <p>Devuelve una fila por tabla. Para ver las columnas devueltas, consulta la documentación respectiva</p>

<code>ResultSet getColumnns(String catalogo, String esquema, String NombreTabla, String NombreColumna)</code>	Devuelve un ResultSet con las columnas de la tabla « NombreTabla », de la base de datos de nombre « catalogo ». El resto a null . El nombreColumna es para filtrados. Devuelve una fila por campo. Para ver las columnas devueltas, consulta la documentación respectiva.
---	--

Estas tres consultas devuelven, para el catálogo indicado y la tabla indicada, un Resultset con estas características:

Método	Devuelve...
<code>ResultSet getPrimaryKeys(String catalogo, String esquema, String patronNombreTabla)</code>	Los campos que son clave principal.
<code>ResultSet getImportedKeys(String catalogo, String esquema, String patronNombreTabla)</code>	Los campos que son apuntados por una clave ajena.
<code>ResultSet getExportedKeys(String catalogo, String esquema, String patronNombreTabla)</code>	Los campos de donde sale un clave ajena.

Con estos métodos de aquí ya podemos consultar lo imprescindible de la metainformación de la base de datos, como veremos en los casos prácticos a continuación.

Propiedad de McGraw-Hill®

1. Consultas a la base de datos

1.1. Operaciones sobre la base de datos

Antes de lanzar una consulta, debemos componer la propia consulta (lo que sería la sentencia SQL). Habitualmente, esta consulta la compondremos utilizando un **string** y adicionalmente algunos argumentos, como veremos. Para crear las sentencias, tenemos las interfaces **Statement** y **PreparedStatement**, obtenidas a partir de los objetos **Connection**. Posteriormente, las ejecutaremos con **execute()** (consulta que no genera resultados) o **executeQuery()** (consulta que sí genera resultados y deberemos procesarlos).

IMPORTANTE

Los ejemplos que verás en este contenido puedes encontrarlos en el archivo **AD_U02_A05_01.zip**, basados en la base de datos **Instituto.sql**, que también encontrarás en el archivo.

1.2. Tipos de sentencias

A. SENTENCIAS FIJAS

Son consultas que son «constantes», es decir, que no dependen de ningún argumento. Son las más simples. Se crea el **Statement** y se lanza la consulta.

```
String SQL="Select * from Persona";  
Statement stm=laConexion.createStatement();  
ResultSet rst=stm.executeQuery(SQL);  
while (rst.next()){  
    System.out.println(rst.getString("nombre") +  
        " " + rst.getString("apellidos") +  
        ": " + rst.getInt("edad"));  
}
```

Como la tabla **Persona** está formada por los campos **idPersona**, **nombre**, **apellidos** y **edad**, podemos acceder a dichas columnas mediante su posición (empezando por la 1) o por su nombre. Fijate en que, además, para acceder a las columnas de cada una de las filas del **ResultSet** utilizamos **getXXX**, donde **XXX** es el tipo de datos de cada columna. Si lo desconocemos, siempre podemos recurrir a **getObject()**.

Otra consulta podría ser una actualización o inserción, como, por ejemplo:

```
String SQL = "Insert into Persona(nombre,apellidos,edad)"+  
    " values ('Isabel','Grau Sainz',30);";  
Statement stm = laConexion.createStatement();  
int filas = stm.executeUpdate(SQL);  
if (filas==1)  
    System.out.println("Inserción realizada con éxito");  
else  
    System.out.println("Error en la inserción");
```

Aquí el detalle es que la sentencia se ejecuta con un **executeUpdate()**, ya que se modifica la base de datos, y dicha ejecución devuelve un entero, que es el número de filas afectadas. En caso de ser 1, es que la inserción ha tenido éxito.

B. SENTENCIAS VARIABLES

En el ejemplo anterior, se presenta el problema de que la sentencia está *hard-coded* (valores dentro del código), y ya hemos comentado que esto representa un problema. Los valores deben estar en variables; por lo tanto, podríamos mejorar esta consulta reescribiendo el **query** de esta manera:

```
String nombre=Leer.leerTexto("Dime el nombre: ");  
String apellidos=Leer.leerTexto("Dime los apellidos: ");  
int edad=Leer.leerEntero("Dime la edad: ");  
String SQL = "Insert into Persona(nombre,apellidos,edad)" +  
    " values ('"+nombre+"', '"+apellidos+"', '"+edad+"')";
```

Como podemos ver, los datos ahora están en variables, pero la construcción del SQL es más compleja. Fíjate en que los textos deben estar entre comillas y los números no, lo que hace muy probable la equivocación al programar. Además, este tipo de código puede hacer incurrir en problemas de **inyección SQL**, como vemos en el ejemplo que sigue:

```
String idPersona=Leer.leerTexto("Dime el id que consultar: ");  
String SQL = "Select * from Persona where idPersona="+idPersona;
```

Si el usuario introduce **4**: mostrará la persona de **idPersona 4**.

Si el usuario introduce **4 or 1=1**: mostrará todas las personas.

Este tipo de consultas debemos evitarlas en sentencias de validación de usuarios; para ello utilizaremos las sentencias preparadas.

C. SENTENCIAS PREPARADAS

Para evitar el problema de la **inyección SQL**, siempre que tengamos parámetros en nuestra consulta haremos uso de las sentencias preparadas. En las sentencias preparadas, donde tengamos que hacer uso de una variable, en vez de componerla con concatenaciones dentro del **String**, la indicaremos con un interrogante (**?**), carácter que se denomina *placeholder*.

A continuación, deberemos asignar valores a dichos *placeholders*, mediante métodos **setXXX(int pos)**, donde **XXX** es el tipo de datos que vamos a asignar y **pos** la posición del *placeholder*, empezando por el 1. Puedes ver un ejemplo en el método **cinco_SelectConSentenciaPreparada()** del archivo **AD_U02_A05_01.zip** que indicamos al principio.

Es importante que en este ejemplo revises lo siguiente:

- Al crear la sentencia preparada ya indicamos el **String SQL** que contiene los *placeholders*.
- El segundo argumento es opcional; en este caso, el **RETURN_GENERATED_KEYS** nos viene bien porque nos devuelve un **ResultSet** con las claves generadas de manera automática por el SGBD.
- El **ExecuteUpdate** retorna el número de filas insertado.
- El método **getGeneratedKeys** devuelve un **ResultSet** (una tabla) con las claves principales generadas (de tipo *long*)

D. METADATOS DE LAS CONSULTAS

Independientemente del tipo de sentencia que ejecutemos (preparada o no), siempre que realicemos una consulta (**executeQuery()**) el resultado es un **ResultSet**, que, como hemos comentado, es una tabla con los datos. Podemos consultar metainformación de esa tabla retornada por la **Select**, mediante un objeto que podemos extraer del **ResultSet**, que es el **ResultSetMetaData**. Dicho objeto contiene:

<code>int getColumnCount()</code>	Devuelve cuántas columnas tiene el ResultSet (campos de la consulta). Muy cómodo en consultas tipo Select* , que <i>a priori</i> no sabemos lo que tienen.
<code>String getColumnName(int index)</code>	Estos dos métodos nos indican, a partir del índice de la columna, el nombre del campo y el tipo de dato de cada campo.
<code>String getColumnType(int index)</code>	

Puedes ver un ejemplo en el método `seis_ResultSetConMetaDAta()` del archivo `AD_U02_A05_01.zip` que indicamos al principio.

1.3. Scripts

Un script, que habitualmente tenemos creado en un fichero externo, no es más que un conjunto de **sentencias SQL** ejecutadas en un orden de arriba abajo. Podríamos coger como estrategia ir leyendo línea a línea el fichero e ir ejecutando, pero JDBC permite ejecutar un conjunto de instrucciones en bloque. Para ello, lo primero que debemos hacer es habilitar la ejecución múltiple, añadiendo un parámetro a la conexión, que es `allowMultipleQueries=true`.

A continuación, deberemos de cargar el fichero y componer un **string** con todo el script. Para normalizarlo y hacerlo totalmente portable, deberemos ir con cuidado con los saltos de línea, ya que, dependiendo del sistema, es un «\n» o una combinación «\r\n».

Para ello, podemos ir leyendo línea a línea y guardándolo en un **StringBuilder**, añadiendo como separadores el `System.getProperty("line.separator")`.

Después, solo necesitaremos crear una sentencia con dicho **string** y ejecutarla con un `executeUpdate()`. En el ejemplo siete, queda solucionado muy sencillo.

1.4. Transacciones

Si queremos proteger la integridad de los datos, así como evitar situaciones de bloqueos inesperados en aplicaciones multihilo, deberemos proteger nuestras operaciones, especialmente las que modifiquen datos mediante el uso de transacciones.

Una transacción define un entorno de ejecución en el que las operaciones de guardado se quedan almacenadas en memoria hasta que finalice esta. Si en un determinado momento algo falla, se devuelve el estado al punto inicial de la misma, o algún punto de marcado intermedio. **Por defecto, al abrir una conexión se inicia una transacción.**

Cada ejecución sobre la conexión genera una transacción sobre sí misma. Si queremos deshabilitar esta opción para que la transacción englobe varias ejecuciones, deberemos marcarlo mediante

```
laConexion.setAutoCommit(false);
```

Para aceptar definitivamente la transacción, lo realizaremos mediante `laConexion.commit()`; y para cancelar la transacción, `laConexion.rollback()`;

1.5. ResultSet actualizables

Los **ResultSet** que obtenemos de las consultas, por lo general, nos servirán para cargar datos que mostrar en nuestros programas. Muchas veces, dichos datos serán modificados, y, por lo tanto, aparte de cargar los datos, deberemos guardar la información. Ahí es donde aparecen los **ResultSet** actualizables, que dependerán del modo que se creó la sentencia (preparada o no), con la sintaxis:

```
public abstract Statement createStatement(  
    int arg0,    // resultSetType  
    int arg1,    // resultSetConcurrency  
    int arg2)    // resultSetHoldability  
    throws SQLException
```

Donde **ResultSetType**:

TYPE_FORWARD_ONLY	Por defecto. Un solo recorrido.
TYPE_SCROLL_INSENSITIVE	Permite <i>scroll</i> o rebobinado a posición absoluta o relativa. Los datos son los que se cargan en la apertura. MySQL solo soporta este (versión 8).
TYPE_SCROLL_SENSITIVE	Permiten <i>scroll</i> , y, si hay modificaciones en la base de datos, los cambios son visibles en el ResultSet .

IMPORTANTE

No todos los SGBD soportan todos los tipos. Puede consultarse con **DatabaseMetaData.supportsResultSetType (type)** .

Donde **ResultSetConcurrency**:

CONCUR_READ_ONLY	Solo se soporta lectura. Cambios solo con update. Opción por defecto en MySQL.
CONCUR_UPDATABLE	Permite actualizar el ResultSet .

Donde **ResultSetHoldability** determina el comportamiento cuando se cierra una transacción con un **commit**:

HOLD_CURSORS_OVER_COMMIT	El ResultSet no se cierra al finalizar la transacción.
CLOSE_CURSORS_AT_COMMIT	El ResultSet se cierra. Mejora el rendimiento.

Como hemos visto, el cursor no solo admite avanzar.

- **Next, previous, first, last**: adelante, atrás, al principio o al final: devuelven **true** si se ha posicionado sobre una fila y **false** si se ha salido del **ResultSet**.
- **beforeFirst** y **afterLast**: se sitúan antes del primero o después del último.
- **relative (int n)**: se desplaza *n* filas hacia adelante.
- **absolute (int n)**: se sitúa en la fila *n*.

A. BORRADOS

Después de situar el cursor sobre la fila que vamos a eliminar, podemos eliminarla del **ResultSet** (y de la base de datos) con el método **deleteRow()**. Al borrar una fila, el cursor quedará apuntando a la fila anterior a la borrada.

B. ACTUALIZACIONES

Después de situar el cursor sobre la fila deseada, debemos actualizar las columnas que queremos, mediante el método **updateXXX (int, nuevoValor)**, donde se asigna a la columna *i-ésima* (o con su nombre) el valor nuevo valor del tipo **XXX**. Modificados todas las columnas, se guardan los cambios con **updateRow()**.

C. INSERCIONES

Si queremos insertar una nueva fila en un **ResultSet**, primero debemos generarla en blanco, y esto se consigue con el método **rst.moveToInsertRow()**, que crea una fila «virtual» en blanco. Sobre esta fila le aplicamos los métodos **updateXXX (int, nuevoValor)**, y finalmente procederemos a insertar la nueva fila con **rst.insertRow()**.

IMPORTANTE

- Estas operaciones de actualización, borrado e inserción solo pueden realizarse sobre consultas que tienen origen en una tabla sin agrupaciones.
- Para evitar complejidad en nuestros programas, cabe valorar la conveniencia de «traducir» las actualizaciones de **ResultSet** a SQL puro y ejecutarlo nativamente en la bases de datos mediante nuevas sentencias.

En la sección Descarga de documentos puedes acceder al documento **Tabla resumen del esquema de trabajo**, en el que se resume el contenido de este apartado.

Propiedad de McGraw-Hill®



Unidad 3.

Herramientas de mapeo Objeto-Relacional. JPA



1. Herramientas de mapeo. Características

Vamos a presentar las herramientas de mapeo para tratar de solventar el desfase objeto-relacional. Esta no será la única solución, ya que, como veremos en las unidades posteriores, podemos, o bien añadir directamente objetos a nuestras bases de datos relacionales, en las BB. DD. OR (bases de datos objeto-relacionales), o bien trabajar con BB. DD. OO (bases de datos orientadas a objetos).

Las técnicas de los **ORM** (herramientas de mapeo objeto-relacional) se encargan, mediante un conjunto de descripciones y metadatos (datos que describen datos), de realizar una correspondencia entre los datos primitivos de ambos modelos y sus estructuras: entre tablas y objetos, campos y atributos, sus identificadores y sus claves principales. Esta correspondencia no será siempre sencilla, y habrá que disponer de metadatos que puedan expresar una mayor complejidad. Por ejemplo, nos podemos encontrar con que, a veces, puede interesar almacenar una propiedad en más de una columna, o varias propiedades en una columna única; en otras ocasiones, puede haber propiedades que no se almacenan, o campos de la base de datos que no aparezcan en los objetos; o bien que se utilizan atributos con tipos de datos no primitivos que haya que convertir en otras tablas, y decidir qué campos serán claves externas que apuntan a las nuevas tablas.

Del mismo modo que la definición de los datos, necesitaremos un mecanismo de persistencia de los objetos, de manera que los objetos puedan ser «rastreados» en memoria, y que los cambios en estos se vean reflejados directamente en la base de datos.

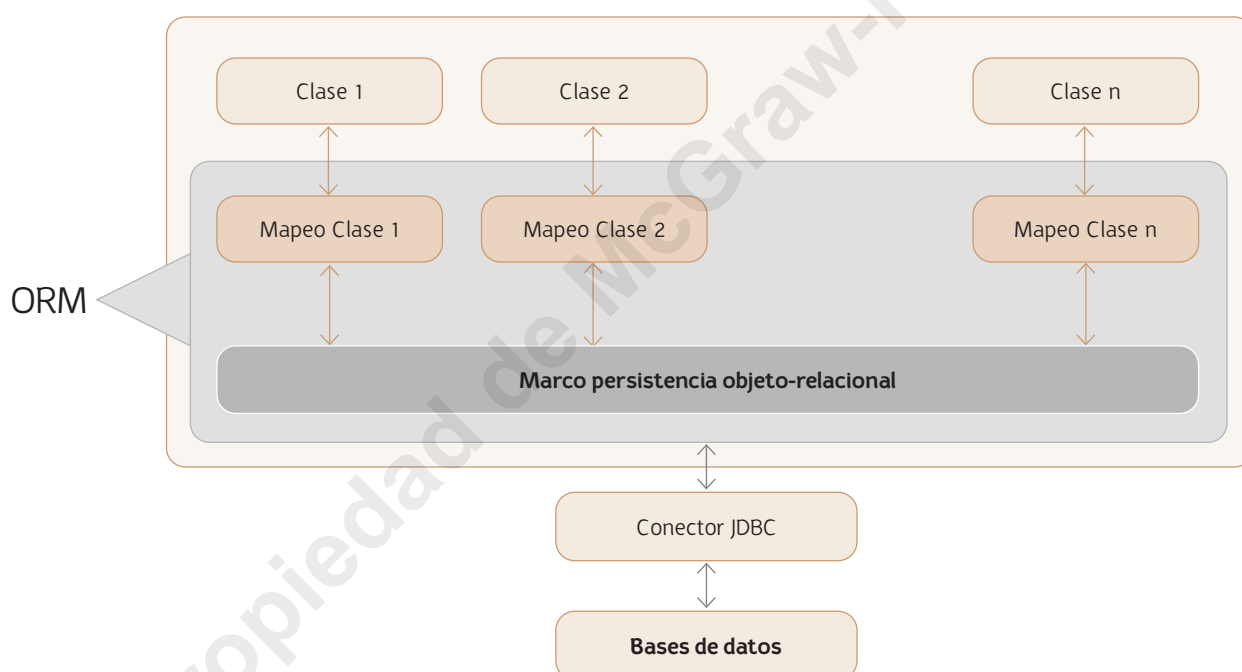


Tabla 1.1. Funcionamiento de un ORM.

Este tipo de herramientas aportan, entre otras, las siguientes características:

- Disminuyen el tiempo de desarrollo.
- Permiten realizar una abstracción del SGBD subyacente, en parte gracias a JDBC.
- Manipularemos solo objetos en nuestro programa, olvidándonos del concepto de tabla, fila y columna.

Cabe destacar que estas ventajas también suponen un coste, el de realizar estas traducciones, que se hacen en cualquier consulta, por lo que el rendimiento no será el mejor.

1.1. Herramientas de mapeo. Características

La mayoría de los lenguajes de programación disponen de herramientas ORM. En estas herramientas ORM hay que distinguir tres componentes:

Técnicas de mapeo	Consistentes en un sistema para expresar la correspondencia entre las clases y el esquema de la base de datos.
Un lenguaje de consulta orientado a objetos	Realmente accederá a las tablas, permitiendo salvar el desfase objeto-relacional.
Técnicas de sincronización	Suponen el núcleo funcional para posibilitar la sincronización de los objetos persistentes de la aplicación con la base de datos.

A. TÉCNICAS DE MAPEO

Destacamos dos técnicas de mapeo objeto-relacional:

- Aquellas que incrustan las definiciones dentro del código de las clases y están vinculadas al lenguaje, como las macros de C++ o las anotaciones de PHP y Java.
- Aquellas que guardan las definiciones en ficheros independientes del código, generalmente en XML o JSON.

No se trata de técnicas excluyentes; ambas están disponibles en la mayoría de los entornos, y pueden incluso convivir en la misma aplicación.

B. LENGUAJE DE CONSULTA

Todos los SGBDR llevan el soporte de SQL, como es obvio. En las herramientas de mapeo, existe lo que se denomina el **OQL** (*object query language*). Este lenguaje tiene muchas similitudes con SQL, pero una gran diferencia, que se basa en objetos y no en tablas. Cada ORM lleva el OQL a su campo de juego, realizando algunas pequeñas variaciones en cuanto a sintaxis.

C. TÉCNICAS DE SINCRONIZACIÓN

La sincronización es uno de los aspectos más delicados de las herramientas ORM. Consiste en procesos complejos que implican técnicas para las siguientes funcionalidades:

- Descubrir los cambios que sufren los objetos durante su ciclo de vida para poder almacenarlos.
- Crear e iniciar nuevas instancias de objetos a partir de los datos guardados en la base de datos.
- A partir de los objetos, extraer su información para reflejarla en las tablas de la base de datos.

1.2. Hibernate

Hibernate es un **framework ORM** para Java, que facilita el mapeo de atributos entre una base de datos relacional y el modelo de objetos de nuestra aplicación mediante **ficheros XML** o anotaciones en los **beans** de las entidades. Se trata de software libre distribuido con licencia GPL 2.0, por lo que se puede utilizar en aplicaciones comerciales.

La principal función de Hibernate será ofrecer al programador las herramientas para detallar su modelo de datos y las relaciones entre ellos, de forma que sea el propio ORM quien interactúa con la base de datos, mientras el desarrollador se dedica a manipular objetos.

Además, ofrece un lenguaje de consulta, denominado **HQL** (*Hibernate query language*), de forma que es el propio ORM quien traduce este lenguaje al propio de cada motor de bases de datos, manteniendo así la portabilidad a expensas de un ligero incremento en el tiempo de ejecución.

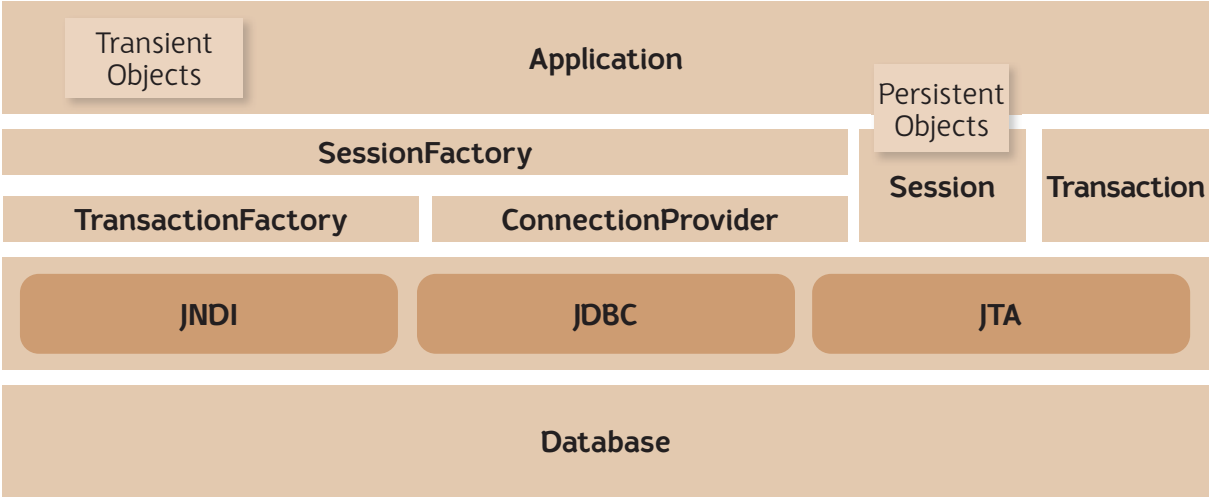


Tabla 1.2. Fuente: <https://docs.jboss.org/hibernate/orm/3.5/reference/es-ES/html/architecture.html>

Tabla 1.3. Arquitectura de Hibernate.

Cuando una aplicación crea objetos, estos están almacenados en memoria, con la volatilidad que esto implica. Dichos objetos se denominan transitorios o **transient**. Con Hibernate, los objetos que tenemos que persistir se «rastrear» en lo que se conoce como una sesión (**Session**), creada a partir de un **SessionFactory**, de acuerdo con la configuración proporcionada. También se proporciona la gestión de conexiones y transacciones.

Como es lógico, los objetos volátiles podrán persistirse, pasando de unos estados a otros, como veremos más adelante, con métodos como **save()** o **persist()**. También se proporciona una interfaz **query**, para lanzar consultas en **HQL** (*Hibernate query language*, su versión del **OQL**).

La parte subyacente de la sesión, como se observa, permite utilizar varias tecnologías, entre ellas **JDBC**, para conectarse a los SGBD necesarios.

Propiedad de McGraw-Hill®

1. Configuración e instalación de Hibernate

Para empezar a utilizar Hibernate no hace falta realizar una compleja tarea de descarga de componentes y demás, ya que Hibernate, como buen **framework**, se integrará en nuestro proyecto en forma de librerías. Podemos optar por descargar las librerías en formato **JAR** y añadirlas a nuestro proyecto, pero esto supone que cada vez que portemos un proyecto debamos recordar añadirlas.

La solución, como siempre, son los gestores de dependencias, que permiten automatizar dichas tareas y, en las construcciones del proyecto, en caso de no estar físicamente en el sistema, descargar las librerías y añadir las referencias.

1.1. Proyecto con Hibernate y MySQL

Supongamos que queremos realizar un proyecto en el cual vamos a utilizar Hibernate como ORM entre nuestro proyecto Java y un SGBDR MySQL. Para ello buscamos las dependencias en el gestor de repositorios de **Maven Central** (los enlaces los tienes en el fichero asociado).

Para trabajar con MySQL e Hibernate, una vez buscadas sus dependencias, añadiremos al fichero correspondiente:

Maven (en POM.XML)

```
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.27</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core -->
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.6.3.Final</version>
</dependency>
```

Gradle (en build.gradle)

```
dependencies {
    // https://mvnrepository.com/artifact/mysql/mysql-connector-java
    implementation group: 'mysql', name: 'mysql-connector-java', version:
    '8.0.27'
    // https://mvnrepository.com/artifact/org.hibernate/hibernate-core
    implementation group: 'org.hibernate', name: 'hibernate-core', version:
    '5.6.3.Final'
}
```

IMPORTANTE

Hibernate ya dispone de la versión 6 en la fecha de redacción del presente contenido, pero aún está en su versión alfa, por lo que es probable que presente algún error.

1.2. Estructura de un proyecto de Hibernate

Bien, ya tenemos las dependencias añadidas. Ahora necesitamos ver los tipos de archivos con los que vamos a trabajar. Solo introduciremos algunos de ellos, que se desarrollarán en los siguientes apartados.

Cabe destacar que, aunque sin formar parte del proyecto de programación, tendremos ya creada muy probablemente la base de datos, a la que deberemos adaptar nuestro proyecto.

A. BEAN (CLASES)

Deberemos crear las clases que representan a nuestros objetos. Inicialmente, con las salidas de los ORM, estas clases se denominaban **POJO** (*plain old java object*). Los POJO son objetos comunes, que no pueden heredar ni implementar clases ni interfaces preestablecidas de Java ni tener anotaciones. Básicamente son clases del tipo: «necesito un libro... creo un libro».

Los POJO no tienen ningún requerimiento de acceso a sus atributos, ni de cantidad ni de tipo de constructores.

Como una extensión de los POJO aparecen los **beans**, cápsulas o granos (de café), que son más restrictivos, y tienen las siguientes características:

- Hacer sus atributos privados.
- Implementar la interfaz **serializable**.
- Acceder a los campos mediante **getters** y **setters** públicos.
- Implementar un constructor por defecto.

Por tanto, los **beans** serán los componentes de la capa de acceso a datos, y representarán a las diferentes entidades con que trabaja nuestra aplicación.

B. FICHEROS DE MAPEADO

Una vez definidas las entidades, necesitaremos el fichero de mapeado para cada **bean**. En dicho mapeado se indica a qué tabla de la base de datos se guardará dicho **bean**, así como con qué columna y tipo de datos debe coincidir cada atributo de este.

Deberá existir pues un fichero de mapeado por cada **bean**. Si el **bean** se llama, por ejemplo, **Empleado.java**, el **bean** asociado de llamará **Empleado.hbm.xml** (hbm por Hibernate mapping). Hablaremos de los mapeados en el apartado siguiente.

C. CONFIGURACIÓN DE HIBERNATE

Teniendo los **beans** y los mapeados, deberemos especificar la configuración de Hibernate, del mismo modo que establecíamos una configuración en las conexiones con las bases de datos para JDBC. Esta configuración contendrá muchas más cosas, de las que destacamos las más relevantes para trabajar.

La configuración de Hibernate puede especificarse de varios modos:

1. Por código.
2. Por fichero **properties**.
3. Por archivo de configuración **XML**.

El tercero es el más utilizado, y el que realizaremos aquí. Hay que comentar que dicho fichero de configuración debe situarse en la raíz del **CLASSPATH** del proyecto, y su nombre debe ser **hibernate.cfg.xml**.

D. RESTO DE CLASES Y APLICACIÓN

Por último, tendremos el resto de clases del programa, así como la aplicación o clase principal. En caso de estar diseñando una aplicación con entorno gráfico, estarían las clases de representación de los datos o las vistas. Del mismo modo, en caso de una aplicación web, faltarían los controladores y los servicios.

1.3. Configuración del proyecto

Vamos a ver con detalle la configuración de Hibernate. Básicamente, tenemos que realizar dos operaciones, configurar el proyecto y cargar dicha configuración al ejecutarlo, configuración que pasamos a desarrollar. Para obtener información de todas las opciones completas, puede consultarse la documentación oficial en el Enlace 2 de la sección de enlaces.

A. FICHERO HIBERNATE.CFG.XML

En la sección Ejemplos para trabajar y analizar tienes disponible el fichero AD_U03_A02_01.zip, dentro del que encontrarás `hibernate.cfg.xml` con todas las opciones. Hay que tener en cuenta las partes del fichero:

- Configuración de la base de datos.
- Para mostrar las consultas SQL se recomienda tenerla a **true**, al menos en los primeros proyectos, para ver cómo se produce en realidad el mapeo de los objetos a consultas SQL.
- La opción de **hbm2ddl** es muy potente, ya que, si partimos solo del modelo orientado a objetos, Hibernate creará la base de datos por nosotros (evidentemente, vacía de datos). Veremos en una práctica posterior que nos aparecerá otra muy interesante, **hbm2java**, que mediante ingeniería inversa nos permitirá crear nuestros beans a partir del diseño relacional.
- Los ficheros de mapeo XML (`<mapping resource="clase2.hbm.xml">`) deben estar junto a las clases Java, en el mismo paquete.
- Los mapeos dentro de las clases (`<mapping class="clase2.java">`) hacen referencia a los propios beans, como veremos en el siguiente apartado.

B. CARGA DE LA SESIÓN Y LA CONFIGURACIÓN. CLASE HIBERNATEUTIL.JAVA

Para cargar la configuración y obtener un objeto de tipo **Session**, deberemos crear una clase que realice esta tarea. Esta clase habitualmente se llama `HibernateUtil.java`, y el código lo tienes disponible dentro del archivo AD_U03_A02_01.zip.

En dicha clase se crea una instancia única (**Singleton**) de la factoría de sesiones, a partir del fichero de configuración indicado. A partir del **SessionFactory** ya podemos empezar a crear sesiones, y transacciones en ellas.

Propiedad de McGraw-Hill

1. Mapeo de objetos. JPA

Hemos visto en el apartado anterior la preparación de un proyecto para conectarse con las bases de datos. Ahora tenemos que empezar a mapear entidades, y necesitamos tener el modelo relacional que vamos a mapear. Para ello partiremos en este primer ejemplo de un supuesto, con la entidad «Pelis» en una base de datos «Cine».

```
CREATE TABLE `Peli` (  
  `idPeli` int(11) NOT NULL AUTO_INCREMENT,  
  `titulo` varchar(45) NOT NULL,  
  `anyo` varchar(45) NOT NULL,  
  `director` varchar(45) NOT NULL,  
  PRIMARY KEY (`idPeli`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

Suponemos que partimos del proyecto con el que finalizamos el apartado anterior, es decir, un proyecto que tiene las dependencias de MySQL y de Hibernate. También disponemos de la clase `HibernateUtil.java` para la gestión de la sesión.

A continuación, vamos a crear el bean que contendrá las Pelis. Lo crearemos en un nuevo paquete al que llamaremos, en inglés, «**Model**», ya que contendrá los beans que conforman el modelo de datos. Dicha clase quedará:

```
package Model;  
import java.io.Serializable;  
  
/**  
 *  
 * @author joange  
 */  
public class Peli implements Serializable{  
  private Long idPeli;  
  private String titulo;  
  private int anyo;  
  private String elDirector;  
  
  public Peli() {  
  }  
  
  public Peli(String titulo, int anyo, String elDirector) {  
    this.titulo = titulo;  
    this.anyo = anyo;  
    this.elDirector = elDirector;  
  }  
  // eliminamos getters, setters y toString  
}
```

1.1. Mapeo de entidades. Archivo de mapeo

Está claro que aún no podemos persistir este tipo de objeto. Para poder persistirlo, debemos crear un archivo externo a la clase, de extensión `.hbm.xml` y con el mismo nombre de la clase. La localización del archivo no importa *a priori*, aunque es buena idea tener las clases del modelo por un lado y los archivos de mapeo por otro. Así pues, crearemos un paquete **ORM** y dentro de él crearemos el archivo `Peli.hbm.xml`.

La sintaxis básica de estos archivos la vemos a partir del ejemplo:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD
3.0//EN" "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="Model.Peli" table="Peli" >
    <id column="idPeli" name="idPeli" type="long">
      <generator class="native"></generator>
    </id>
    <property name="titulo" type="string"/>
    <property name="anyo" />
    <property column="director" name="elDirector" />
  </class>
</hibernate-mapping>
```

La zona del mapeo viene determinada por la etiqueta `<hibernate-mapping>`, donde tenemos:

- **<class>**: indica el mapeo de una clase cuyo nombre está en el atributo **name**, y que se guardará en la tabla indicada por el atributo **table**. Pasaremos ahora a definir los campos de la clase:
 - **id**: este campo representa aquel campo de la base de datos que es la clave principal:
 - a) **column**: indica el nombre que tiene en la base de datos, solo cuando es distinto a...
 - b) **name**: el nombre que tiene en la clase.
 - c) **type**: el tipo de datos. Aquí solo hace falta ponerlo cuando la correspondencia no es equivalente (**String-varchar**, **int-numeric**, **double-decimal**, etc.).
 - **generator**: suele utilizarse cuando dicho campo va a ser generado por la base de datos (se corresponde con **AUTO_INCREMENT**). En los objetos, dicho campo **no** se va a facilitar, y es en el momento de persistir los datos cuando se genera (en la base de datos) y se asigna al objeto (en el programa).
 - **property**: para el resto de campos (que no son claves principales).



AMPLIACIÓN

El resto de opciones de claves ajenas y demás se verán más adelante. Para conocer más opciones, consulta en la sección de enlaces las equivalencias entre tipos.

Puedes ver el uso de esta anotación en el `U3_A3_Ejemplo1.java`, disponible dentro del proyecto `AD_U03_A03_01.zip`, en la zona Ejemplos para trabajar y analizar.



NOTA

Se hablará más en detalle de los métodos `get()` y `save()` aquí utilizados en los siguientes apartados.

1.2. Mapeo de entidades. Anotaciones

Como hemos podido comprobar, el mapeo resulta muy sencillo, y se trata simplemente de indicar las equivalencias. El inconveniente es que requiere que debamos manejar dos ficheros, la clase Java y el fichero de mapeo.

Por eso, podemos «fusionar» dicho mapeo y eliminarlo, y podemos realizar dentro de la clase Java las anotaciones pertinentes, mediante el estándar **JPA (Java persistence API)**. La ventaja es clara: solo manipularemos un fichero, aunque, por el contrario, si deseamos dejar de persistir una clase, nos quedará «emborronada» con las anotaciones.

Veamos cómo quedaría la clase y la comentamos.

```
@Entity
@Table(name="Peli")
public class Peli_Anotada implements Serializable{

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long idPeli;

    @Column
    private String titulo;

    @Column
    private int anyo;

    @Column(name="director")
    private String elDirector;

    . . . . // obviemos el resto de la clase
```

En este caso, se indica que la nueva clase **Peli_Anotada** se comportará como una entidad (**@Entity**), que estará guardada en una tabla de nombre «Peli» (**@Table(name="Peli")**). En el fichero **hibernate.cfg.xml** tenemos que cambiar el tipo de mapeo, que será, de este modo, **<mapping class="Model.Peli_Anotada" />**

En el interior de la clase, simplemente indicaremos el campo que es clave principal con (**@Id**) y que es auto-incremental (**@GeneratedValue**). Para el resto de campos que queremos que sean mapeados automáticamente, bastará con indicarlos con **@Column**. En el caso del campo **elDirector**, como el nombre es distinto en la base de datos, hay que indicarlo explícitamente (**name=director**). Si algún atributo no posee ni **@Column** ni **@Id**, no será persistido en la base de datos.

Ya podemos ejecutar el programa de ejemplo **U3_A3_Ejemplo2.java**, disponible dentro del proyecto **AD_U03_A03_01.zip** (que puedes encontrar en la sección **Ejemplos para trabajar y analizar**), que realiza lo mismo, pero con la clase anotada.

1.3. Componentes (@Embedded)

Una componente se da cuando en Java tenemos una clase como tal, pero dicha clase solo tiene existencia o sentido dentro de otra clase, sin tener sentido en ninguna otra clase. Un ejemplo podría ser la puntuación en IMDB de una película, que solo tiene sentido que se guarde para dicha película.



NOTA

Las componentes sustituyen relaciones uno a uno con restricción de existencia, lo que durante el diseño de bases de datos puede agruparlo todo en una única entidad.

Así pues, mediante anotaciones crearemos una clase y la marcamos como **@Embeddable** y, dentro de nuestra clase **Peli**, marcamos dicho campo componente como **@Embedded**:

```
@Embeddable
public class IMDB
    implements Serializable {
    @Column
    private String url;
    @Column
    private double nota;
    @Column
    private long votos;
}
```

```
// dentro de Peli
@Embedded
private IMDB imdb;
public Peli_Anotada(String titulo,
int anyo,String elDirector, IMDB
imdb) {
    this.titulo = titulo;
    this.anyo = anyo;
    this.elDirector = elDirector;
    this.imdb = imdb;
}
```

Al ejecutar código del archivo **U3_A3_Ejemplo3.java** (disponible dentro del proyecto **AD_U03_A03_01.zip** en la sección **Ejemplos para trabajar y analizar**) por primera vez después de modificar con el atributo **IMDB**, es interesante revisar el *log* de Hibernate:

```
Hibernate: alter table Peli add column nota double precision
Hibernate: alter table Peli add column url varchar(255)
Hibernate: alter table Peli add column votos bigint
Hibernate: insert into Peli (anyo, director, nota, url, votos, titulo)
values (?, ?, ?, ?, ?, ?)
Peli_Anotada{idPeli=78, titulo=La terminal, anyo=2004, elDirector=Steven
Spielberg, imdb=IMDB{url=https://www.imdb.com/title/tt0362227/, nota=7.4,
votos=438348}}
```

Podemos ver que los campos de la clase **IMDB** se han «incrustado» dentro de la tabla **Peli**, y se ha mapeado sin ningún tipo de problema, lo que convierte esta anotación en muy interesante.

Propiedad de McGraw-Hill®

1. Mapeo de relaciones en Hibernate

Estudiados ya los mapeos de las entidades, vamos a completarlos con las necesarias relaciones. Antes de empezar a comentar la cardinalidad de las relaciones, tenemos que considerar el sentido de dichas relaciones, y vamos a revisar el concepto de direccionalidad de las relaciones.

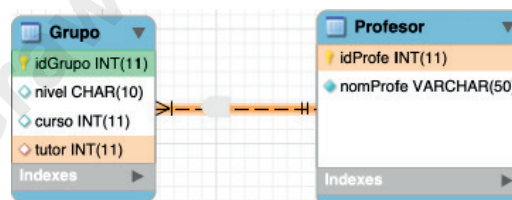
Unidireccional	Una relación es unidireccional cuando accederemos al objeto relacionado (componente) a partir de otro objeto (propietario). Por ejemplo, si en un coche montamos un motor, lo lógico es que el propietario es el coche, y, a partir de él, obtendremos el motor.
Bidireccional	Cuando los elementos relacionados suelen tener la misma «ponderación» o entidad. Por ejemplo, un grupo de un instituto y un tutor. A partir de un grupo, sí tiene sentido tener el tutor, y también podemos, a partir de un profesor, acceder al grupo al cual tutoriza.

IMPORTANTE

Es crucial sensibilizar acerca de las revisiones periódicas en los programas de educación para la salud oral, especialmente en los colectivos de mayor riesgo.

1.1. Relaciones uno a uno (@OneToOne)

Para la explicación de los ejemplos, veremos el diseño e implementación en la base de datos de cada caso y cómo queda en Hibernate. Para este ejemplo vamos a representar una relación 1:1 entre Grupo y Profesor, donde, como se ve, un grupo posee un tutor.



A. RELACIÓN 1:1 UNIDIRECCIONAL

En la tabla de Grupo (columna izquierda) existe un campo «tutor» del que parte una FK a Profesor. La anotación de **referencedColumnName** es opcional. Por defecto, es la PK de Profesor. Como obligatoriamente un grupo debe poseer un tutor, grupo será la propietaria de la relación (desde grupo «parte» o «sale» la clave ajena). Este ejemplo descrito anteriormente es una relación.

```
@Entity
@Table (name="Grupo")
public class Grupo {
    @Id
    @GeneratedValue (strategy =
        GenerationType.IDENTITY)
    @Column (name="idGrupo")
    private Long idGrupo;
    ...
    @OneToOne (cascade=CascadeType.
        ALL)
    @JoinColumn (name = "tutor",
        referencedColumnName = "idProfe")
    private Profesor elTutor;
    ...
}
```

```
@Entity
@Table (name="Profesor")
public class Profesor {
    @Id
    @GeneratedValue (strategy =
        GenerationType.IDENTITY)
    @Column (name="idProfe")
    private Long idProfe;
}
```

B. RELACIÓN 1:1 BIDIRECCIONAL

En el caso de querer mapearla de manera bidireccional, simplemente habría que indicar que en la clase Profesor el grupo que queremos que tutorice. El resultado sería el siguiente:

<pre> @Entity @Table (name="Grupo") public class Grupo { @Id @GeneratedValue (strategy = GenerationType.IDENTITY) @Column (name="idGrupo") private Long idGrupo; ... @OneToOne (cascade=CascadeType. ALL) @JoinColumn (name = "tutor", referencedColumnName = "idProfe") private Profesor elTutor; ... </pre>	<pre> @Entity @Table (name="Profesor") public class Profesor { @Id @GeneratedValue (strategy = GenerationType.IDENTITY) @Column (name="idProfe") private Long idProfe; ... @OneToOne (mappedBy = "elProfe") private Grupo elGrupo; ... </pre>
---	---

Hemos indicado con este nuevo campo que el propietario de la relación es Grupo.

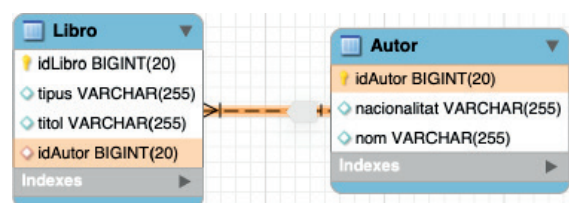
Algunas opciones de Cascade:

- **CascadeType.ALL**: se aplican todos los tipos de cascada.
- **CascadeType.PERSIST**: las operaciones de guardado de las entidades propietarias se propagarán a las entidades relacionadas. Solo se aplica si las entidades se guardan con el método **persist()** en vez de **save()**. Para utilizar **save()** con seguridad, reemplaza **CascadeType.PERSIST** por **CascadeType.SAVE_UPDATE**.
- El resto de opciones, **CascadeType.MERGE**, **CascadeType.REMOVE**, **CascadeType.REFRESH** y **CascadeType.DETACH**, realizan lo que su nombre indica, unir, eliminar, refrescar y sacar de la unidad de persistencia. Hay que tener especial cuidado con el **Remove**, para evitar borrados en cascada.

Puedes revisar el archivo Ejemplo1.java (dentro de AD_U03_A04_01.zip), con un ejemplo de ejecución de la relación 1:1.

1.2. Relaciones uno a muchos (@OneToMany/@ManyToOne)

Para esta explicación partiremos del siguiente modelo, en el cual un Libro posee un Autor que lo ha escrito, y un Autor puede haber escrito varios Libros. En el esquema relacional la relación es de **idAutor** en Libros, que es FK hacia la tabla Autor.



A. RELACIÓN UNIDIRECCIONAL

En este caso, la unidireccionalidad debe aparecer **solo** en la tabla propietaria (Libro). Así pues, en la clase libro indicaremos que contiene un Autor, mapeándola como se ve en el cuadro a continuación. En la tabla Autor no se indicará nada.

En el campo tipo Autor (objeto) se mapea con la anotación **@ManyToOne** (muchos libros escritos por un Autor) a la columna **idAutor**. Se indica que cualquier guardado en Libro provoca que se persista también el Autor (**CascadeType=PERSIST**). También se anota que la FK tendrá como nombre **FK_LIB_AUT**. Veamos qué ocurre al ejecutar el siguiente programa.

B. RELACIÓN UNO A MUCHOS BIDIRECCIONAL

El mapeado de la relación bidireccional nos va a permitir acceder a la información relacionada en ambas direcciones. Para ello añadiremos en Autor un conjunto (**set**) de todos los libros que ha escrito.

<pre> @Entity @Table(name="Libro") public class Libro implements Serializable { @Id @GeneratedValue(...) private Long idLibro; ... @ManyToOne(cascade=CascadeType.PERSIST) @JoinColumn(name="idAutor",foreignKey = @ForeignKey(name = "FK_LIB_AUT")) private Autor elAutor; </pre>	<pre> @Entity @Table(name="Autor") public class Autor { @OneToMany(mappedBy="elAutor", cascade=CascadeType. PERSIST, fetch = FetchType.LAZY) private Set<Libro> losLibros; </pre>
--	---

En Autor, como tabla relacionada no propietaria, se indica que la información de la relación está en el campo «**elAutor**», dentro de la anotación **@OneToMany**. Aparece un nuevo marcador **fetch**, (existente en las colecciones), cuyo comportamiento determinará el momento en que se realiza la carga de los datos de las colecciones, y cuyos valores pueden ser:

FetchType.EAGER	Literalmente, como «modo ansioso». No podemos esperar, y en el momento de la carga del Autor, Hibernate resolverá la relación y cargará todos los libros con todos los datos internos de cada libro. Tenemos todos los datos al momento.
FetchType.LAZY	Carga en diferido; al cargar el Autor, Hibernate solo carga los atributos propios del Autor, sin cargar sus Libros. En el momento en que queramos acceder a dicha información es cuando se realiza la carga de estos. Es decir, en el modo LAZY , los datos se cargan «cuando hacen falta».

En el documento Ejemplo2.java (dentro de AD_U03_A04_01.zip), puedes ver un ejemplo de las relaciones aquí explicadas.

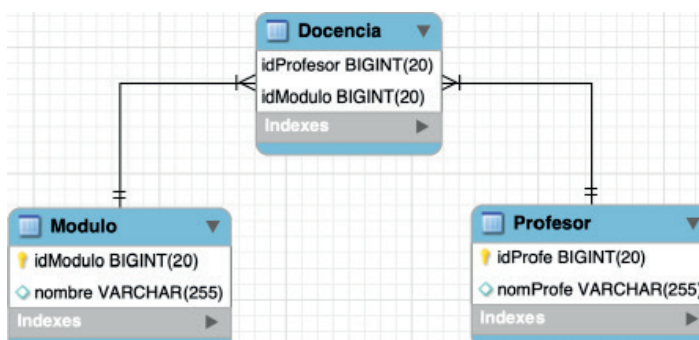
1.3. Relaciones muchos a muchos (@ManyToMany)

Dentro de las relaciones binarias, podemos encontrar dos posibilidades: relaciones que simplemente indican la relación (por ejemplo, que un personaje puede llevar o no tal tipo de arma en un juego de rol) o relaciones que, aparte de indicarla, añaden nuevos atributos a esta (un actor participa en una película con un tipo de papel: principal, secundario, etc.). En el modelo relacional, ambos casos terminaban modelándose como una nueva tabla (con o sin el atributo), pero en el modelado OO:

- En el primer caso, no modelamos una clase para mapear dicha tabla de la base de datos.
- En el segundo caso, debe modelarse con una clase con el atributo de la relación, por lo que la relación N:M entre dos tablas serán dos relaciones uno a muchos, 1:N y N:1, cosa que ya sabemos resolver.

Vamos a modelar el típico caso de un Profesor que imparte varios Módulos, que pueden ser impartidos por varios profesores. El esquema se ve a continuación:

Como podemos observar, queda la típica tabla central de la relación N:M. Como se ha comentado anteriormente, la tabla de Docencia no existirá en el modelado OO, ya que únicamente sirve para relacionar los elementos.



Las clases **Modulo** y **Profesor** quedan como sigue (se muestra solo la parte relacionada con la relación), eligiendo en este caso **Profesor** como propietaria de la relación.

Profesor

```
@ManyToMany(cascade=CascadeType.PERSIST, fetch=FetchType.LAZY)

@JoinTable(name="Docencia",
            joinColumns = {@JoinColumn(name="idProfesor",
                                       foreignKey = @ForeignKey(name = "FK_DOC_PROF" )}),
            inverseJoinColumns = {@JoinColumn(name="idModulo",
                                              foreignKey = @ForeignKey(name = "FK_DOC_MOD" )})

private Set<Modulo> losModulos=new HashSet<>();

public void addModulo(Modulo m) {
    if (!this.losModulos.contains(m)) {
        losModulos.add(m);
    }
    m.addProfesor(this);
}
```

Módulo

```
@ManyToMany(cascade = CascadeType.PERSIST, fetch = FetchType.LAZY,
            mappedBy = "losModulos")

private Set<Profesor> losProfesores=new HashSet<>();

public void addProfesor(Profesor p) {
    if (!this.losProfesores.contains(p)) {
        losProfesores.add(p);
    }
    p.addModulo(this);
}
```

Fíjate en lo siguiente:

- En ambas clases, el mapeo es **@ManyToMany**, conteniendo un **Set** de objetos de la otra clase, indicando las operaciones en cascada (**cascade**) y la carga de los objetos relacionados (**fetch**).
- En la clase propietaria (**Profesor**) se inicia la relación, enlazando con una tabla **Docencia** siguiendo la FK —desde el origen hasta la punta de la flecha—, donde:
 - Se va a enlazar (**joinColumns**) con el campo **idProfe** (**@JoinColumn**).
- Se mapea desde la tabla **Docencia** hasta la entidad de origen **Modulo** de manera inversa (de punta al origen de la flecha):
 - Esto se consigue con **inverseJoinColumns**, enlazando desde el campo **idModulo** (**@JoinColumn**)
- En la clase relacionada (**Modulo**), que no es la propietaria, simplemente le indicamos que la propietaria es **Profesor**, mediante **mappedBy="losModulos"**.

IMPORTANTE

Fíjate en que los métodos **addModulo** de **Profesor** y **addProfesor** de **Modulo** se llaman entre sí. Para evitar entrar en una recursión, se ha realizado un control que frene dicha posible casuística. Además, al estar enlazados garantizamos la persistencia en cascada, tal y como hemos programado las relaciones.

En el documento **Ejemplo3.java** (dentro de **AD_U03_A04_01.zip**), puedes ver un ejemplo de las relaciones N:M aquí explicadas.

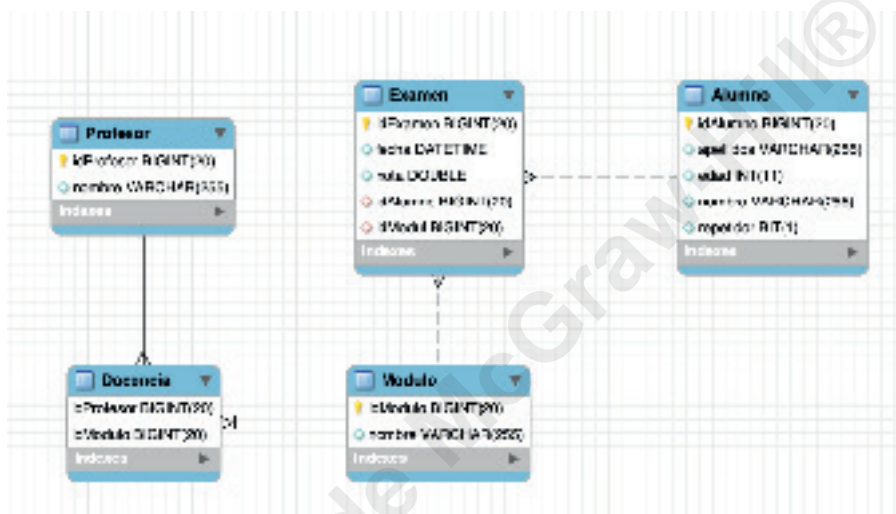
Propiedad de McGraw-Hill®

1. Consultas con HQL

Para concluir con la unidad, necesitamos ver maneras de recuperar información desde la base de datos. El lenguaje **HQL** (*Hibernate query language*) nació con la finalidad de salvar de nuevo el modelo relacional, ya que es un supraconjunto de SQL [ampliación de SQL]. La primera consideración es que, por defecto, su funcionalidad es la de **recuperar objetos de la base de datos**, no tablas, como hacíamos en el lenguaje SQL mediante los **ResultSet**.

Las consultas con HQL se realizarán a partir de una interfaz **Query**, que será el modo donde especificaremos qué queremos recuperar. Opcionalmente podemos añadir a la consulta los parámetros necesarios para su ejecución, para evitar consultas *hard-coded*.

En la sección de Descarga de documentos dispones del archivo complementario **Migración de SQL a HQL**, en donde se especifican detalles generales de HQL. Del mismo modo, en la sección de descargas dispones del fichero **AD_U03_A05_01.zip**, que contiene el modelo de datos para la ejecución de los ejemplos, lo cuales se modelan con la imagen que tienes a continuación.



1.1. Recuperación de objetos simples

Estas consultas son las que permiten recuperar un objeto o colección de objetos desde las bases de datos. Veamos los ejemplos; el primero muestra todos los alumnos:

```
Query q=laSesion.createQuery("Select a from Alumno a"); //todos
// o bien
Query q = laSesion.createQuery("Select a from Alumno a where
a.idAlumno=1"); //un alumno

List<Alumno> alumnos = q.list();

for (Alumno a : alumnos) {
    System.out.println(a);
}
```

Este segundo ejemplo permite obtener solo uno, con lo que se indica en la cláusula **where**.

```
Alumno a = (Alumno)q.uniqueResult();
Query<Alumno> q = laSesion.createQuery("Select a from Alumno a where
a.idAlumno=1",Alumno.class);
Alumno a = q.uniqueResult();
```


IMPORTANTE

En consultas con `uniqueResult`, si el resultado es más de uno, aparecerá la excepción `org.hibernate.NonUniqueResultException`, ya que el `where` no filtra por el identificador.

Cuando los resultados sean muchos, igual no conviene recuperarlos todos de golpe, sino ir accediendo a ellos de diez en diez o de una forma similar, al igual que las páginas de búsqueda de Google o Amazon. Esto lo podemos conseguir lanzando consultas más pequeñas de manera repetida, aplicándole al `query`:

- `Q.setFirstResult(int inicio)`: indica la primera fila que devolverá.
- `Q.setMaxResult(int cuantos)`: indica cuantas filas devuelve.

Con un algoritmo apropiado, podemos realizar un bucle, desplazando en cada iteración el inicio e incrementándolo en la cantidad de filas recuperadas en la anterior iteración. Implicaría muchas consultas pequeñas frente a una grande.

1.2. Consultas mixtas

Entendemos por consultas mixtas aquellas que no devuelven objetos enteros como tales. Estas consultas devolverán, o bien parte de objetos, o bien un objeto más algo más. La novedad es que el resultado será un array de objetos (`Object[]`), y, por lo tanto, deberemos ser muy cuidadosos con el tipo de cada celda, así como con el tamaño de dicho array, ya que estará fuertemente ligado a la propia consulta. Veamos la siguiente consulta, Mostrar nombre y edad de los alumnos:

```
Query q = laSesion.createQuery("Select a.nombre,a.edad from Alumno a");
List<Object[]> losAlumnos = q.list();
for (Object[] alu : losAlumnos) {
    System.out.println("El alumno " + alu[0] + " tiene " + alu[1] + " años");
}
```

Lo que cambia es la manera de procesar el resultado. Al ser un array de `Object`, deberemos acceder a las columnas como si fuera una tabla. En este ejemplo para mostrar, gracias al polimorfismo de Java, no hace falta nada más, pero sí que sería interesante hacer cásting de `Object` a los tipos necesarios para manipular los datos.

1.3. Los múltiples Select en cascada

Cuando realicemos una consulta a una clase que contiene objetos enlazados, habitualmente por relaciones, está consulta generará una consulta anidada por cada objeto enlazado, para cargar su contenido. Este es el comportamiento con una consulta ansiosa (`Eager`). Habrá que valorar el cargarlas en diferido o `Lazy`. Puedes observar este comportamiento en el archivo externo `Ejemplo1.java`, incluido en el fichero `AD_U03_A05_02.zip`, ubicado en la sección `Ejemplos para trabajar y analizar`.

1.4. Consultas sobre colecciones

Vamos a consultar el nombre de los alumnos y cuántos exámenes ha realizado cada uno. Dicha información está en el set de exámenes, por lo que necesitaremos manipular dicha colección:

```
Query q = laSesion.createQuery("Select a.nombre,size(a.losExamenes) from Alumno a");
List<Object[]> losAlumnos = q.list();
for (Object[] alu : losAlumnos) {
    System.out.println("El alumno " + alu[0] + " ha hecho " + alu[1] + " exámenes");
}
```

Como puede apreciarse, hemos aplicado la función **size()** a la colección para ver su tamaño. Podemos aplicar, por tanto:

- **Size(colección)**: recupera el tamaño.
- **Colección is empty | colección is not empty**: para determinar si está vacía. Equivale a comparar el **size** con 0.
- Pueden combinarse los operadores **in**, **all** mediante el operador **elements(colección)**.

1.5. Consultas con parámetros. Consultas nominales

Normalmente, la mayoría de consultas necesitarán unos parámetros, en general para el filtrado de objetos en la cláusula **where**. Ya se comentaron en la unidad anterior las bondades de parametrizar las consultas para evitar problemas de inyección SQL o similares.

La gestión de parámetros se realiza del mismo modo que con las sentencias (**Statements**) y puede realizarse mediante parámetros posicionales o nominales.

A. PARÁMETROS POSICIONALES

Para indicar parámetros posicionales, se marcarán dentro de la consulta mediante el símbolo de interrogación (?1, ?2, etc.), a diferencia de las sentencias del apartado anterior, en las que solo bastaba con el símbolo. Para poder después asignarle el valor al parámetro, Hibernate nos ofrece una amplia batería de métodos sobrecargados, **setParameter(posicion, valor)**, donde valor puede tomar desde los tipos básicos hasta Objeto.

B. PARÁMETROS NOMINALES

Las posiciones están bien, pero si podemos indicar los parámetros de manera nominal quedará un programa mucho más legible. Los parámetros se indican con **nombreParametro** y se asignarán con **setParameter("nombreParametro", valor)**, indicando el nombre del parámetro (sin los dos puntos).

Tienes ejemplos en el archivo externo **Ejemplo2.java** incluido en el fichero **AD_U03_A05_02.zip**, ubicado en la sección **Ejemplos para trabajar y analizar**.

C. CONSULTAS NOMINALES

Es una buena práctica crear las consultas más importantes o que tengamos previsión de que sean las que más se utilizarán junto a la definición de la misma clase, mediante lo siguiente.

Fuera de la definición de la clase se creará una colección **@NamedQueries**, que contendrá un array (indicado con llaves) de elementos **@NamedQuery(nombre="", definicion="")**.

Para invocarlos, en vez de crear un objeto **Query**, lo crearemos mediante un **NamedQuery**, indicando el nombre de este, y asignándole parámetros, si los tuviese. En el archivo externo **Ejemplo3.java**, incluido en el fichero **AD_U03_A05_02.zip**, ubicado en la sección **Ejemplos para trabajar y analizar**, tienes ejemplos de estos queries.

1.6. Inserciones, actualizaciones y borrados

Por último, vamos a analizar el resto de las operaciones CRUD. Hay que comentar que estas operaciones pueden realizarse directamente sobre los objetos; por eso se explican con carácter complementario y no principal. Hacemos notar que, para la inserción, no se permite asignar los valores directamente, sino que solo los podemos obtener de una subconsulta:

```
insert into entidad (propiedades) select_hql
```

La sintaxis de update o delete es similar a la de SQL:

```
update from entidad set [atrib=valor] where condicion  
delete from entidad where condicion
```

Además de todo lo visto:

- Estas instrucciones pueden contener parámetros.
- El **where** es opcional, pero lo borrará o actualizará todo.
- Estas consultas se ejecutan todas mediante **executeUpdate()**, que retorna un entero con el número de filas afectadas.

En el archivo externo **Ejemplo4.java**, incluido en el fichero **AD_U03_A05_02.zip**, ubicado en la sección **Ejemplos para trabajar y analizar**, tienes ejemplos de estos queries.



RECUERDA

En el caso de manipulación de objetos (borrados, modificaciones, etc.) tenemos suficientes herramientas para realizarla sin las consultas HQL, pero implican cargar la información de la base de datos a nuestro programa.

Estas consultas son más adecuadas para procesamiento de grandes volúmenes de información sin tener que recurrir a cargar la información a nuestro programa para tener que procesarla.

Propiedad de McGraw-Hill®



Unidad 4.

Bases de datos orientadas a objetos



1. Introducción. El modelo de datos ODMG

1.1. Evolución de los SGBD

Las **BDR** surgen a partir del modelo relacional, como un sistema que representa fielmente la realidad, con una gran solidez por la lógica relacional subyacente. Dicho modelo representa la perspectiva estática del modelado de la aplicación, y en él se desglosan todos los datos hasta niveles atómicos, ya que, como recordamos, no se permiten ni valores multivaluados ni compuestos.

Dicho modelo ha sufrido el **desfase objeto-relacional**, en el cual los lenguajes de programación realizaron una evolución de las estructuras, adoptando la metodología de la orientación a objetos. Esto provoca que diseñemos por un lado la base de datos, siguiendo el modelo relacional, y que también necesitemos, por otra parte, el diseño de clases de la aplicación. Dichos diseños, que no suelen coincidir, podemos «encajarlos» con las herramientas **ORM** estudiadas en la unidad anterior para mitigar el desfase objeto-relacional.

De este desfase surge la necesidad de añadir un diseño más orientado a objetos dentro de la propia base de datos. Si analizamos un esquema orientado a objetos y tratamos de aplicarle la teoría de la normalización, nos encontraremos que los objetos se nos «descomponen» en varias entidades, lo que provoca que aparezcan muchas tablas.



Esta cantidad elevada de tablas conduce, como consecuencia, a un aumento de las referencias entre estas, lo que incrementa considerablemente las relaciones entre ellas, y, por tanto, un mayor número de restricciones que controlar (**FOREIGN KEY**), lo cual supone un acrecentamiento en el número de comprobaciones que debe realizar el SGBD.

Así pues, las BDOO (bases de datos orientadas a objetos) permiten una definición de tipos complejos de datos, frente a los simples que incorpora el SGBDR, lo que posibilita la definición de tipos estructurados e incluso multivaluados. Con todo esto, las BDOO deberían simplemente permitir el diseño mediante objetos (de forma similar al UML), indicando los objetos que participan, sus atributos y métodos, y las relaciones que afectan a estos, ya sean participaciones o herencias.

Estas BDOO no terminan de despegar, y algunas alternativas que implementan las soluciones comerciales consisten en dotar al sistema de bases de datos relacionales de las capacidades semánticas de la orientación a objetos, con lo que aparecen las **BDOR (bases de datos objeto-relacionales)**.

IMPORTANTE

Recuerda que estos valores estructurados y multivaluados violan la teoría de la normalización del modelo relacional.

Estos valores añadidos a los SGBD tradicionales minimizan el impacto de los ORM y acercan el modelo relacional al diseño orientado a objetos, por lo que los convierte en más cercanos al lenguaje de programación.

1.2. BDOO

En estas bases de datos, todo lo que se almacena son objetos, es decir, entidades con un **estado** (determinado por el valor de sus atributos), que pueden modificarse mediante un **comportamiento** (determinado por las acciones que podemos hacer con ellos, sus métodos) y que posee un identificador único, que hace los objetos diferentes.

Las BDOO deben permitir, entre otras cuestiones:

Identificación de objetos (OID)	El sistema debe independizar su valor respecto de su estado.
Encapsulamiento	Los datos de implementación están en los objetos.
Navegabilidad	Permite acceder, a partir de objetos, a datos almacenados en otros objetos mediante referencias, evitando las uniones típicas del modelo relacional. Los tipos de datos admitidos se pueden dividir en atómicos, estructurados y colecciones, y estas últimas, a su vez, en ordenadas y desordenadas, y pueden contener o no elementos repetidos.
Herencia	Permite la creación de entidades a partir de entidades ya existentes previamente. Este paso tendrá estrecha relación con las especializaciones del modelo relacional, ya que la herencia es la sustituta natural de la especialización en los modelos OO.
Identidad	El sistema debe ofrecer un mecanismo de identificación de objetos, aparte de las claves principales.
Consultas sencillas	El sistema debe proveer un lenguaje sencillo de consultar. Normalmente se basan en SQL, y OQL (<i>object query language</i>) es la variante más reconocida.

1.3. Implementación del estándar ODMG

El ODMG (Object Data Management Group) representa a un conglomerado de compañías de la industria de las bases de datos, que propusieron en su momento un conjunto de características que debían implementar las BDOO. La última versión es ODMG v3.0, publicada en el año 2000. Posteriormente, se ha transferido la tarea de seguir con las especificaciones a **Object Management Group (OMG)**.

En cuanto a rasgos que deben implementar los SGBDOO, según ODMG, son:

- Modelado de datos, según las especificaciones de OMG.
- Lenguaje de definición de objetos: ODL.
- Lenguaje de consulta de objetos: OQL.
- Enlazado con el lenguaje de programación, conocido como *binding*, principalmente para C++ y Java.

1.4. BDOR

En el apartado anterior, hemos visto algunas características generales de las BDOO, las cuales se definen solo con objetos.

La norma ANSI SQL1999 (SQL99, y posteriormente la SQL2003, como continuación de SQL92, en la cual se adaptan las características del modelo relacional) permite añadir características de orientación a objetos a los BDR. Esto permite que SGBDR sólidos hayan «adoptado» e implementado dichas características requeridas por el ODMG. Entre estas características destacan:

- Definición de nuevos tipos de datos por parte del usuario.
- Adaptación para dar cabida a tipos de datos binarios de gran tamaño, como imágenes y documentos.
- Posibilidad de almacenar elementos compuestos, como **arrays**.
- Almacenamiento directo de referencias a otras tablas.
- Definición de objetos y herencia.
- Definición de funciones que manejan las estructuras anteriormente definidas.

Las estructuras de almacenamiento de la información siguen siendo las tablas, aunque el concepto de «fila», tal y como lo conocemos, puede variar, debido a la presencia de datos estructurados.

CONCLUSIÓN

Partiendo de los SGBDR, podemos añadir las capacidades de la orientación a objetos, para conseguir complementar el modelado que deseemos.

Si partimos exclusivamente del modelo OO, y no necesitamos las características relacionales, entonces precisaremos el modelo de una BDOO.

Aun así, ambos sistemas deben mantener las características de persistencia, gestión de transacciones y comunicación con los lenguajes de programación.

Propiedad de McGraw-Hill®



1. Adición de objetos a bases de datos relacionales. PostgreSQL

Visto el contexto de los SGBD existentes, nos centramos en este apartado en **PostgreSQL**, como el sistema gestor de bases de datos objeto-relacionales de código libre que más repercusión ha tenido, y rival directo de la gran Oracle.

IMPORTANTE

Para el trabajo en esta unidad, es importante disponer de un servidor Postgres. Puedes instalarlo en tu sistema de la manera que quieras o mediante contenedor Docker, como se ha visto en unidades anteriores.

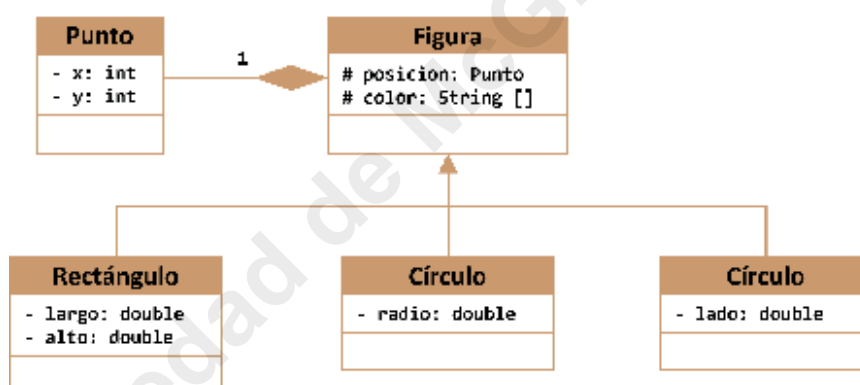
1.1. Definición de tipos en PostgreSQL

En **PostgreSQL** aparecen los tipos habituales que hay en los **SGBD** modernos, pero cabe resaltar los habituales numéricos, textos y fechas, con los que no entraremos en detalles. Destacan, además, tipos especiales para almacenar direcciones de internet (*network address types*), XML y JSON para el guardado y procesamiento de dichos formatos, tipos propios y colecciones. Con esto, podemos fijarnos en los grandes avances de **PostgreSQL**, que, incluso, podríamos considerar como objeto-relacional-documental.



ESQUEMA DE TRABAJO

Para trabajar en los ejemplos siguientes, partiremos de un ejemplo con el que hemos trabajado al principio del curso, que presentamos en el siguiente esquema UML.



A. CREACIÓN DE TIPOS ENUMERADOS

Los tipos enumerados son valores tipos que permiten solo unos valores concretos, habitualmente conocidos también como dominios. En **PostgreSQL** podemos crearlos de la siguiente manera:

```
CREATE TYPE nombre_enumerado AS ENUM  
( [ 'valor' [, ... ] ] );
```

Como ejemplo, podemos crear los posibles colores básicos para pintar unas figuras o tipos de calle para una futura dirección.

```
create type colores_basicos as  
enum( '#FF0000', '#00FF00', '#0000FF' );  
create type TipoCalle as enum ( 'Calle', 'Avenida', 'Plaza' );
```

Esta manera de definir este tipo nos evitará las comprobaciones de valores (cláusulas **CHECK**) existentes en algunos SGBD relacionales.

B. CREACIÓN DE TIPOS ESTRUCTURADOS

Los tipos estructurados son los precursores de los objetos propiamente dichos. Si recordamos, en la programación estructurada, a partir de los tipos básicos, podíamos crear estructuras de datos donde todos

elementos fueran iguales (vectores, arrays y colecciones), o estructuras donde sus elementos podían ser de distintos tipos. Estas estructuras evolucionaron a los actuales objetos al añadirles comportamiento y otras características más.

En el modelo relacional, dado que debemos respetar la atomicidad de los datos, no podíamos generar dichas estructuras. En Postgres podemos crear estos nuevos tipos de datos estructurados con la siguiente sintaxis, muy parecida a la creación de una tabla:

```
CREATE TYPE nombre_tipo AS
( nombre_atributo tipo_de_dato
  [, ... ]      -- uno o varios
);
```

Podemos ver, como ejemplos de definición de tipos:

```
create type Punto as (
  x integer,
  y integer
);
```

Y, además, en el segundo caso, reutilizando los tipos:

```
create type Direccion as (
  tipo TipoCalle,
  calle varchar,
  numero int
);
```

Ya podemos utilizar estos tipos en nuestras tablas, por ejemplo:

```
create table persona(
  idPersona serial,
  nombre varchar,
  direccion Direccion
);

insert into persona(nombre) values ('Joange');
insert into persona(nombre,direccion) values ('Joange',null);
insert into persona(nombre,direccion)
values ('Joange', ('Calle', 'Calvario', 1));

select direccion from persona p;
select (direccion).calle from persona p;
```

IMPORTANTE

A los campos a los que pertenecen los tipos creados no se les puede aplicar restricciones de **NOT**, **NULL**, **DEFAULT** ni **CHECK**.

- La creación de tipos tiene sentido en datos que no tienen existencia por sí mismos, que necesitan ser embebidos en otras estructuras o tablas.
- Al usarse dentro de una tabla y manipular la inserción, esta se hará en bloque, entre paréntesis, ya que determina una estructura.
- Para seleccionar un subtipo, deberemos encerrar el tipo general entre paréntesis, ya que, si no, Postgres lo confunde con una tabla y genera un error.

1.2. Definición de «clases»

Vamos a crear una **clase Figura**, que será el punto inicial de una herencia del modelo presentado al principio de la unidad. Fijémonos en lo que incorpora respecto a las implementaciones del modelo relacional. La **Figura** contiene una clave principal, e incluirá un **Punto** para ubicarla en el plano. Además, contiene una colección de colores, para realizar posibles degradados, guardada como un array. Guardar colecciones es también una capacidad añadida que no admite el modelo relacional, dada la ausencia de multivaluados.

Una clase se crea con la misma sintaxis de una tabla, ya que, a efectos prácticos, es lo mismo desde el punto de vista estructural. Posteriormente, la **herencia** sí que distingue que una tabla «hereda» de otra.

```
create table Figura (
    fID serial primary key, -- identificador
    posicion Punto,        -- posición que ocupa
    color TEXT []           -- color(es) de la figura
);
```

Para insertar nuevos registros, hay que tener en cuenta que:

- Los elementos de tipo **Punto** deben almacenarse mediante un constructor, que crea una «fila» abstracta, denominada **ROW**, o los paréntesis.
- Para los arrays, también necesitamos un constructor denominado **ARRAY**, con una lista de elementos.

```
insert into Figura(posicion,color) values
(row(0,0),array[ 'FFFFFF' , '#00CC00' ] );
```

A partir de ahí, crearemos nuevas clases para representar los círculos, cuadrados y rectángulos a partir de la **Figura**, mediante la herencia. La sintaxis es la siguiente:

```
create table tabla_heredera(
    -- definición de atributos de la tabla
) inherits (super_tabla);
```

Como podemos observar, simplemente añadimos inherits para crear la relación de herencia. Para el diseño que teníamos anteriormente:

<pre>create table Rectangulo(alto int, ancho int) inherits (Figura);</pre>	<pre>create table Cuadrado(lado int) inherits (Figura); create table Circulo(radio int) inherits (Figura);</pre>
--	---

Insertamos algunas filas, fijándonos en que tenemos que incluir también los atributos de la superclase.

```
insert into Cuadrado(posicion,color,lado) values
(row(10,10),array[ '#00BBCC' , '#BBCC00' ],40);
insert into Cuadrado(posicion,color,lado) values
(row(10,15),array[ '#AA6633' , '#CCFF00' ],27);
insert into Circulo(posicion,color,radio) values
(row(30,25),array[ '#BBCC' , '#CCCC00' ],20);
insert into Circulo(posicion,color,radio) values
(row(10,-10),array[ '#00BBCC' , '#CCCC00' ],20);
insert into Rectangulo(posicion,color,alto,ancho) values
(row(10,5),array[ '#00BBCC' , '#CCCC00' ],20,50);
insert into Rectangulo(posicion,color,alto,ancho) values
(row(30,-10),array[ '#00BBCC' , '#CCCC00' ],20,50);
```

Si visualizamos las inserciones con DBeaver, observamos las agrupaciones de tipos:

figura Enter a SQL expression to filter results (use Ctrl+Space)					
Grilla	fid	posicion		color	
		123 x	123 y		
1	1	2	3	{#AABBCC,#FFCC00}	
2	2	0	0	{FFFFFF,#00CC00}	
3	3	-2	7	{#AABB00,#FFCCCC}	
4	7	10	5	{#00BBCC,#CCCC00}	

rectangulo Enter a SQL expression to filter results (use Ctrl+Space)						
Grilla	fid	posicion		color	123 alto	123 ancho
		123 x	123 y			
1	7	10	5	{#00BBCC,#CCCC00}	20	50
2	8	30	-10	{#00BBCC,#CCCC00}	20	50
3	9	-10	15	{#00BBCC,#CCCC00}	20	50
4	10	15	5	{#00BBCC,#CCCC00}	20	50

cuadrado Enter a SQL expression to filter results (use Ctrl+Space)					
Grilla	fid	posicion		color	123 lado
		123 x	123 y		
1	4	10	5	{#00BBCC,#CCCC00}	20
2	11	10	10	{#00BBCC,#BBCC00}	40
3	12	10	15	{#AA6633,#CCFF00}	27

circulo Enter a SQL expression to filter results (use Ctrl+Space)					
Grilla	fid	posicion		color	123 radio
		123 x	123 y		
1	5	30	25	{#BBCC,#CCCC00}	20
2	6	10	-10	{#00BBCC,#CCCC00}	20

Como es lógico, cabe pensar que al seleccionar datos de la tabla general (**Select * from Figura**) aparecerán todos los elementos de las subclases. Si quisiéramos seleccionar solo las que son Figura, podríamos hacerlo con (**Select * from ONLY Figura**).

Vamos a completar el ejemplo creando un dibujo con todas las figuras que tenemos almacenadas. El dibujo lo almacenaremos en una nueva clase que contiene el identificador del dibujo y nos guardaremos una colección con los identificadores de las figuras que forman el dibujo.

```
create table Dibujo(
    idDibujo serial primary key,
    elementos int[]
);

insert into Dibujo (elementos) values (ARRAY[2,4,5,6]);
insert into Dibujo (elementos) values (ARRAY (select fid from figura));
```

Hay que comentar que la selección de los identificadores de Figura puede ser directa, o bien seleccionando aquellas que deseemos, mediante una consulta embebida dentro del constructor de **ARRAY**. Esto puede realizarse cuando la **select** devuelve una sola columna.



Si queremos obtener los dibujos que hemos insertado (`select * from dibujo`), se nos muestran los elementos como una colección de figuras (un `array`).

select * from dibujo			Enter a SQL expression to filter results
Grilla	123 id dibujo	elementos	
1	1	{2,4,5,6}	
2	2	{1,2,3,7,8,9,10,4,11,12,5,6}	

Podemos deconstruir el vector, para así poder acceder a cada una de las figuras que lo componen, mediante la función `unnest`.

```
select id dibujo, unnest(elementos) from dibujo ;
```

Se proporciona toda la implementación en el fichero adjunto `AD_U04_A02.zip`, en el apartado Ejemplos para trabajar y analizar.

select id dibujo, unnest(elementos) from dibujo			Enter a SQL expression to filter results
Grilla	123 id dibujo	123 unnest	
1	1	2	
2	1	4	
3	1	5	
4	1	6	
5	2	1	
6	2	2	
7	2	3	
8	2	7	
9	2	8	
10	2	9	
11	2	10	
12	2	4	
13	2	11	

Propiedad de McGraw-Hill®

1. BSOO nativas. ObjectDB

En este apartado, como **SGBDOO** se ha optado por **ObjectDB**, ya que es muy versátil, es gratuito e incluso permite embeberse dentro de nuestros proyectos Java, lo que proporciona una gran simplicidad para el desarrollo de aplicaciones pequeñas, debido a la eliminación de un servidor.

1.1. Instalación y acceso a ObjectDB

ObjectDB no requiere instalación como tal, ya que todo su código va integrado en una API de acceso a la base de datos, empaquetado en un fichero **JAR**. Desde la web oficial podemos descargarnos el **ObjectDB Development Kit**. En el momento de la redacción de esta unidad, la versión es la 2.8.6. Este kit contiene, entre otros:

- Dependencias para proyectos Java.
- Utilidades para la visualización y consulta de la BD.
- Servidor para aplicaciones distribuidas.
- Documentación.

Una vez descomprimido, solo necesitaremos la máquina virtual de Java instalada en nuestro sistema para ejecutar todos los elementos. Para utilizar **ObjectDB** en nuestro proyecto, deberemos añadir el archivo **objectdb.jar** a las dependencias de nuestro proyecto, o bien realizarlo mediante el gestor de dependencia de **maven** o **gradle**.

En este momento, ya podemos conectarnos a la base de datos; la centralización de la conexión a la base de datos se realiza mediante una instancia de un objeto **EntityManagerFactory**, a partir del cual podemos obtener varias instancias de un **EntityManager**.

A partir del **EntityManager**, ya podremos realizar las típicas operaciones CRUD, teniendo en cuenta que, siempre que existan modificaciones en esta, deberemos realizar la operación dentro de una transacción para evitar situaciones inconsistentes en ella. Aquí vemos una posible clase con el establecimiento de la conexión y la obtención de un **EntityManager**. Este código puede verse en el fichero **AD_U04_A03.zip**, que encontrarás en la sección Ejemplos para trabajar y analizar.

1.2. Creación y persistencia de objetos

A. PERSISTIR CLASES

Para persistir un objeto, necesitaremos:

- Anotar su clase y marcarla como **@Entity**.
- Definir un campo como identificador **@Id**, y, opcionalmente, que sea autoincremental con **@GeneratedValue**.
- El resto de atributos de la entidad, por defecto, se persisten automáticamente sin ningún tipo de anotación. En caso de no querer persistir alguno, podemos indicarlo con **@transient**.

```
@Entity
public class Alumno {
    @Id @GeneratedValue
    private Long id;
    private String nombre;
}
```

Para almacenar un **Alumno**, bastará con crear un **Alumno** y persistirlo en la base de datos, como se ve a continuación, suponiendo el objeto con de tipo **conexionODB** visto en el fichero adjunto.

```
EntityManager em= con.getEM();
em.getTransaction().begin();
```

```
Alumno alu=new Alumno("Antonio Ramos");
em.persist(alu);
em.getTransaction().commit();
```

B. CLASES EMBEBIDAS O COMPONENTES

En Java existen en ocasiones clases que no tienen existencia propia, a menos que existan dentro de otra clase, como, por ejemplo, una clase Dirección. No tiene sentido crear un objeto Dirección *ad hoc*; en cambio, sí que tiene sentido crearlo de manera que exista una Dirección, por ejemplo, dentro de un Alumno.

Estas cases (débiles) que existen embebidas dentro de otras clases debemos declararlas como embebibles o incrustadas mediante la anotación `@Embeddable` y marcarlas como embebidas (`@Embedded`) dentro de la clase en que existen.

<pre>@Embeddable public class Direccion { ... }</pre>	<pre>@Entity public class Alumno { ... @Embedded private Direccion direccion; }</pre>
<pre>// en main Alumno alu= new Alumno("Joan Gerard"); Direccion d= new Direccion("C/ del calvario"); alu.setDireccion(d); em.persist(alu);</pre>	

En la base de datos se almacenará una entidad Alumno, pero la dirección no existe como objeto en sí mismo.

1.3. Relaciones

A. UNO A UNO

La más sencilla es la 1-1, en la cual un objeto contiene a otro objeto. Lo marcaremos, como ya hacíamos en Hibernate, con el modificador `@OneToOne`, indicando que el guardado sea en cascada (`cascade=CascadeType.PERSIST`).

A partir de este momento, al guardar un instancia de un objeto, se guardará una instancia propia del objeto relacionado y se enlazará. El objeto enlazado tendrá existencia en sí mismo (en caso de estar marcado como `@Entity`). Un ejemplo básico es que una clase tiene un único Tutor; basándonos en el caso en que una Clase (de un instituto) tiene asociado un Tutor, el ejemplo será el siguiente:

<pre>@Entity public class Profesor { ... }</pre>	<pre>@Entity public class Clase{ ... @OneToOne (cascade=CascadeType.PERSIST) private Profesor elTutor; }</pre>
<pre>Profesor p=new Profesor("Pepe"); Clase c= new Clase("2DAM"); c.setTutor(p); em.persist(c); //al guardar la clase se guarda el tutor</pre>	

B. UNO A MUCHOS

Vamos a referirnos ahora a una relación clásica, en la que un Profesor es el tutor de varios Alumnos. Estas relaciones pueden ser unidireccionales o bidireccionales. En este ejemplo, la veremos bidireccional, de manera que, dado un Alumno, podemos saber quién es su Profesor tutor, y, dado un Profesor, podemos obtener los alumnos que tutoriza:

```
@Entity
public class Alumno {
    ...
    @ManyToOne(cascade=CascadeType.PERSIST)
    private Profesor elTutor;
}

@Entity
public class Profesor {
    ...
    @OneToMany(cascade=CascadeType.PERSIST, fetch=FetchType.EAGER)
    private List<Alumno> losAlumnos;
}
```

Hay que fijarse en que es muy similar a Hibernate, tanto en el **CascadeType** como en el **FetchType**. Los usos son iguales que en **Hibernate**; **Eager** es el modo de carga anticipada o «ansiosa», y **Lazy**, la carga en diferido o «vaga».

C. MUCHOS A MUCHOS

Las relaciones muchos a muchos podemos enfocarlas de varios modos. Pongamos el ejemplo de la docencia entre Profesores y Alumnos. Si simplemente queremos indicar quién da clase a quién, bastaría guardar una colección de profesores en cada alumno (los profesores que le dan clase a dicho alumno), y, de manera simétrica, en cada profesor una colección de alumnos (los alumnos a los que imparte clase). En este caso, sería una bidireccional, ya que desde una clase podemos navegar hasta la otra, lo que queda de este modo:

```
@Entity
public class Alumno {
    ...
    @ManyToMany(cascade=CascadeType.PERSIST, fetch=FetchType.LAZY)
    private Set<Profesor> profesores=new HashSet<>();
}

@Entity
public class Profesor {
    ...
    @ManyToMany(cascade=CascadeType.PERSIST, fetch=FetchType.LAZY)
    private Set<Alumno> losAlumnos=new HashSet<>();
}
```

Si necesitamos guardar dentro de dicha relación alguna información, como, por ejemplo, las notas que ha recibido el alumno, o las incidencias puestas, entonces debemos crear una nueva clase, que incorporará los atributos propios de la relación, y realizar relaciones 1 a M desde cada entidad (Alumno/Profesor) hacia la nueva entidad (Docencia). Este supuesto es el famoso «Las relaciones N-M generan tabla con los atributos que poseen» del modelo relacional.

1.4. Consultas

Vamos a revisar cómo podemos cargar los datos que hemos guardado previamente en la base de datos. Supongamos que tenemos una clase `Alumno`, mapeada con entidad y con identificador (`idAlumno`). La manera más sencilla de cargar un `Alumno`, conocido su ID, es el método `find(class, id)`:

```
Alumno a=em.find(Alumno.class, 2);
```

Busca en aquellas entidades de dicha clase la que tiene dicho identificador.

1.5. Queries y TypedQueries

El resto de cargas debemos realizarlas mediante consultas, en un lenguaje JPQL (*Java persistence query language*), que, de nuevo, es similar al HQL de Hibernate. Lo que cambian son los tipos de datos para montar dichas consultas.

Existen dos clases `Query` y `TypedQuery` (la segunda hereda de la primera), que habitualmente se usan, en el primer caso, cuando desconocemos el resultado de la consulta, y en el segundo, cuando sabemos el resultado. La primera es polimórfica; por tanto, hará un enlace dinámico de resultados, y la segunda verifica el resultado con la clase actual. En la documentación oficial se recomienda el uso de la segunda, `TypedQuery`, para consultas. El `Query` lo utilizaremos, sin embargo, para actualizaciones y borrados.

Creación del Query (q) o del TypedQuery (tq)

<code>Object q.getSingleResult();</code> <code>T tq.getSingleResult();</code>	Queries que devuelven un solo objeto. Genérico o concreto.
<code>List q.getResultList();</code> <code>List<T> tq.getResultList();</code>	Queries con múltiples resultados. List genérico o List concreto.
<code>q.executeUpdate();</code>	Para borrados y actualizaciones.



EJEMPLO

```
TypedQuery<Alumno> tq=em.createQuery("Select a from Alumno a where  
a.ampa=true", Alumno.class);  
List<Alumno> alumnosAmpa=tq.getResultList();
```

Para evitar consultas *hard-coded*, podemos parametrizarlas de manera nominal:

```
TypedQuery<Alumno> tq=em.createQuery("Select a from Alumno a where  
a.ampa= :ampa", Alumno.class);  
tq.setParameter("ampa", true);  
List<Alumno> alumnosAmpa=tq.getResultList();
```

Podemos también reaprovechar las consultas, creando consultas etiquetadas. Se define una consulta y se etiqueta. Posteriormente, se puede invocar indicando la etiqueta y la clase. Pueden contener parámetros.

```
@NamedQueries({  
    @NamedQuery(query = "Select a from Alumno a where a.nombre = :name",  
        name = "find alu by name"),  
    @NamedQuery(query = "Select a from Alumno a", name = "find all alu")  
})  
...  
TypedQuery<Alumno> tq=em.createNamedQuery("get all alu",Alumno.class);  
List<Alumno> alumnosAmpa=tq.getResultList();
```

1.6. Borrado y actualización

Para terminar, vamos a revisar las operaciones CRUD que nos quedan. Las actualizaciones son totalmente transparentes al usuario, ya que cualquier modificación que se realice dentro del contexto de una transacción será automáticamente guardada al cerrarla con un **commit()**. Además, pueden realizarse **Queries** de actualización.

Para los borrados, si el objeto ha sido recuperado de la base de datos, y por tanto está en la transacción actual, puede eliminarse con **em.remove(objeto)**. Se borrará de la base de datos al realizarse el **commit**. Si dicho objeto está referenciado por alguna relación, deberá revisarse el **CascadeType**, para no producir eliminaciones en cascada indeseadas.

Propiedad de McGraw-Hill®



Unidad 5.

Generación de servicios de acceso a datos. Spring



1. Introducción de componentes de software

1.1. Componentes de software

Cuando comienzas a desarrollar un proyecto, debes realizar un análisis del problema que hay que resolver. Entran en juego todos los conceptos que has estudiado sobre análisis de requerimientos, modelado de datos, etc. Todo ello, independientemente de los lenguajes que vas a utilizar.

Muchas veces nos centramos en la resolución del problema, y perdemos de vista detalles del mantenimiento de la aplicación que desarrollaremos, su escalabilidad y la integración con otras aplicaciones y sistemas. Por lo tanto, existe «algo más» por encima de la programación orientada a objetos, el modelo relacional o el paradigma que utilicemos. Es la programación orientada a componentes, enfocada a fusionar aplicaciones distribuidas, lo que permite la reutilización de componentes, principalmente de acceso a datos.

Un componente (SW o HW) se refiere a una parte de la solución que queda detallada por lo que hace y por cómo se comunica con su alrededor, es decir, su **comportamiento** y su **interfaz**. Desde el principio de la ingeniería del software, se persigue trabajar y programar con esa perspectiva, reutilizando y adaptando componentes que sabemos que sirven en otros proyectos. Eso implica poder desarrollar estos componentes por separado, de manera que después puedan «ensamblarse» con otros sin perder funcionalidad. Esto nos permitirá el desarrollo de software de manera más rápida y fiable.

Son características de un componente, entre otras:

- **Encapsulado e independencia:** para realizar su tarea no necesita de otro componente, y solo se comunica a través de su interfaz.
- **Independencia:** de software, hardware y plataforma.
- **Reutilizable:** puede utilizarse en distintas aplicaciones.

Los componentes se crearon inicialmente para agilizar el desarrollo de aplicaciones gráficas, gracias a Microsoft, con sus modelos OLE y COM, que evolucionaron hacia las componentes OCX, las cuales desembocaron en el actual .NET.

Por su parte, en el mundo Java destacan las **JavaBeans** para la **Java Standard Edition** y los **EJB (Enterprise Java Beans)** para la **Java Enterprise Edition**. Estas son clases que cumplen ciertos requisitos formales, como **getters**, **setters**, **datos simples**, etc.

1.2. Aplicaciones web

Hoy en día, las aplicaciones no solo se ciñen a los entornos de escritorio. Móviles, tabletas y dispositivos **IoT (Internet of the things)** han propiciado desarrollos para cada plataforma, lo que dificulta a veces el despliegue de la misma aplicación para cada plataforma. Las aplicaciones web son herramientas que permiten el acceso a servidores web (y, por ende, a los datos) a través de internet mediante una aplicación cliente. Habitualmente, ese cliente será el **navegador** donde se ejecutan dichas aplicaciones.



NOTA

Aunque estamos situados en un ciclo de desarrollo de aplicaciones multiplataforma, el hecho de que desarrollemos una aplicación web, o parte de ella, no implica que vayamos a desarrollar contenido web puramente dicho. En este tema nos vamos a centrar en el acceso y retorno de los datos a aplicaciones que los soliciten, ya sean móviles, clientes, navegadores o de cualquier índole.

La estructura de una aplicación web se divide en dos partes:

Frontend	Es la parte del desarrollo que está visible al usuario, y que gestiona la interfaz gráfica de usuario (GUI). El especialista <i>frontend</i> se encarga del diseño gráfico de la interfaz, de la interacción con el usuario y de mejorar la UX (<i>user experience</i>). En aplicaciones web, utilizaríamos HTML, CSS, Angular y JavaScript embebido en el propio navegador. Para aplicaciones web híbridas, con una aplicación de escritorio, la tarea de desarrollo de la interfaz gráfica podría ser mediante Python o JavaFx , o incluso mediante tecnologías móviles para Android e IOS.
Backend	Es la parte de desarrollo oculta al usuario. Habitualmente se ejecuta en un servidor web, y está en comunicación con un servidor de la base de datos. El <i>backend</i> se encarga de recibir las peticiones desde el <i>frontend</i> , y facilitar los resultados solicitados. Existen multitud de tecnologías aplicables para este desarrollo: lenguajes como Java , NodeJS y Rust permiten el desarrollo de estos servidores.

IMPORTANTE

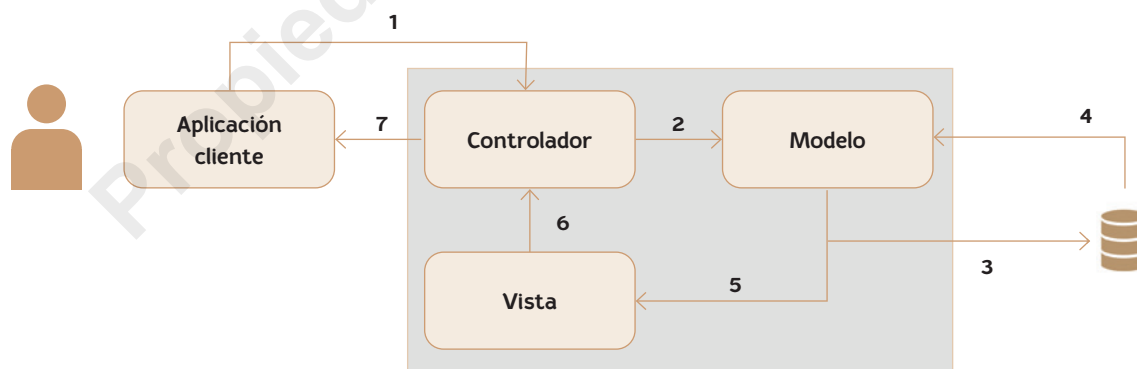
Fíjate en el paralelismo de este modelo con la implementación de servicios, que habrás estudiado en el módulo de programación de servicios y procesos:

- El usuario interacciona con el cliente y se comunica con el servidor mediante **sockets**.
- El servidor recoge la petición desde el **socket**, la procesa y le da una respuesta.
- Ambos deben ponerse de acuerdo en el protocolo y el formato de intercambio de mensajes.

Aparte de *backend* y *frontend*, aparece el **desarrollo fullstack**, el desarrollo por completo ambas partes. Este concepto es complejo, debido a la rápida evolución de las tecnologías, lo que hace muy complicado conocer completamente las dos visiones de la aplicación. Aun así, debemos **entender ambas partes del diseño** y de la estructura, en vez de ver la otra parte como una caja negra y no preocuparnos de lo que ocurre dentro.

1.3. Modelo vista controlador

El modelo vista controlador (MVC) es una arquitectura para el desarrollo de aplicaciones, que separa en capas los datos, la interfaz y la lógica de los controles.



1. El usuario, al utilizar la aplicación cliente (habitualmente un navegador), realiza una petición a través del protocolo http (HTTP_REQUEST). Estas peticiones las recibe el módulo del controlador. El controlador gestiona un listado de operaciones o eventos que puede procesar, determinado por el propio protocolo de este.
2. En caso de ser alguna petición que solicita alguna consulta o modificación de los datos, pasamos a la capa del modelo los parámetros proporcionados en la petición. Esta capa accede a los datos, realizando los pasos 3 y 4 mediante operaciones al SGBD. El modelo está ahora en disposición de dar una respuesta.
3. Consulta a la base de datos.
4. Recuperación de la consulta a la base de datos.

5. El modelo transfiere a la vista el conjunto de resultados obtenidos.
6. La vista es la responsable de recibir los datos y presentarlos o formatearlos de manera adecuada.
7. El controlador devuelve la presentación generada por la vista (que habitualmente será un HTML dinámico) a la aplicación cliente, mediante la petición HTTP_RESPONSE.

A. MODELO REST

El modelo **REST** (*representational state transfer*) nos permite dotar a nuestro servidor de un servicio para recuperar y manipular los datos de manera sencilla. En dicho modelo, se delega la parte de la vista al cliente, y quedan en el servidor el controlador y el modelo. En dicho modelo, destacaremos:

- El protocolo sigue el modelo cliente/servidor sin estado (al igual que HTTP): una petición solo contestará según la información recibida en la misma petición.
- Soporte de operaciones CRUD, mediante las especificaciones HTTP equivalentes: **GET** (consultar), **POST** (crear), **PUT** (modificar) y **DELETE** (borrar).
- Disposición del modelo **HATEOAS** (*hypermedia as the engine of application state*, hipermedia como motor del estado de la aplicación). Este principio permite incluir en las respuestas los hiperenlaces a los recursos.

En comparación con el MVC, sería como eliminar la capa de la vista y devolver los datos procesados por el modelo. Habitualmente, los servidores REST devuelven los datos en formato JSON.

Ejemplo de respuesta a la petición <https://swapi.dev/api/people/1> (API pública de Star Wars).

Como puedes observar, al incluir enlaces en la propia API REST se cumple del principio de HATEOAS, para acceder a objetos a partir de la propia respuesta.

1.4. Spring Boot

Para concluir, veremos algunas características del entorno de trabajo escogido para la presente unidad, ya que nos cubrirá todas las partes analizadas.

Debemos entender el concepto de **framework** (banco de trabajo): conjunto de herramientas, reglas y convenciones para aumentar la velocidad de desarrollo, automatizando la velocidad de desarrollo, mediante la automatización de tareas. No hay que confundir *framework* con IDE, aunque en ciertas ocasiones los IDE se adaptan a los *frameworks* mediante ciertos **plugins**.

Spring inicialmente simplifica el desarrollo de aplicaciones Java, incorporando mecanismos como estos:

- **Inyección de dependencias mediante la inversión de control:** básicamente, el programador delega al *framework* la tarea de creación de objetos en sus clases, y es este último el encargado de cargar los objetos e inyectarlos a las clases que los necesitan cuando son necesarios.
- **Gestión sencilla de componentes POJO y BEANS.**
- **Programación orientada a aspectos:** separa las diferentes tareas que debe realizar una clase en nuestra aplicación.
- **Adición de comportamientos a las clases mediante el uso de proxies:** el programador creará objetos básicos, pero, mediante el preproceso de estos proxies, añadiremos comportamientos (métodos) comunes a nuestros objetos.

Spring Boot apareció años después, con el objetivo de completar o mejorar a Spring, ya que Spring *framework* sigue funcionando de manera subyacente a Spring Boot. Spring Boot es un acelerador para la creación de proyectos, aplicando el patrón de **convención antes de la configuración**. Este patrón reduce el número de acciones que tome el desarrollador en el momento de la creación del proyecto. El programador decide sobre el comportamiento de su futura aplicación y Spring Boot seleccionará automáticamente las dependencias que necesita. Si se requiere alguna más, dota de flexibilidad al programador para añadir o configurar, pero el objetivo es minimizar el tiempo en esta parte.

El flujo de trabajo en el desarrollo de aplicaciones web es:

1. **Seleccionar comportamientos** (y para ello seleccionar todas las dependencias para conseguir dicho comportamiento).
2. **Desarrollar** la aplicación.
3. **Desplegarla** en un servidor.

Spring Boot automatiza prácticamente las fases 1 y 3, ya que incluye un servidor **Tomcat** embebido, y, al lanzar la aplicación, pondrá en marcha dicho servidor. Esto facilita mucho el trabajo y el despliegue mediante contenedores (Docker o similares) en servicios almacenados en la nube.

Propiedad de McGraw-Hill®

1. Instalación y configuración de Spring

Spring es un *framework* para el desarrollo de aplicaciones, y Spring Boot permite acelerar aún más el proceso de preparación y despliegue de los proyectos Spring. Es decir, podemos añadir **plugins** a nuestros editores favoritos, como Eclipse, Netbeans o Visual Studio Code, pero en nuestro caso vamos a utilizar **Spring Tool Suite**.



Spring Tools Suite es básicamente una distribución de Eclipse, preparada exclusivamente para el trabajo con Spring, lo que nos evita las tareas de añadir versiones, configurar y administrar los **plugins** (complementos) necesarios. Podemos descargarla desde la página oficial, disponible en la sección de enlaces de la unidad.

Esta versión de Eclipse se instala como un **bundle** (empaquetado o autocontenido), incorporando en la misma carpeta donde descomprimos la aplicación tanto los ficheros ejecutables como complementos y herramientas adicionales. Existen versiones gratuitas para Linux, Mac y Windows.

1.1. Crear un proyecto. Estructura

Una vez que hemos instalado la aplicación, y que arrancamos el programa y configuramos la carpeta de trabajo (donde se crearán nuestros proyectos), debemos empezar con la creación de este. Aquí es donde aparece la potencia de Spring Boot, ya que no crearemos un proyecto y tendremos que añadir las dependencias que necesitemos, sino que, en el asistente mismo, le indicaremos cómo deseamos que se comporte nuestra aplicación, y el asistente añadirá las dependencias necesarias para su desarrollo.

NOTA

Por supuesto, podremos añadir nuevas dependencias, aunque no hayamos especificado algún comportamiento, ya que Spring gestiona sus proyectos mediante Maven de manera subyacente. No tendremos más que acudir al fichero POM.XML y añadirla sin ningún problema.

En el momento de la creación, debemos especificar las características del proyecto y añadir los comportamientos deseados. En esta unidad, trabajaremos con Web y DevTool.

Durante el asistente, se nos pregunta por estos apartados, necesarios al ser un proyecto Maven. Estas opciones serán usadas para referenciar el proyecto de manera completa.

- **groupID:** viene a ser la organización que desarrolla.
- **artifactID:** el contenido que estamos desarrollando.
- **versión:** la versión de este.

Una vez creado el proyecto, Spring realizará las siguientes tareas:

- Generará el fichero **pom.xml**, con las características indicadas en el asistente.
- Descargará las dependencias.
- Creará la jerarquía de las carpetas necesarias para trabajar.
- Añadirá los comandos de compilación embebidos dentro del **mvnw** (*maven wrapper*).

Con esto, nos quedará una estructura mínima como sigue:

- **/src:** contiene el código completo de la aplicación.
 - **main/java:** contendrá el código Java de nuestro proyecto. Aquí ya veremos que lo separaremos en distintos paquetes según la funcionalidad de las clases (modelo, controladores, etc.).
 - **main/Resources:** contendrá el código estático de nuestra aplicación, como pueden ser páginas HTML, estilos, etc., así como ficheros de propiedades y de configuración.
- **/target:** contendrá nuestro proyecto una vez empaquetado para distribuir.

Debemos indicar la clase que funcionará como controlador mediante anotaciones, tanto a la clase como a los métodos:

Anotación	Significado
@Controller Afecta a Clase	Hereda de @Component , lo que permite que sea detectada explorando el classpath de la aplicación. Contendrá manejadores de eventos.
@RequestMapping("ruta") Afecta a Clase	Define que todos los métodos dentro de esta clase estarán mapeados dentro de una ruta a partir de la URI de la aplicación; por ejemplo, <code>http://localhost/ruta/</code> .
@XXXMapping("subRuta") Afecta a Método	Indica que dicho método mapeará la petición http-XXX (donde XXX es Get , Post , Put o Delete). Dentro del cuerpo de dicho método, programaremos la lógica de cómo tratar dicha petición.
@ResponseBody Afecta a Método	Indicamos al método que el objeto que devolverá se serialice a JSON y se devuelva a quien formalizó la petición. Con la ausencia de esta anotación se devolverá un Modelo con los datos empaquetados para que la vista los renderice.
@RestController Afecta a Clase	Esta anotación se utiliza cuando todos los mapeos de dicho controlador son peticiones REST . Nos permitirá eliminar de los manejadores de peticiones la anotación @ResponseBody .



NOTA

En versiones antiguas de Spring hay un método genérico donde indicamos, como parámetros, la ruta y el método que debe activarlo: **@RequestMapping(value="ruta", method=RequestMethod.XXX)**, donde **XXX** será **Get**, **Put**, **Post**, **Delete**.

1.3. Recoger parámetros de las peticiones

Visto ya cómo podemos mapear las peticiones HTTP de los usuarios, nos queda ver cómo podemos recuperar los argumentos que vendrán con dicha petición. Estos argumentos los definiremos como argumentos del manejador de cada petición como sigue. Debemos poner tantos argumentos del manejador como argumentos tendrá la petición HTTP. Veamos los ejemplos:

```
@GetMapping("/ejemplo")
public String manejador(@RequestParam(required=false,defaultValue = "vacío") String codigo) {
    return "El valor de tu código es " + codigo;
}

@GetMapping("/ejemplo2")
public String manejador(
    @RequestParam(required=false,defaultValue = "vacío") String codigo,
    @RequestParam(required=true,name="edad") int age) {
    return "El código es " + codigo + " y la edad es " + age;
}

@GetMapping("/ejemplo3")
public String manejador(@RequestMapping(@RequestParam
Map<String,String> todosEnUno) {...})
```

En el primer caso tenemos un atributo opcional, con valor por defecto "", de nombre **id**. Las llamadas podrían ser **http://localhost:8080/ejemplo** o bien **http://localhost:8080/ejemplo?codigo=1234**

En el segundo caso tenemos dos, pero el primero es opcional. El segundo parámetro difiere del nombre de la variable, con lo que las llamadas podrían ser: **http://localhost:8080/ejemplo2?edad=34** o **http://localhost:8080/ejemplo2?edad=34&id=codigo**. En los resultados de ejecución en la siguiente imagen, fíjate en qué ocurre cuando no existe el parámetro edad, que hemos definido como obligatorio.

En el tercer caso, recibiremos un mapa de parámetros (formado por parejas clave-valor), a los que tendremos que acceder a través de los métodos de recuperación de este.



Como finalización de lo visto anteriormente, los parámetros que podemos recuperar son los que vienen en la línea de la URL, pero esta práctica se desaconseja, debido a que se ve el valor de dichos parámetros. La alternativa es enviarlos en el cuerpo de la petición (**body**).

Lo que deberíamos hacer es simplemente cambiar **@RequestParam** por **@RequestBody**, sin pérdida de generalidad. Los parámetros que vienen como variables en la URL serán anotados con **@PathVariable**, como se verá en las próximas unidades.

Propiedad de McGraw-Hill®

1. Creación de un servicio REST

En este apartado vamos a crear un servicio REST con Spring Boot. Para ello, lo primero que tenemos que entender son las componentes que vamos a utilizar y cómo las enlazaremos entre ellas.

Hay que indicar también que la creación de un proyecto de un servicio REST implica muchos detalles y casuísticas que contemplar, que escapan a la unidad, incluso al módulo entero. Estamos viendo la punta de un iceberg. Se pretende con esto dotar al alumno de las nociones mínimas para conocer la estructura y poder integrarse con facilidad en un equipo de trabajo en este tipo de desarrollos.

NOTA

Mientras desarrollamos los contenidos del tema, se irán viendo capturas o bloques de código de un proyecto de ejemplo. El proyecto versa sobre la base de datos de Cine, con datos de películas y directores de estas. Tienes en el apartado de descargas el script de creación (**AD_U5_Cine.sql**), así como el proyecto de ejemplo en el archivo comprimido (**AD_U5_A3.zip**).

1.1. Creación del servicio REST

Un proyecto completo de Spring, cuando se aborda por primera vez, puede resultar apabullante, debido a la cantidad de clases que entran en juego. Por esta razón, vamos a explicar poco a poco cada una de estas clases y su separación en paquetes, así como los objetivos de estas.

A. CONFIGURACIÓN DEL PROYECTO

Cuando creamos un proyecto con Spring, tenemos que indicar los comportamientos que va a tener. En el asistente de creación (o bien mediante Spring Initializr) deberemos seleccionar los siguientes arquetipos: **DevTools**, **Lombok**, **MySQL** (o el SGBD preferido), **Spring Data JPA** y **Spring Web**. Con todos ellos, el asistente nos crea la estructura del proyecto y el fichero pom.xml, dado que **Spring** desarrolla por defecto con **Maven**. En principio, este fichero no hará falta modificarlo, aunque sí revisarlo. A continuación, deberemos configurar la parte de puerto de escucha del servidor, configuración de la base de datos y configuraciones de **Hibernate**.

B. MODELO

En primer lugar, tengamos como punto de partida la Unidad 3, donde estudiaste las herramientas ORM con **Hibernate**. **Spring** permite mapear nuestras bases de datos de manera muy cómoda con **Hibernate** y anotaciones JPA, por lo que la parte de acceso a la base de datos se realizará de la misma manera. Aquí, en el modelo, es donde estarán las clases conocidas como beans (clases con atributos, constructores, **getters** y **setters** y algunos métodos más).

NOTA

Estas clases podrán estar relacionadas entre ellas con las anotaciones de relaciones **@OneToOne**, **@OneToMany** y **@ManyToMany**, sin pérdida de generalidad.

Tienes disponible una interesante librería en el material complementario **Introducción a Lombok**, en la sección de descargas.

C. REPOSITORIO

El repositorio es una clase fundamental en el proceso de creación de aplicaciones con acceso a base de datos. Un repositorio contendrá los métodos que permiten acceder a las operaciones básicas, como cargar, guardar, borrar y actualizar.

Con **Spring** nos ahorraremos mucho trabajo, ya que simplemente definiremos una interfaz de trabajo, en la cual indicaremos el tipo de repositorio que queremos crear, la clase sobre la cual va a trabajar y el tipo de datos que funciona como identificador de dicha clase:

```
public interface classRepositorio extends xxxRepository<Class t, Type t>{
    // dejarlo vacío
}
```

Spring provee de la interfaz **Repository**, de quien hereda **CrudRepository**, que incluye la definición de las operaciones CRUD básicas. De esta última hereda **PagingAndSortingRepository**, que añade funciones de paginación y ordenación, y por último tenemos **JpaRepository**, que incluye operaciones específicas para JPA.

La importancia de la definición genérica del **Repositorio<Clase, Tipo>** es que todos los objetos que recuperará son de dicha clase, y el tipo indica el tipo de la clave principal de dicha clase. Siguiendo nuestro ejemplo, la definición del repositorio sería:

```
public interface DirectorRepo extends JpaRepository<Director, Long>{ }
```

Con esto, Spring ya nos permite acceder a la base de datos y realizar las operaciones básicas. Los métodos siguientes vienen implementados por defecto, y no necesitaremos implementarlos, solo definirlos:

Recuperar datos	<code>findAll()</code> , <code>findById(Id)</code> , <code>findById(Iterable<Id>)</code> : recupera una o todas las ocurrencias de un identificador o una colección de identificadores.
Borrar datos	<code>delete(Object)</code> , <code>deleteAll()</code> , <code>deleteById(Id)</code> , <code>deleteAllById(Iterable<Id>)</code> : borran según el objeto identificado o todos.
Contar y comprobar	<code>count()</code> , <code>existsById()</code>
Guardar	<code>save(Object)</code> , <code>save(Iterable<Object>)</code> : guarda el objeto u objetos.

Puede añadir funciones personalizadas más concretas definiendo un query, que se ejecutará directamente sin tener que programarlo (a menos que tenga argumentos).

D. SERVICIO

El servicio, o la declaración de una clase mediante el estereotipo **@Service**, indicará que dicha clase gestiona las operaciones de la lógica de negocio. Estos servicios son los que contendrán las llamadas al repositorio.

Para la creación de servicios (y después los controladores), podemos hacerlo de dos maneras, programando e implementando la clase directamente, o definiendo previamente una interfaz con los métodos que haya que implementar y posteriormente la clase que implemente dicha interfaz. Esta segunda manera sigue el patrón FACADE (fachada); podrás encontrarla en muchos ejemplos, pero queda fuera del alcance del curso.

```
// fachada del servicio
public interface DirectorService {
    public List<Director>findAllDirector();
    public Optional<Director> findDirectorById(Long id);
    public Director saveDirector(Director nuevoDirector);
    public String deleteDirector(Long id);
    public String updateDirector(Director nuevoDirector);
}

//implementación del servicio
@Service
public class DirectorServiceImpl implements DirectorService{

    @Autowired
    DirectorRepo directorRepositorio;

    @Override
    public List<Director>findAllDirector() {
        return directorRepositorio.findAll();
    }
}
```



```
@Override  
public Optional<Director> findDirectorById(Long id) {  
    return directorRepositorio.findById(id);  
}  
}
```

Como se puede comprobar, el servicio se encarga de invocar los métodos del repositorio, y de realizar algunas comprobaciones en caso de que sea necesario.

E. CONTROLADOR

El controlador será el encargado de responder a las peticiones del usuario con la aplicación. Incluirá los servicios, y en caso de crear una aplicación MVC podrá invocar motores de plantillas, como Thymeleaf, que estudiaremos en el siguiente apartado. Como hemos comentado aquí, lo implementaremos en una sola clase (sin seguir el patrón anterior).

El controlador invocará al servicio asociado a dicha petición y devolverá los datos obtenidos o la respuesta al mismo cliente. Debemos marcar la clase con el estereotipo **@Controller**. Para el caso de servicios REST, además debemos indicar que las devoluciones de los métodos de la clase sean serializadas a JSON, y eso lo conseguimos con **@ResponseBody**. Desde la versión 4 de Spring, las dos anotaciones se han fusionado en una, mediante **@RestController**. Dejaremos solo **@Controller** para proyectos en los que devolvamos una vista (HTML + CSS + JS). Así pues, el controlador quedaría:

```
@RestController  
public class DirectorController {  
    @Autowired  
    private DirectorService directorService;  
  
    @GetMapping(value = "/director")  
    public List<Director> getDirector() {  
        return directorService.findAllDirector();  
    }  
  
    @GetMapping(value = "/director/{id}")  
    public Optional<Director> getDirectorById(@PathVariable Long id) {  
        return directorService.findDirectorById(id);  
    }  
    // resto de métodos
```

IMPORTANTE

Aquí aparece la clase de envoltura **Optional<T>**. Es una clase que, básicamente, contiene un objeto de tipo T. Esta clase tiene dos métodos, **isPresent()** y **get()**, que nos sirven para saber si se ha devuelto algo (no es **null**) y para recuperar el objeto, respectivamente.

Podemos mejorar los métodos para el caso de que no existan resultados o haya ocurrido algún error, encapsulando los resultados en un **ResponseEntity<Resultado>**. Esta clase devuelve el resultado a la aplicación cliente, pero permite añadir un argumento, que será el código de estado HTTP. Este código puede capturarse en el cliente para la gestión de errores.

```
public ResponseEntity<Result> metodoDelControlador() {  
    // recogemos los datos del servicio/repositorio  
    if (!error) {  
        return new ResponseEntity<>(Resultados, HttpStatus.OK); // TODO BIEN
```

```
}  
  
    return new ResponseEntity<> (HttpStatus.NOT_FOUND); // ALGO FALLA  
  
}
```

IMPORTANTE

Tanto en el servicio como en el controlador, aparece la anotación **@Autowired**, que se encarga de inyectar el código necesario de aquella clase que implemente el servicio. Mediante esta anotación se crean los objetos necesarios sin necesidad de que lo realice el programador, de manera automática.

1.2. Consulta del servicio con Postman

En este apartado vamos a utilizar Postman como herramienta de prueba de servidores **REST**, ya que con el navegador solo podemos probar las peticiones **GET**. En el material complementario **Tutorial de Postman** encontrarás una breve guía de cómo crear las peticiones.

Propiedad de McGraw-Hill®

1. Generación de contenidos dinámicos con Spring

1.1. Creación de una aplicación web con Spring y MVC

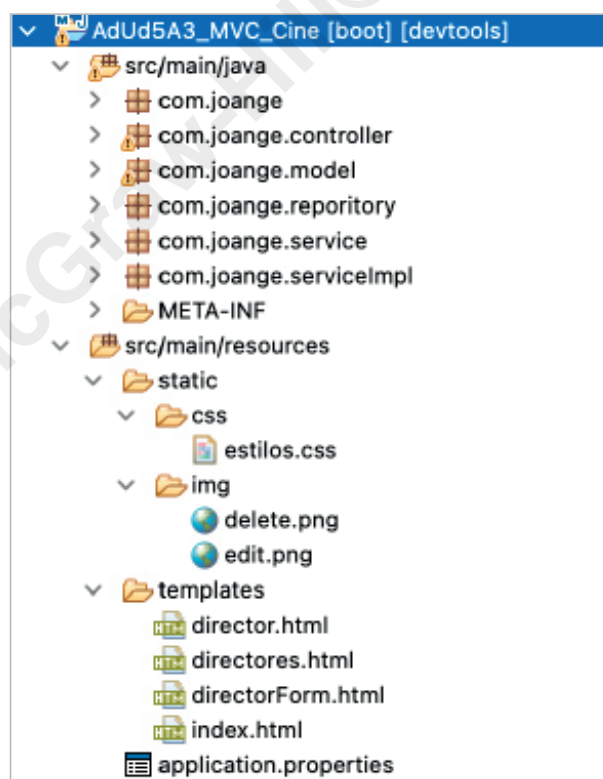
En la unidad anterior hemos construido un API REST con Spring. Como se vio, la aplicación permite gestionar las peticiones de las aplicaciones cliente (realizadas mediante el programa Postman, navegador o similar) y devolver los datos en formato JSON. Estos datos devueltos en crudo deberán ser presentados a los usuarios mediante su interfaz de usuario (GUI) en las aplicaciones cliente.

Lo que vamos a realizar ahora es la propia presentación de los resultados (que no de los datos) en una interfaz web, añadiendo la parte de la vista a nuestra aplicación. Además, esa vista se integrará con la vista para realizar también peticiones al controlador y a la lógica de negocio. El resultado será una aplicación web, en donde añadiremos la parte de la vista, y así cubrir la totalidad del patrón MVC, introducido en el primer apartado del tema (puedes revisar la sección 3 del primer apartado).

1.2. Estructura del proyecto

Si recordamos, el flujo de eventos que ocurren en un programa diseñado mediante MVC es el siguiente:

1. El usuario realiza la petición, atendida por el controlador.
2. El controlador decide qué servicio tiene que atender dicha petición, y mediante el modelo recuperará los datos de la base de datos.
3. El modelo devuelve los datos y los envía a la vista que ha indicado el controlador. Este envío de datos del controlador a la vista es el que presenta la gran novedad, dado que los datos se empaquetarán en un objeto especial, denominado **Model**, y se cargará la vista.
4. La vista para generar dicha presentación representará los datos desempaquetando el objeto **Model** recibido, y finalmente será devuelta al cliente.



Así pues, en nuestro proyecto, y partiendo del proyecto anterior, existen tres paquetes que no vamos a modificar para nada, que son el modelo, el repositorio y el servicio. Los datos van a seguir siendo los mismos, y las operaciones de consulta y recuperación desde la base de datos tampoco van a variar. La tarea la tenemos en el controlador y en la vista; sobre todo en esta última, que tenemos que crear desde cero.

A. LA VISTA

Las vistas de nuestra aplicación se encontrarán en la carpeta **src/main/resources**. En esta carpeta hasta ahora teníamos el fichero **application.properties**, pero añadiremos dos subcarpetas:

- **Recursos estáticos:** son contenidos que no van a variar, como su nombre indica. Nos referimos a imágenes, estilos (y, si fuera necesario, animaciones y sonidos). Dichos contenidos serán cargados, bien por la aplicación, bien por las plantillas.
- **Plantillas:** estarán en la carpeta templates. Dichas plantillas parecen documentos HTML, pero están programadas para generarlos de manera dinámica, después de procesarlos por el motor de plantillas. En esta unidad vamos a utilizar Thymeleaf como motor de plantillas. Podemos organizarlos también en subcarpetas, para tener organizadas las plantillas de cada apartado.

B. EL MOTOR DE PLANTILLAS

Como hemos comentado, Thymeleaf es un motor de plantillas que, a partir de un código HTML anotado y un modelo de datos, se encarga de generar un HTML completo, que será entregado al cliente como resultado final. Thymeleaf debe añadirse en la creación del proyecto como dependencia de este (con Spring Initializr o con el asistente de Spring Boot Studio).

C. NAVEGACIÓN

Ante el reto de diseñar una página web, hay muchos factores importantes. Uno es la navegabilidad entre pantallas. En el material complementario **AD_U05_MC_Navegación_Basica_Páginas_Web** puedes encontrar detalles de cómo navegar entre las distintas páginas (vistas) de un sitio completo.

1.3. Cambios en el controlador

Veamos los cambios necesarios en el controlador. Partiremos, para los ejemplos, del resultado del Apartado 3, donde hemos programado una API Rest que nos gestiona información de directores de cine. El primer cambio que hay que realizar está en la propia definición de la clase controlador:

Rest	MVC
<code>@RestController</code> <code>public class nombreControlador</code>	<code>@Controller</code> <code>public class nombreControlador</code>

En el API Rest, la anotación `@RestController` equivale a anotar la clase como `@Controller` y además indicar que todos los métodos tienen la anotación `@ResponseBody`. Esta anotación indicaba que el valor devuelto por el método se convierte en JSON y que dicha respuesta se devuelve al cliente que solicitó la petición. Esto solo tiene sentido si trabajamos con **REST**.

Al dejar la anotación con solo `@Controller`, los métodos deberán devolver ahora un **string**, en vez de cualquier tipo de dato. Dicho string será **el nombre de la vista que deberá generarse** y devolverse al cliente. Además, existirá un argumento en los métodos controladores, de tipo **Model**. Dicho objeto funciona como un **Map (atributo - valor)**, y deberán añadirse los datos necesarios para generar la vista de manera adecuada por el motor de plantillas. Estos datos se añadirán de manera sencilla con un método `model.addAttribute(etiqueta, valor)`. Dicha etiqueta será el mecanismo de acceso al valor desde la plantilla de la vista.

Aunque tengamos la anotación `@Controller`, esto no significa que no necesitemos implementar algún método que tiene que devolver un objeto JSON al cliente, por ejemplo, para validarse, anotarlo con `@ResponseBody`, y poder convivir métodos que devuelven vistas o JSON.

Supongamos que tenemos una clase **Producto** y que tenemos completamente implementada la parte del servicio, repositorio y modelo. Vemos el ejemplo de la petición de recuperar a todos los productos, comparándola con el controlador **REST**.

<pre>@GetMapping(value = "/producto") public List<Producto> getProducto() { return productoService.findAllProducto(); }</pre>
Ante la petición de GET /producto , se accede al servicio de recuperar todos los productos (<code>findAllProducto</code>), que, como vimos en el tema anterior, accede al repositorio y, en consecuencia, a la base de datos. La colección que retorna el servicio se devuelve a la petición. Como dicho método está dentro de la clase <code>@RestController</code> , esa lista de productos se transforma a JSON.
<pre>@GetMapping(value = "/producto") public String getProducto (Model model) { List<Producto> losProductos= productoService.findAllProducto(); model.addAttribute("productos", lossProductos); return "productosView"; }</pre>

Hemos de fijarnos en el cambio del prototipo de la función, que devuelve un **string** y tiene el **Model** como argumento. Se invoca de forma exactamente igual el servicio. Esto nos demuestra que el Servicio, el Modelo de datos y el Repositorio no han cambiado para nada. Añadimos la lista devuelta al **Model**, con la etiqueta «**productos**», y devolvemos un **string** «**productosView**». Al **Model** se le puede añadir toda la información que sea necesaria.

Este último valor devuelto dispara el motor de plantillas **Thymeleaf**, que busca en sus recursos un archivo «**productosView.html**» (la extensión no hace falta indicarla). A partir de aquí, Thymeleaf procesará la plantilla con el **Model** recibido, que contiene la colección de productos, y generará el HTML definitivo que será devuelto al cliente.

En el material complementario **AD_U05_MC_EjemploControladorVista** puedes visualizar un ejemplo de cómo se transfieren los datos desde el controlador a la vista.

1.4. La vista

Vamos a estudiar el último eslabón de la cadena, que es la vista. Como hemos comentado, la vista está compuesta por plantillas **Thymeleaf**, que combina HTML con código para procesar los datos recibidos del controlador. En el material complementario **AD_U05_MC_Tutorial** de Thymeleaf tienes una breve guía con los comandos más importantes de Thymeleaf, que usaremos en esta unidad, y dispones, en la sección de enlaces, del Enlace 5, en el que se referencia la página oficial, con mucha documentación e incluso un curso interactivo.

IMPORTANTE

Los contenidos de HTML necesarios para el diseño de estas plantillas los estudiaste en el módulo de primero de Lenguaje de Marcas. A partir de ahí, si necesitas material o te surgen dudas, se recomiendan los recursos que ofrece W3Schools, tanto para HTML como para hojas de estilo.

Supongamos, para nuestro ejemplo, que vamos a crear una vista para mostrar una serie de productos (descripción y precio) que hemos recuperado de la base de datos con un controlador (que suponemos implementado). También se facilitará el rol del usuario actual, y, si es un «**admin**», mostraremos el stock de dicho producto. Así pues, el controlador sería algo como:

```
@GetMapping(value = "/productos")
public String getProductos(Model model) {}

    List<Producto>losProductos= ...;
    model.addAttribute("productos",losProductos);
    model.addAttribute("admin",user.role=="admin");           // true o false
    return "productos";
}
```

La estructura de la plantilla es un simple HTML, pero debemos indicar que cargue las plantillas de sintaxis de Thymeleaf. En la cabecera podemos añadir todo lo necesario, y en el cuerpo, el contenido.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"><!-- carga de sintaxis
Thymeleaf-->
    <head>        </head>
    <body>
        <h1>Listado de productos</h1>
        <table>
            <thead>
                <tr>
                    <th>Descripcion</th>
                    <th>Precio</th>
                    <th th:if="${admin}">Disponible</th>
```

Descripcion	Precio	Disponible
Chair	\$25,00	18-feb-2013
Table	\$150,00	15-feb-2013
Armchair	\$85,00	20-feb-2013
Wardrobe	\$450,00	21-feb-2013
Kitchen table	\$49,00	15-feb-2013
Bookcase	\$80,00	17-feb-2013

```
</tr>
</thead>
<tbody>
  <tr th:each="producto : ${losProductos}">
    <td th:text="${producto.descripcion}">Descripción</td>
    <td th:text="${producto.precio}">Precio</td>
    <td th:if="${admin}">
      <span th:text="${producto.disponible}">Disponible</span></td>
    </td>
  </tr>
</tbody>
</table>
</body>
```

Conceptos claves:

- La cabecera de la tabla se generará siempre. La tercera columna, solo cuando el **if** se cumple, es decir, cuando **admin** tenga el valor a **true**. También se aplica dentro de los datos.
- En la creación de las filas de la tabla (**<tr>**) está la etiqueta **th:each**, que realiza un bucle para cada producto de la lista, lo que generará una fila por cada producto. El valor de cada objeto individual es el primer argumento del **each**, mientras que el segundo es la colección de productos.
- El contenido de una etiqueta se sustituye por el texto del contenido de las variables.

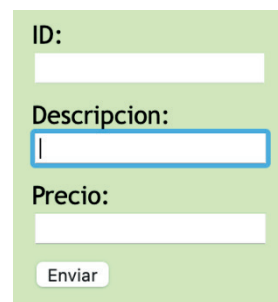
FORMULARIOS CON THYMELEAF

Los datos que se envían a las vistas pueden plasmarse como una página web, para solo visualizarse, o mediante un formulario, de forma que el usuario final puede editar los datos. Para este caso, la sintaxis cambia un poco. Imaginemos un ejemplo en que se solicitan los datos de un producto; permitimos al usuario que los edite, para posteriormente mandarlos a otra parte del controlador para guardarse:

```
<form th:action="@{/productos/save}" th:object="${producto}" method="POST">
  <div th:if="*{idProd}"> ID: <input type="text" th:field="*{idProd}"
  readonly></div>
  <div> Descripción: <input type="text" th:field="*{descripcion}"></div>
  <div> Precio: <input type="text" th:field="*{precio}"></div>
  <input type="submit" th:value="${nuevo}? 'Guardar' : 'Modificar' "/>
</form>
```

Este formulario deberá recibir dos variables: **nuevo**, un valor booleano para indicar si estamos ante un alta o una modificación, y el objeto **Producto**, que pasaremos, o bien con información para modificarlo, o bien con un **new Producto()** para darlo de alta.

- Si el producto es nuevo, hay que tener en cuenta que no poseerá un identificador, por lo que no existirá el cuadro para poner su identificador, ya que se genera automáticamente. Si ya existe, se muestra, pero es **readonly**, ya que el identificador no lo podemos modificar.
- Fíjate en que en el formulario se declara un **th:object="\${producto}"**, y todos los campos están identificados por el nombre de sus atributos (**idProd**, **descripcion** y **precio**) mediante **th:field**. Esto es importante para poder recoger los valores al enviar los datos.
- El valor de la booleana **nuevo** permite generar la etiqueta el botón de enviar como Guardar/Modificar.



Finalmente, el botón de enviar genera una petición POST a `/productos/save`, que vemos a continuación:

```
@PostMapping(value = "/productos/save")
public String updateDirector(Model model,
    @ModelAttribute("producto") Producto elProducto) {
    productoService.saveProducto(elProducto);
    return "redirect:/productos";
}
```

La novedad está en la anotación `@ModelAttribute("producto") Producto producto`, donde se indica que tenemos un elemento en la plantilla que se llama `producto`, y que corresponde a un objeto de tipo `Producto`, y accedemos a él dentro del método llamándolo `elProducto`. En la función, simplemente guardamos el producto a través del servicio correspondiente y se redirecciona al índice de la página.

IMPORTANTE

Una vez concluido el desarrollo de la aplicación, es interesante conocer cómo podemos desplegarla. En el documento adjunto `AD_U05_MC_Ampliación_Despliegue`, en la sección de descargas, se desarrollan varias opciones distintas.

Propiedad de McGraw-Hill



Unidad 6.

**Bases de datos
documentales.**

MongoDB y BBDD XML

1. Bases de datos NoSQL

1.1. El movimiento NoSQL

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

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

Con la llegada de la web 2.0, el crecimiento de datos en Internet creció de forma exponencial, de la mano, principalmente, de las redes sociales y del contenido multimedia: Facebook, Twitter, YouTube, etcétera, plataformas donde son los propios usuarios quienes aportan la mayor parte de contenido, lo que provoca un rápido crecimiento de los datos almacenados y un serio problema para los sistemas que no estaban preparados para ello. Con la web 3.0, 4.0 y el auge de la inteligencia artificial, Internet se ha convertido en una gran base de datos, ya no solo para proporcionar contenido a la web, sino a cualquier tipo de aplicación.

Las bases de datos NoSQL permiten asumir estos escenarios donde las bases de datos resultan problemáticas, debido principalmente a la falta de escalabilidad (posibilidad de crecimiento) y bajo rendimiento de estas, cuando miles de usuarios acceden de forma simultánea a la información.

Todo esto no significa que las bases de datos tradicionales se vayan a reemplazar por las NoSQL, sino que estamos en un panorama donde no podemos encontrar una única tecnología que sea apropiada para todos los escenarios. Es por ello por lo que estamos hoy en día ante un buen número de soluciones específicas que mejoran ciertos problemas; todas diferentes, pero englobadas dentro del movimiento NoSQL.

1.2. Tipos de bases de datos NoSQL

Dentro del panorama NoSQL podemos encontrar diferentes tipos de bases de datos, según su forma de almacenar los datos. Entre ellas, podemos destacar las siguientes.

A. BASES DE DATOS CLAVE-VALOR

Se trata de un modelo de base de datos bastante sencillo y popular, donde cada elemento se identifica por una clave única, siguiendo el modelo de las tablas *hash*, de modo que el dato se recupera de forma muy rápida. Generalmente, los objetos se almacenan como objetos binarios (BLOB).

Algunas bases de datos de este tipo son **Cassandra (Apache)**, **Bigtable (Google)** o **Dynamo (Amazon)**.

B. BASES DE DATOS DOCUMENTALES

Este modelo almacena la información en forma de documentos, generalmente XML o JSON, y se utiliza una clave única para cada registro, por lo que se permiten búsquedas por clave-valor. La diferencia respecto a las bases de datos clave-valor anteriores es que aquí el valor es el propio documento, no un dato binario.

Como veremos más adelante, son muy versátiles, de modo que no necesitamos tan siquiera tener una estructura común a los documentos que guardamos.

El máximo exponente de este tipo de bases de datos es MongoDB.

C. BASES DE DATOS EN GRAFO

Un **grafo** es un conjunto de vértices o nodos unidos por aristas, que nos permiten representar relaciones entre ellos.

Las bases de datos en grafo pretenden seguir este modelo, de manera que la información se representa como nodos en un grafo, y las relaciones entre ellos se representan mediante aristas. De este modo, aprovechando la teoría de grafos, podemos recorrer la información de forma óptima.

Algunos ejemplos de este tipo de bases de datos son **Amazon Neptune**, **JanusGraph (Apache)**, **SQL Server (Microsoft)** o **Neo4j**.

1.3. Bases de datos documentales con MongoDB

Vamos a centrarnos en el estudio de uno de los sistemas de bases de datos NoSQL más utilizados hoy en día: MongoDB.

MongoDB es una base de datos orientada a documentos, basada en el almacenamiento de sus estructuras de datos en documentos de tipo JSON con un esquema dinámico. Aunque empezó siendo desarrollado por la empresa **10gen**, hoy en día es un proyecto de código abierto, con una gran comunidad de usuarios.

Un servidor MongoDB puede contener varias bases de datos, y cada una de ellas compuesta por un conjunto de colecciones, que podríamos equiparar a las tablas de una BD relacional. Cada colección almacena un conjunto de documentos JSON, formado por atributos clave-valor, que vendrían a ser los registros de una base de datos relacional.

A grandes rasgos, podríamos establecer las siguientes equiparaciones:

Modelo relacional	MongoDB
BD relacional	BD orientada a documentos
Tabla	Colección
Registro/fila	Documento JSON
Atributos/columnas	Claves del documento JSON

Veamos, a modo de ejemplo, una colección **Películas** con dos documentos:

```
[{
  _id: 1,
  titulo: "La Amenaza Fantasma",
  anyo: 1999,
  director: {
    nombre: "George",
    apellidos: "Lucas",
    anyo_nacimiento: 1944
  }
},
{
  _id: 2,
  titol: "El Ataque de los Clones",
  year: 2002,
  director: {
    nombre: "George",
    apellidos: "Lucas",
    anyo_nacimiento: 1944
  }
}]
```

Como podemos ver, cada documento de tipo película posee sus propios atributos, y estos no tienen por qué coincidir entre dos documentos. Otra característica interesante es que, como podemos observar, disponemos de toda la información sobre el director dentro del propio documento.

CARACTERÍSTICAS DE MONGODB

Algunas de las principales características de MongoDB son:

- Es una base de datos **orientada a documentos**: los diferentes documentos u objetos de la base de datos se pueden mapear fácilmente a los objetos de las aplicaciones.
- Como en todo documento JSON, el valor asociado a una clave puede ser también un documento JSON, lo que aporta mucha **flexibilidad**. Esto sería el equivalente a las tablas embebidas en las bases de datos objeto-relacionales. Este hecho de poder tener documentos **embebidos** en otros facilita las consultas, ya que no se necesita hacer **Joins**.
- Los **esquemas** de los documentos son **dinámicos**. Esto significa que dos documentos no tienen por qué seguir el mismo esquema (un documento puede tener campos diferentes a otros documentos), con lo que el polimorfismo resulta más sencillo.
- Proporciona un **alto rendimiento**, ya que los documentos embebidos posibilitan las lecturas y escrituras más rápidas, los índices pueden incluir claves de documentos embebidos y vectores, y, además, se pueden realizar escrituras en **streaming** cuando no sea necesario confirmar el final de la escritura.
- Proporcionan **alta disponibilidad**, con servidores replicados con gestión automática de errores del sistema.
- Es fácilmente **escalable**, con **sharding** (fragmentación) automático, que distribuye los datos de una colección entre varias máquinas, y las lecturas se pueden distribuir por los servidores replicados.

Respecto de los SGBD, podemos destacar las siguientes diferencias:

- MongoDB ofrece una **mayor flexibilidad**, al no estar sujeto a la estricta definición de un esquema. Por ejemplo, cuando en un SGBDR necesitamos un campo nuevo para una fila concreta, hay que incorporar el campo en todas las filas, mientras que en MongoDB, cuando necesitamos añadir un campo a un documento, solo se lo tenemos que añadir al propio documento, sin que los demás se vean afectados.
- **No se utiliza el lenguaje SQL** para hacer consultas, sino que se pasa como parámetro un JSON que describe lo que se quiere recuperar.
- MongoDB **no admite los JOIN**, sino que ofrece tipos de datos multidimensionales, como vectores u otros documentos dentro del propio documento. Por ejemplo, un documento puede tener un vector con todos los objetos relacionados de otra colección.
- No es necesario realizar transacciones cuando las colecciones sean en un mismo documento, pues siempre se realiza de forma atómica. En el caso que se use distintos documentos, **MongoDB** (a partir de la versión 4.2) sí ofrece herramientas para poder realizar transacciones.

1.4. El ecosistema de MongoDB

MongoDB abarca un gran abanico de posibilidades, desde servidores para bases de datos locales hasta bases de datos en la nube. En su **web**, podemos descubrir los diferentes productos y servicios que se ofrecen, entre los que encontramos:

- **Servidor de BD MongoDB**, con sus dos versiones, la Community, versión comunitaria y gratuita, y la Enterprise, su versión comercial orientada al mundo empresarial y con características adicionales que mejoran el rendimiento y el soporte. Además del servidor en sí, en la web se ofrecen también el servidor preparado para su uso en contenedores, mediante operadores Kubernetes. El servidor está disponible en varias plataformas: Linux, Solaris, MacOS X y Windows.
- **MongoDB Atlas**, la plataforma MongoDB en la nube (DBaaS o *database as a service*), que permite su despliegue en servicios como AWS, Azure o Google Cloud.
- **Realm**, un servicio de datos pensado para aplicaciones móviles y web, y que incluye, además de BD en la nube, varios servicios de *backend* completamente administrados.

1. Trabajar con MongoDB: operaciones básicas desde la shell

1.1. Colecciones y documentos

Como ya sabemos, la unidad de información con que trabaja MongoDB es el **documento**, que sería el equivalente a un registro en un modelo relacional. Se trata de documentos JSON, formados por pares **clave-valor**, y que representan la información de forma bastante intuitiva. Los servidores MongoDB, por su parte, almacenarán estos datos en formato BSON (Binary JSON), un formato binario de serialización.

Respecto a los documentos JSON para MongoDB, hay que tener en cuenta algunos aspectos:

Respecto a las claves	<ul style="list-style-type: none"> – No pueden ser nulas. – Pueden estar formadas por cualquier carácter UTF-8, salvo los caracteres «.» o «\$». – Son <i>case-sensitive</i>. – Deben ser únicas dentro de un mismo documento.
Respecto a sus valores	<ul style="list-style-type: none"> – Pueden ser de cualquier tipo permitido.
Respecto al documento	<ul style="list-style-type: none"> – Debe poseer un campo _id, con un valor único, que actuará como identificador del documento. Si no especificamos esta clave, MongoDB la generará automáticamente, con un objeto de tipo ObjectId.

Si los documentos son el equivalente a los registros, las **colecciones** son el equivalente a las tablas, con la diferencia de que las colecciones poseen un esquema dinámico, con lo que documentos de la misma colección pueden presentar claves o tipos de datos diferentes entre ellos.

Los nombres de las colecciones estarán sujetas a las siguientes restricciones:

- No pueden ser la cadena vacía (""), ni el carácter nulo, ni contener el símbolo \$.

Podemos utilizar el punto (.) en los nombres de colecciones para añadir prefijos a esta, pero no se pueden crear colecciones con el prefijo **system.**, ya que este se usa para colecciones internas del sistema. Por ejemplo, **db.system.prueba** no sería válido, pero **db.systema.prueba**, sí.

1.2. Operaciones básicas con MongoDB

En la siguiente tabla, vamos a ver algunas de las operaciones que podemos realizar sobre MongoDB:

Operación	Significado	Ejemplo
insertOne (documento) insertMany (documentos)	Añade un documento a la colección.	db.coleccion.insertOne({ a:1 })
find(criterio)	Obtiene todos los documentos de una colección que coinciden con el patrón indicado.	db.coleccion.find({a:1});
findOne (Criterio)	Obtiene un elemento de la colección coincidente con el patrón.	db.coleccion.findOne();
updateOne (Criterio, Operacion, [opciones]) updateMany (Criterio, Operacion, [opciones])	Actualiza un documento de la colección (o varios, en el caso de updateMany). Requiere dos parámetros: el criterio de busca del documento que se va a actualizar y la operación de actualización. Admite un tercer parámetro opcional para las opciones.	db.coleccion.updateOne({a:1}, {\$set: {a:2}})
deleteOne (Criterio) deleteMany (Criterio)	Borra los documentos de una colección que cumplen el criterio.	db.coleccion.deleteOne({a:1})

En los siguientes apartados profundizaremos en las diferentes operaciones.

1.3. Tipos de datos

Los tipos de datos con que trabaja MongoDB son similares a los que podemos encontrar en JavaScript. Aunque dispones de una descripción más detallada en la documentación adicional, vamos a examinar algunos de estos tipos:

A. TIPOS BÁSICOS

MongoDB admite los **tipos básicos** que se describen en la siguiente tabla:

Tipo	Descripción
null	Representa tanto el valor nulo como un campo que no existe.
boolean	Permite los valores true y false .
number	Representan valores numéricos en coma flotante. Si queremos utilizar tipos enteros o enteros largos, hay que utilizar las clases propias: NumberInt (32 bits) o NumberLong (64 bits).
String	Representan cualquier cadena de texto UTF-8 válida.
Date	Representa fechas, expresadas en milisegundos.
array	Listas de valores que se representan como vectores.
Documentos incrustados	Los documentos pueden tener otros documentos incrustados en ellos .
ObjectId	Se trata del tipo por defecto para los campos _id , y está diseñado para poder generar de forma sencilla valores únicos de forma global.

Vamos a ver algunas peculiaridades de interés sobre algunos de estos tipos.

B. EL TIPO DATE

Mongo utiliza el **tipo Date** de JavaScript. Cuando generamos un nuevo objeto de tipo **Date**, hay que utilizar el operador **New**, puesto que en caso contrario obtendríamos una representación de la fecha en forma de **string**.

Por ejemplo, si definimos las variables **a** y **b** del siguiente modo:

```
test> let a=Date()  
test> let b=new Date()
```

Podemos ver que los resultados son bastante diferentes:

```
test> a  
Sun May 08 2022 06:46:01 GMT+0200 (hora de verano de Europa central)  
test> typeof(a)  
string  
  
test> b  
ISODate("2022-05-08T04:46:09.371Z")  
  
test> typeof(b)  
object
```

C. VECTORES

Los vectores pueden utilizarse tanto para representar colecciones ordenadas, como listas o colas, o bien colecciones no ordenadas, como los conjuntos. Al igual que en JavaScript, y a diferencia de otros lenguajes, como Java, cada elemento del vector puede tener un tipo de dato diferente, incluso otros objetos de tipo vector.

Veamos algunos ejemplos sobre vectores en JavaScript y por tanto en MongoDB:

- Creación de un vector:

```
test> let v={objetos: ["casa", 10, {texto: "hola"}, false] }
```

- Consulta del vector o sus componentes:

```
test> v
{ objetos: [ 'casa', 10, { texto: 'hola' }, false ] }
test> v.objetos
[ 'casa', 10, { texto: 'hola' }, false ]
test> v.objetos[1]
10
test> v.objetos[2]
{ texto: 'hola' }
```

- Modificación de valores:

```
test> v.objetos[3]=!v.objetos[3]
true
test> v
{ objetos: [ 'casa', 10, { texto: 'hola' }, true ] }
```

D. DOCUMENTOS INCRUSTADOS O EMBEBIDOS

Un par clave-valor en un documento puede tener como valor otro documento. Esto se conoce como documentos incrustados (embed), y no sería más que utilizar un objeto JSON dentro de otro. Por ejemplo:

```
> let peli={
  titulo: "Rogue One. A Star Wars Story.",
  anyo: 2016,
  director: {
    nombre: "Gareth",
    apellidos: "Edwards",
    anyo_nacimiento: 1975,
    nacionalidad: "británica"
  }
}
```

Como vemos, el propio documento lleva información sobre la película y sobre su director. En un modelo relacional, normalmente tendríamos dos tablas relacionadas entre ellas. En este caso, es posible que, si deseamos mantener información específica sobre los directores, acabemos teniendo información redundante.

E. SOBRE LOS OBJECTIDS

La clase **ObjectId** utiliza 12 bytes, organizados de la siguiente forma:

0	1	2	3	4	5	6	7	8	9	10	11
Timestamp				Machine			PID		Increment		

- **Timestamp (bytes 0-3):** el *timestamp* en segundos desde el 1 de enero de 1970.
- **Machine (bytes 4-6):** identificador único de la máquina; generalmente, un *hash* de su *hostname*.
- **PID (bytes 7-8):** identificador del proceso que genera el **ObjectId**, para garantizar la unicidad dentro de la misma máquina.
- **Incremento (bytes 9-11):** valor autoincremental, para garantizar la unicidad en el mismo segundo, máquina y proceso.

Como vemos, se trata de un mecanismo más robusto que un campo autoincremental como en MySQL. Esto se corresponde a la naturaleza distribuida de MongoDB, de forma que se puedan generar los objetos en un entorno con múltiples hosts.

1.4. Añadir información a las colecciones

La manera natural de añadir elementos a la base de datos es mediante los diferentes métodos **insert**, disponibles en todas las colecciones.

A. INSERTONE()

Permite insertar un único documento en la colección. Por ejemplo, para insertar el documento **pelí** definido anteriormente, podríamos hacer:

```
test> db.misPelis.insertOne(peli)
{
  acknowledged: true,
  insertedId: ObjectId("6277510ab54867b80b742ddf")
}
```

Como vemos, la respuesta obtenida es un documento JSON que contiene un valor booleano que indica si la operación se realizó con éxito, y un **ObjectId**, con el ID asignado automáticamente.

Algunas consideraciones:

- Si la colección a la que añadimos un documento no existe, esta se crea de forma automática.
- Respecto al campo **_id**, como vemos, este se generó automáticamente. No obstante, podemos indicar nosotros dicho identificador, sin que este sea además de tipo **ObjectId**; la única restricción es que esta sea única, para evitar duplicados.
- Hay que destacar que no hemos utilizado ningún esquema para la colección, ya que cada documento que insertemos puede tener un esquema diferente.

B. INSERTMANY()

Permite añadir varios documentos en una colección. Para ello, le proporcionaremos un vector de **documentos** por añadir a la colección.

Por ejemplo, si definimos tres documentos nuevos del siguiente modo:

```
test> let peli2={titulo: "Star Wars. A new Hope", anyo: 1977};
test> let peli3={titulo: "Empire Strikes Back", anyo: 1981};
test> let peli4={titulo: "Return of the Jedi", anyo: 1984};
```

Podemos insertarlos con:

```
test> db.misPelis.insertMany([peli2, peli3, peli4])
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId("627759a5b54867b80b742de0"),
    '1': ObjectId("627759a5b54867b80b742de1"),
    '2': ObjectId("627759a5b54867b80b742de2")
  }
}
```

Hay que tener en cuenta que, cuando se produce un error en la inserción de un elemento del vector (por ejemplo, si hemos indicado una clave duplicada), finaliza la ejecución del proceso, de modo que ni el documento que ha producido el error ni los siguientes en el vector se insertarán a la colección.

1.5. Eliminar información de MongoDB

Para eliminar documentos de una colección usaremos las órdenes **deleteOne()**, **deleteMany()** o **findOneAndDelete()**, proporcionándoles como parámetro un JSON, con la condición que queremos que cumpla el documento o documentos que se quieren eliminar.

La orden **deleteOne** eliminará únicamente el primer elemento que coincida con el criterio, de modo que si lo que deseamos es eliminar un documento concreto, deberemos utilizar criterios que se correspondan con índices únicos, como, por ejemplo, el **_id**.

La orden **deleteMany** eliminará todos los documentos que coincidan con el criterio.

Tanto **deleteOne** como **deleteMany** devuelven un documento con un booleano, indicando si se ha realizado la operación, así como el número de elementos eliminados (**deletedCount**).

Por su parte, **findOneAndDelete** también elimina un documento, de acuerdo con criterios de selección y ordenación, pero devolviendo además el documento que se ha eliminado.

Por ejemplo, creamos una colección con varios elementos:

```
db.pruebas.insertMany([{"x":1}, {"x":2}, {"x":3}, {"x":4}, {"x":5}, {"x":6}, {"x":7}]);
```

Ahora podemos:

- Borrar un elemento:

```
test> db.pruebas.deleteOne({})
{ acknowledged: true, deletedCount: 1 }
```

Con lo que habrá eliminado el primer registro.

- Eliminar varios elementos que cumplan un criterio; por ejemplo, que el valor de **x** es mayor que 3:

```
test> db.pruebas.deleteMany({"x":{"$gt":3}})
{ acknowledged: true, deletedCount: 4 }
```

Como vemos, se han eliminado cuatro documentos. El criterio **{\$gt:3}** lo veremos posteriormente, pero sirve para seleccionar aquellos registros cuyo valor en **x** sea mayor que 3.

- Eliminar un documento y devolverlo:

```
test> db.pruebas.findOneAndDelete({"x":2})
{ _id: ObjectId("6277687fb54867b80b742deb"), x: 2 }
```

Por otro lado, también podemos eliminar una colección completa mediante la orden **drop**. Hay que tener especial cuidado con esta orden, ya que elimina todos los metadatos asociados y puede resultar peligrosa.

```
test> db.pruebas.drop()
```

1.6. Actualización de documentos

Para la actualización de documentos, podemos optar, bien por actualizaciones de remplazo, mediante el método `replaceOne()`, o bien por realizar modificaciones sobre los documentos existentes, mediante los métodos `updateOne()`, `updateMany()` y `findOneAndUpdate()`. Estos métodos recibirán dos argumentos: el primero será el criterio que deberán cumplir los documentos que se quieren actualizar, y el segundo será un documento, bien con el nuevo documento, bien con las actualizaciones que se quieren aplicar.

A. ACTUALIZACIÓN DE REEMPLAZO (*REPLACE*)

La operación de reemplazo, como su nombre indica, reemplaza todo un documento que cumpla el criterio de actualización por otro documento nuevo.

Por ejemplo, creamos una nueva colección *agenda*, para guardar contactos, con información sobre teléfonos:

```
test> db.agenda.insertOne({nombre: "Jose",  
  telefonos: [{trabajo: "55512345", casa: "555111222"}]}  
)  
{  
  acknowledged: true,  
  insertedId: ObjectId("627783dbb54867b80b742df8")  
}
```

Como vemos, este método nos devuelve el `_id` del objeto, mediante el cual vamos a poder identificar inequívocamente este documento. Así pues, podríamos reemplazar este documento por otro mediante:

```
test> db.agenda.replaceOne({"_id": ObjectId("62778439b54867b80b742df9")},  
  {nombre: 'Jose', correos: [{trabajo: "jose@empresa.com"}, {personal:  
  "jose@proveedor.com"}]} )  
  
{  
  acknowledged: true,  
  insertedId: null,  
  matchedCount: 1,  
  modifiedCount: 1,  
  upsertedCount: 0  
}
```

Como podemos ver, se trata de reemplazar el documento completo, con lo que podemos incluso modificar la estructura de este.

B. ACTUALIZACIONES

Como hemos anticipado, las modificaciones se realizan mediante los métodos `updateOne()`, `updateMany()` y `findOneAndUpdate()`. De forma similar a las operaciones de borrado, el método `updateOne()` modificará solamente el primer documento que coincida con el criterio dado, y el método `updateMany()`, todos los que cumplan el criterio. Por su parte, el método `findOneAndUpdate()` modifica el documento y devuelve, por defecto, el documento original, aunque esto es configurable mediante opciones. Puedes consultar los enlaces recomendados para obtener más información acerca de este método.

C. MODIFICADORES

Los modificadores son claves especiales que nos permiten especificar operaciones de actualización más complejas. Normalmente, no necesitaremos reemplazar todo el documento, como en el caso anterior, sino añadir o modificar campos concretos. En la siguiente tabla vamos a ver los diferentes modificadores que tenemos a nuestra disposición:

Modificador	Descripción	Ejemplo de sintaxis (indistintamente con <code>updateOne</code> o <code>updateMany</code>)
<code>\$set</code>	Asigna valor a un campo al documento. Si este no existe, lo crea.	<code>db.coleccion.updateOne({criterio}, {\$set: {campo:valor}});</code>
<code>\$unset</code>	Elimina un campo de uno o varios documentos. Dado que debemos introducir un par clave-valor, añadiremos un booleano como valor.	<code>db.coleccion.updateMany({criterio}, {\$unset: {campo:true}});</code>
<code>\$inc</code>	Incrementa o decrementa el valor numérico de una clave (no se refiere al identificador), creando una nueva, si no existe.	<code>db.coleccion.updateOne({criterio}, {\$inc: {campo:incremento}});</code>
<code>\$push</code>	Añade elementos a un vector. Si el vector no existe, lo crea, con los elementos que indicamos en el <i>push</i> , mientras que, si ya existe, los añade al final de este.	<code>db.coleccion.update({criterio}, {\$push: {nombre_array:{lista_de_valores}}});</code>
<code>\$pull</code>	Elimina elementos de un vector de acuerdo con algún criterio.	<code>db.coleccion.update({criterio}, {\$pull: {vector:elemento}});</code>
<code>\$pop</code>	Elimina elementos de un vector tratado como una pila o cola, es decir, eliminando el primer [-1] o el último [1] elemento.	<code>db.coleccion.update({criterio}, {\$pop: {vector: [-1 1]}});</code>

D. UPSERTS

Cuando no se encuentra ningún documento que coincida con los criterios de una actualización, como es de esperar, no se produce ninguna modificación en la colección.

En cambio, en ocasiones, podemos desear que, si un documento con ciertos criterios no existe cuando queremos modificarlo, este se cree. Esto se consigue mediante **updates** especiales, denominados **upserts**. Con esta operación, nos ahorramos buscar primero en la colección, para saber si tenemos que realizar una operación de inserción (si no existe) o de modificación (si existe).

Para realizar un **upsert**, utilizaremos el tercer argumento de los **updates**, que consiste en un documento con diferentes opciones en formato clave-valor, añadiendo la clave **upsert** a valor **true**.

```
db.coleccion.updateOne({criterio}, {modificación}, {upsert:true});
```

1. Consultas con MongoDB

1.1. El comando *find()*

La orden **find** nos permite recuperar los documentos de una colección que coincidan o hagan match con un criterio especificado como un documento JSON. Su sintaxis básica es la siguiente:

```
db.coleccion.find({criterio_en_formato_JSON});
```

Debemos tener en cuenta aspectos como que los tipos de datos que utilicemos sí son una cuestión importante, ya que no es lo mismo el documento `{edad:20}` que `{edad:"20"}`.

Por otro lado, hay que considerar también que el documento vacío `{}` casa con todos los documentos, de forma que la consulta

```
db.coleccion.find({});
```

devolvería todos los objetos de la colección.

A. ESPECIFICAR LAS CLAVES QUE HAY QUE RECUPERAR

El comando **find** proporciona los documentos completos que coincidan con el criterio de selección. Si no deseamos obtener todas las claves, podemos especificar qué claves deseamos consultar, incorporándolas en un segundo parámetro:

```
db.coleccion.find({documento_de_consulta}, {clave_1:1, clave_2:1});
```

Como vemos, este segundo parámetro también se expresa en formato JSON y está compuesto por dos claves (**clave_1** y **clave_2**), ambas con valor **1**. Este valor numérico se interpreta también con el valor **true**. Es decir, especificamos aquí cuáles son los campos que deseamos mostrar. En caso de que deseemos mostrar todos los campos y ocultar algunos, utilizaríamos la misma sintaxis, pero empleando ahora un **0** para aquellos campos que queramos ocultar.

B. OPERACIONES DE COMPARACIÓN

MongoDB nos permite realizar comparaciones con los datos de tipo numérico, utilizando siempre el formato de documento JSON:

```
db.coleccion.find({clave: {$operador:valor} });
```

Los operadores de comparación que podemos utilizar en MongoDB son:

Operador de comparación	Significado
\$lt	Menor que
\$lte	Menor o igual que
\$gt	Mayor que
\$gte	Mayor o igual que

C. LA OPERACIÓN OR

Si deseamos realizar un filtrado o consulta donde se cumplan varias condiciones (una operación **AND**), no tendremos más que separar estas por comas en el mismo documento JSON que utilicemos como criterio. En cambio, si lo que deseamos es realizar una operación **OR**, deberemos utilizar algún operador especial.

D. OPERADORES \$IN Y \$NIN

Un caso especial de **OR** es cuando queremos comprobar si un campo se encuentra dentro de un conjunto de valores concreto. Es decir, si es **uno** u **otro** valor. Para ello, utilizamos el operador **\$in**, de la siguiente forma:

```
db.coleccion.find({clave: {$in: [vector_de_valores]}})
```

Del mismo modo, existe el operador **\$nin** (**not in**), que obtiene los documentos, donde el valor especificado no se encuentra en la lista. Debemos tener en cuenta que en este último caso también se mostrarán aquellos documentos que tengan para la clave el valor a **null**.

E. EL OPERADOR \$OR

Cuando la operación **OR** la queremos realizar sobre diferentes campos del documento, usaremos el operador **\$OR**, al que le pasamos un vector de posibles condiciones, de la siguiente forma:

```
db.coleccion.find({$or:[condicion1, condicion2,...]}))
```

F. EL OPERADOR \$NOT

El operador **\$NOT** es un operador **metacondicional**, es decir, se aplica siempre sobre otro criterio, invirtiendo su valor de certeza.

Su sintaxis sería:

```
db.coleccion.find({clave:{$not: {criterio}}}).pretty();
```

G. EL OPERADOR \$EXISTS

Recordemos que, en MongoDB, los documentos no poseen una estructura o esquema común, por lo que es posible que existan claves definidas solamente en algunos de ellos. El operador **\$exists** se utiliza para comprobar la existencia o no de determinada clave.

La sintaxis para utilizar sería:

```
db.coleccion.find({clave:{ $exists: true|false }})
```

Con ello, obtenemos los documentos para los cuales la clave existe o no, según hayamos indicado **true** o **false** en la consulta.

1.2. Consideraciones sobre los tipos de datos en las consultas

Los tipos de datos en MongoDB pueden tener algunos comportamientos especiales. Vamos a ver algunos casos, para saber qué hacer en determinadas situaciones.

A. VALORES NULOS

El valor **null** casa con las siguientes situaciones:

- Cuando el valor de la clave es **null**, o bien
- Cuando la clave no existe en el documento (en este caso, se suele decir que no tiene informado el campo).

B. EXPRESIONES REGULARES Y CADENAS DE CARACTERES

Cuando aplicamos un filtrado de documentos por un campo de texto, es posible que no sepamos exactamente el valor del campo por el que deseamos filtrar. Las expresiones regulares ofrecen un mecanismo muy potente para buscar coincidencias en cadenas de caracteres.

MongoDB nos permite utilizar estas expresiones de varios modos, bien utilizando expresiones regulares de JavaScript o bien mediante el operador **\$regex**, que utiliza las expresiones regulares compatibles con Perl (PCRE).

C. EXPRESIONES REGULARES DE JAVASCRIPT

Las expresiones regulares de JavaScript se expresan mediante la siguiente sintaxis:

```
{ clave: /patrón/<opciones> }
```

Como vemos, utilizamos un patrón de forma similar a una cadena de texto, pero usando la barra / como delimitador, en lugar de las comillas.

D. EXPRESIONES REGULARES CON \$REGEX

Por su parte, si usamos el operador **\$regex**, podemos usar las siguientes sintaxis:

```
{ clave: { $regex: /patrón/, $options: '<opciones>' } }
{ clave: { $regex: 'patrón', $options: '<opciones>' } }
{ clave: { $regex: /patrón/<opciones> } }
```

E. OPCIONES PARA EXPRESIONES REGULARES

Opción	Descripción	Ejemplos
i	Las coincidencias son <i>case-insensitive</i> .	<code>{nombre:/juan/i}</code> <code>{nombre: { \$regex: 'juan', \$options: 'i' } }</code>
m	Permite incluir caracteres como <code>^</code> o <code>\$</code> , para hacer <i>matching</i> al principio o al final, en cadenas con múltiples líneas.	<code>{nombre:/^Juan/m}</code> <code>{nombre: { \$regex: 'Juan', \$options: 'm' } }</code>
x	Ignora espacios en blanco en el patrón de \$regex , siempre que no se hayan escapado o incluido en una clase de tipo carácter.	<code>{nombre: { \$regex: ' J u a n', \$options: 'm' } }</code>
s	Permite el carácter <i>punto</i> (<code>.</code>) para representar <i>cualquier</i> carácter, incluido el carácter de nueva línea.	<code>{nombre:/ju.n/s}</code> <code>{nombre: { \$regex: 'ju.n', \$options: 's' } }</code>

Puedes encontrar más información relativa a las expresiones regulares y casos particulares en que se recomienda usar un tipo de expresión u otra en la documentación oficial de MongoDB acerca de **\$regex**.

F. CONSULTAS CON VECTORES

Para buscar elementos coincidentes dentro de un vector, procedemos con la misma sintaxis que si se tratase de cualquier otra clave, mediante el documento de consulta **{clave:valor}**, de forma que la clave es un vector, y el valor, bien **n** valores que debe contener el vector, o bien otro vector ordenado con el que queramos que coincida de forma exacta.

Por ejemplo:

- `db.coleccion.find({ mi_vector : valor })`
 - Coincide con todos los documentos en cuyo vector **mi_vector** aparezca, en la posición que sea el valor indicado.
- `db.coleccion.find({ mi_vector : [valor] })`
 - Coincide con todos los documentos en cuyo vector **mi_vector** aparezca únicamente el valor indicado.

Además, podemos utilizar como condiciones también expresiones regulares, o el resto de operadores que hemos visto.

Por otro lado, también podemos hacer referencia a un elemento concreto del vector por su índice, utilizando la notación punto y entre comillas:

```
db.coleccion.find({ "mi_vector.posicion" : [valor] })
```

G. EL OPERADOR \$ALL

Con **\$all** podemos especificar más de un elemento coincidente dentro del vector:

```
db.coleccion.find({ mi_vector : {$all:[valor1, valor2, ...]} })
```


H. EL OPERADOR \$SIZE

Mediante **\$size** podemos incluir condiciones sobre la longitud de los vectores:

```
db.coleccion.find({ mi_vector : {$size:tamaño} })
```

I. EL OPERADOR \$SLICE

El operador **\$slice** nos permite obtener un subconjunto de elementos del vector, con la siguiente sintaxis:

Opción	Descripción
clave: {\$slice: x}	Si $x > 0$, obtiene los x primeros elementos. Si $x < 0$, obtiene los últimos x elementos.
clave: {\$slice: [x , y] }	Obtiene y elementos a partir del elemento en posición x .

J. DOCUMENTOS INCRUSTADOS

Para realizar consultas sobre documentos incrustados, solo hay que especificar la ruta completa de claves, entrecomillada y separada por puntos:

```
db.coleccion.find({"ruta.a.la.clave":valor_o_condicion})
```

1.3. Cursores

Cuando realizamos una consulta, MongoDB nos devuelve los resultados mediante cursores, que son punteros a los resultados de la consulta. Los clientes que utilizan Mongo son quienes iteran sobre estos cursores para recuperar los resultados, y ofrece un conjunto de funcionalidades, como limitar los resultados, etc.

Cuando realizamos alguna consulta sobre una base de datos con muchos resultados, el cliente nos retorna únicamente 20 resultados y el mensaje **Type "it" for more**, para seguir *iterando* el cursor.

LIMIT, SKIP Y SORT

MongoDB nos permite realizar ciertas limitaciones sobre los resultados. Entre ellas, podemos destacar:

- **límite**: para limitar el número de resultados.
- **skip**: salta un número de resultados concreto.
- **sort**: ordena los resultados. Necesita un objeto JSON con las claves para ordenar, y un valor 1 para ordenar de forma ascendente, o -1 , descendente.

1.4. Introducción al Aggregation Framework

Las consultas de agregación que realizábamos con operadores como **GROUP BY**, **SUM** o **COUNT** en SQL, pueden realizarse con el **Aggregation Framework** de MongoDB.

Las consultas agregadas tienen la siguiente sintaxis:

```
db.coleccion.aggregate( [<pipeline>] )
```

El **pipeline** o tubería tiene un concepto similar a las tuberías de **Unix**: se pasan los resultados de una orden como entrada a otra, para obtener resultados de forma conjunta.

Las operaciones que podemos realizar dentro de estas consultas agregadas son:

- **\$project**: para realizar una proyección sobre un conjunto de datos de entrada, añadiendo, eliminando o recalculando campos para que la salida sea diferente.
- **\$match**: filtra la entrada para reducir el número de documentos, dejando solo aquellos que cumplen ciertas condiciones.
- **\$limit**: restringe el número de resultados.
- **\$skip**: ignora un número determinado de registros.

- **\$unwind**: convierte un vector para retornarlo separado en documentos.
- **\$group**: agrupa documentos según determinada condición.
- **\$sort**: ordena un conjunto de documentos, según el campo especificado.
- **\$geoNear**: se usa como datos geoespaciales, y devuelve los documentos ordenados por proximidad según un punto geoespacial.

Para realizar cálculos sobre los datos producidos por las tuberías, usaremos expresiones. Las expresiones son funciones que realizan cierta operación sobre un grupo de documentos, vector o campo concreto. Algunas de estas expresiones son **\$max**, **\$min**, **\$divide** o **\$substr**.

Puedes encontrar mucha más información sobre el Aggregation Framework en la documentación oficial de MongoDB.

Propiedad de McGraw-Hill®

1. MongoDB y Java

1.1. Drivers

Como ya sabemos, para conectarnos desde nuestras aplicaciones a una base de datos necesitamos de un controlador o **driver**.

MongoDB ofrece controladores oficiales para multitud de plataformas, entre las que encontramos C, C++, C#, NodeJS, Python, y como no, Java, entre muchas otras.

Centrándonos en Java, MongoDB nos ofrece dos **drivers**:

- El *driver* de Java para aplicaciones síncronas.
- El *driver* Reactive Streams para el procesamiento de **streams** asíncronos.

Aunque actualmente se tiende a la programación reactiva, vamos a trabajar con el *driver* síncrono de Java para facilitar la comprensión y centrarnos en el acceso propiamente dicho a los datos.

A. EL DRIVER DE JAVA

Mediante el **driver** de MongoDB para Java podemos conectarnos tanto a una base de datos local o remota como a un clúster de MongoDB Atlas. Este **driver** (MongoDB Java Driver) lo podemos encontrar en los repositorios de Maven, y proporciona una gran cantidad de clases e interfaces para facilitar el trabajo con MongoDB desde Java.

B. CONEXIÓN A UNA BASE DE DATOS

Para conectarnos y comunicarnos con una base de datos necesitamos un cliente. En el caso del **driver** de Java para MongoDB, el cliente se implementa mediante la clase MongoClient.

La clase MongoClient

La clase **MongoClient** representa un conjunto de conexiones a un servidor MongoDB. Estas conexiones son *thread-safe*, es decir, que varios hilos de ejecución pueden acceder de forma segura a ellas.

La forma de crear instancias de **MongoClient** es mediante el método **MongoClients.create()**. Además, generalmente, solo necesitaremos una instancia de esta clase, incluso en aplicaciones multihilo.

El método **create** de **MongoClients** toma como argumento una cadena de conexión (Connection String), con el siguiente formato simplificado (los parámetros más claros son opcionales):

```
mongodb:// [usuario:contraseña @] host[:puerto] /?opciones
```

Así pues, una forma de obtener, por ejemplo, una conexión al servidor local sería:

```
String uri = "mongodb://localhost:27017";  
MongoClient mongoClient = MongoClient.create(uri);
```

La clase **MongoClient**, entre otros, admite los métodos siguientes:

Método	Significado
getDatabase(String nombre)	Obtiene una referencia a una base de datos cuyo nombre le pasamos como argumento.
listDatabaseNames()	Obtiene una lista de strings (interfaz Mongolterable) con los nombres de las bases de datos del servidor.
close()	Cierra la conexión al servidor. Debe realizarse siempre cuando ya no se vaya a utilizar.

En el enlace **Connect to MongoDB**, de la sección correspondiente, dispones de más información acerca de estos y otros métodos de crear una conexión al servidor.

MongoDatabase

El método **getDatabase** de la clase **MongoClient** nos retorna una referencia a un objeto que implementa la interfaz **MongoDatabase**, que representa una conexión a una base de datos.

Esta interfaz define los siguientes métodos:

Método	Significado
getCollection(String nombre)	Obtiene una referencia a la colección.
listCollectionNames()	Obtiene una lista de <i>strings</i> (interfaz Mongolterable) con los nombres de las colecciones en la base de datos.
listCollections()	Obtiene una lista de referencias (MongoCollection) a las colecciones de la base de datos.
createCollection(String nombre)	Crea una nueva colección con el nombre indicado en la base de datos.
drop()	Elimina la base de datos.

En el caso práctico de este apartado, veremos cómo utilizar algunos de estos métodos

C. CONSULTAS

El método **getCollection()** de **MongoDatabase()** nos proporciona una colección de documentos, sobre los que vamos a poder realizar consultas mediante el método **find()**. Este método, que ya conocemos de la **shell** de MongoDB, nos permitirá realizar un filtrado de documentos de acuerdo con ciertos criterios.

Estos criterios se expresan como **filtros** (**query filters** en la documentación), y pueden contener varios operadores de consulta sobre algunos campos, que determinarán qué documentos de la colección se incluyen como resultados.

La clase **Filter** nos proporciona métodos de factoría para realizar estas consultas, de forma similar a como trabajábamos con la **shell** de MongoDB. Esta clase nos proporciona:

- Consulta vacía, con **Filters.empty()**.
- Operadores de comparación para realizar consultas de acuerdo con valores en la colección: **Filters.eq(clave, valor)**, **Filters.gt(clave, valor)**, **Filters.gte(clave, valor)**, **Filters.lt(clave, valor)** o **Filters.lte(clave, valor)**.
- Operadores lógicos para realizar operaciones lógicas sobre el resultado de otras consultas: **Filter.and(otros_filtros)**, **Filter.or(otros_filtros)**...
- Operadores de arrays. Nos permiten realizar consultas de acuerdo con el valor o la cantidad de elementos de un vector: **Filters.size(vector, tamaño)**.
- Otros operadores, como **Filter.exists()** o **Filter.regex()**, para comprobar la existencia de una clave o bien realizar una búsqueda por expresiones regulares.

Además de los filtros, también vamos a poder incluir operaciones de agregación, mediante el método **aggregate()** de una instancia de **MongoCollection**. Puedes consultar la documentación acerca de las agregaciones en la guía de operaciones de agregación de MongoDB, referenciada en los enlaces.

Por otra parte, la API del *driver* de MongoDB también nos permite realizar proyecciones de campos mediante la clase **Projections**, que ofrece los métodos **Projections.fields()**, **Projections.include()** o **projections.excludeID()**.

En el caso práctico veremos algunos de estos operadores en acción.

1.2. Spring Data MongoDB y API REST

Como sabemos, el proyecto Spring Data, incluido en la plataforma Spring, proporciona un *framework* para simplificar el acceso a datos y la persistencia sobre diferentes repositorios de información.

Dentro de este proyecto se encuentra Spring Data MongoDB, que proporciona integración con bases de datos MongoDB, mediante un modelo centrado en POJO, que interactúan con las colecciones de documentos y proporcionan un repositorio de acceso a datos.

En este apartado, y tomando el relevo de la unidad anterior, vamos a abordar el desarrollo de componentes de acceso a datos mediante Spring Data, así como microservicios que ofrezcan estos datos mediante una API REST; todo ello, siguiendo el patrón MVC que ya conocemos.

A. DEFINICIÓN DEL MODELO-DOCUMENTO

Una base de datos en MongoDB está compuesta por colecciones de documentos. Aunque estos documentos puedan tener diferentes estructuras entre sí o diferentes tipos de datos, el modelo sí que requiere de una estructura estática.

Así pues, lo primero que debemos hacer es crear una clase que represente este documento principal para MongoDB, que será el devuelto por las consultas que se vayan realizando.

En este contexto, son dos las principales anotaciones que utilizaremos:

- **@Document**, para indicar que una clase se corresponde con un objeto de dominio (**domain object**), que puede mapearse en la base de datos para ofrecer persistencia. Esta anotación para MongoDB sería el equivalente a **@Entity** en JPA. Si no se indica nada, se interpretará que el nombre de la colección que se va a utilizar se corresponde con el nombre de la clase en minúscula. Así pues, si tenemos la clase **com.mgh.ad.Persona**, se utilizará la colección «persona». No obstante, podemos indicar la colección con la que trabajamos, mediante los atributos **value** o **collection**, con las siguientes sintaxis:
 - **@Document(value="coleccion")**
 - **@Document("coleccion")**
 - **@Document(collection="coleccion")**
- **@Id**: se aplica sobre un campo, y sirve para indicar que el campo se utilizará como identificador. Como sabemos, todo documento en MongoDB requiere de un identificador. Si no se proporciona uno, el controlador asignará un **ObjectID** automáticamente. Es importante destacar que los tipos de datos que podemos utilizar como identificadores pueden ser tanto *strings* como **BigInteger**, ya que Spring se encargará de convertirlos al tipo **ObjectID**.

Además de estas, existen otras anotaciones más específicas que podemos utilizar. Si lo deseas, puedes consultarlas en la documentación de referencia de Spring Data MongoDB.

B. DEFINICIÓN DEL REPOSITORIO

Como sabemos, el repositorio es la interfaz encargada de gestionar el acceso a los datos. Para el caso de MongoDB, esta derivará de **MongoRepository**, que será una interfaz parametrizada por dos argumentos: **MongoRepository<T, Id>**:

- **T**: el tipo de documento, que se corresponderá a la clase definida en el modelo, e
- **Id**: el tipo de datos al que pertenecerá el identificador.

La interfaz **MongoRepository**, como hemos dicho, será específica para MongoDB, y derivará de las interfaces **CrudRepository** y **PagingAndSortingRepository**, de las que heredará todos sus métodos.

De este modo, en el repositorio únicamente deberemos declarar aquellos métodos que sean más específicos para nuestra aplicación, ya que todos los métodos para implementar operaciones CRUD, así como **findAll()** y **findById()** los heredarán de **MongoRepository**.

Para definir en el repositorio nuestras propias consultas, utilizaremos la anotación **@Query**, proporcionándole como valor la consulta en cuestión:

```
@Query(value="{ consulta_parametrizada }")
List<TipoDocumento> nombreMetodo(lista_parametros);
```

Para proporcionar parámetros a la consulta, estos se reciben como argumentos del método, y son referenciados por su orden en la consulta: **?0** para el primer argumento, **?1** para el segundo, etc.

C. DEFINICIÓN DEL SERVICIO

Los servicios se encargan de la capa de negocio de nuestra aplicación, y acceden a los datos a través del repositorio, enviando los resultados al controlador. Estos servicios, en general, se caracterizan por:

- Utilizar la anotación **@Service** para indicar a Spring que se está implementando un servicio.
- Por una parte, suele definirse la interfaz **Service**, y por otra realizarse la implementación mediante la clase **ServiceImpl**.
- Se utiliza la anotación **@Autowired** en las referencias a los repositorios para enlazar o inyectar el servicio en cuestión con dicho repositorio.
- Una vez que obtiene los datos del repositorio, los envía al controlador.

D. DEFINICIÓN DEL CONTROLADOR

Finalmente, nos queda la implementación del controlador, que ya conocemos de Spring. Recordemos las principales características de este:

- Utiliza la anotación **@RestController** a nivel de clase para indicar que se trata de un controlador **REST**.
- Utiliza la anotación **@RequestMapping** a nivel de clase para especificar la ruta base para los **endpoints** del servicio.
- Utiliza la anotación **@Autowired** en las propiedades que hacen referencia al servicio, para inyectar este de forma automática.
- Utiliza las anotaciones **@GetMapping**, **@PostMapping**, **@PutMapping**, **@DeleteMapping** sobre los métodos que implementarán peticiones de tipo **GET**, **POST**, **PUT** o **DELETE**, especificando su **Endpoint**.
- Utiliza la anotación **@PathVariable** en los argumentos de los métodos anteriores para obtener las variables a partir de la URL, que especificaremos entre llaves **{variable}** en el **endpoint**.

Propiedad de McGraw-Hill®