

Desarrollo de aplicaciones en conexión con bases de datos

Marc Gibert Ginestà

P06/M2109/02153

Índice

Introducción	5
Objetivos	6
1. Conexión y uso de bases de datos en lenguaje PHP	7
1.1. API nativa frente a API con abstracción	7
1.2. API nativa en MySQL	8
1.3. API nativa en PostgreSQL	12
1.4. Capa de abstracción PEAR::DB	17
1.4.1. Capa de abstracción del motor de la base de datos	19
1.4.2. Transacciones	24
1.4.3. Secuencias	24
2. Conexión y uso de bases de datos en lenguaje Java	27
2.1. Acceder al SGBD con JDBC	28
2.2. Sentencias preparadas	31
2.3. Transacciones	32
Resumen	34
Bibliografía	35

Introducción

Un curso de bases de datos quedaría incompleto si únicamente viéramos el funcionamiento y administración de los dos gestores anteriormente comentados. Uno de los principales objetivos de un SGBD es proporcionar un sistema de almacenamiento y consulta de datos al que se puedan conectar las aplicaciones que desarrollemos.

Así pues, en este capítulo vamos a abordar este tema, desde una perspectiva totalmente práctica, intentando exponer las bases para usar los SGBD vistos anteriormente desde algunos de los lenguajes de programación y conectores más usados. Los ejemplos proporcionados serán lo más simples posible para centrarnos en el tema que nos ocupa y no en las particularidades del lenguaje de programación en sí.

En primer lugar, veremos las herramientas que ofrece PHP para conectarse con bases de datos, y proporcionaremos algunos ejemplos.

A continuación, pasaremos a examinar la conexión JDBC a SGBD en general y a MySQL y PostgreSQL en particular, proporcionando también los ejemplos necesarios. También comentaremos algún aspecto avanzado como el de la persistencia de la conexión al SGBD.

Objetivos

El objetivo principal de esta unidad es conocer las diferentes técnicas de conexión a bases de datos que ofrecen PHP y Java.

Más concretamente, los objetivos que deberíais alcanzar al acabar el trabajo con la presente unidad son los siguientes:

- Conocer las posibilidades que PHP y Java ofrecen para la conexión y uso de bases de datos en general, y de MySQL y PostgreSQL en particular.
- Saber adaptar los programas desarrollados en estos lenguajes para que utilicen SGBD.

1. Conexión y uso de bases de datos en lenguaje PHP

El lenguaje de *script* PHP se ha popularizado extraordinariamente durante los últimos años, gracias a su sencillez y su sintaxis heredada de otros lenguajes como C, Perl o Visual Basic, que casi todos los desarrolladores ya conocían en mayor o menor grado.

Su fácil integración con los servidores web más populares (Apache, IIS, etc.), sin necesidad de recompilaciones o configuraciones complejas, ha contribuido también a que casi todos los proveedores de espacio web, desarrolladores de aplicaciones de software libre basadas en web, y proveedores de aplicaciones empresariales, lo usen para sus productos.

A lo largo de su corta historia ha progresado significativamente, y la versión 5.0 supone un paso adelante en la orientación a objetos, el tratamiento de excepciones y el trabajo con XML, lo que le hace parecerse en prestaciones a los lenguajes más maduros en el ámbito empresarial.


La versión 5.0 era la más actualizada en el momento de confección del presente material (finales del 2004). En este capítulo, no obstante, trabajaremos con la versión 4, ya que la versión 5.0 es muy reciente y los cambios que incorpora en cuanto a configuración, modelo de programación, etc. no les serán familiares a la mayoría de los estudiantes con conocimientos de PHP.

1.1. API nativa frente a API con abstracción

Desde la versión 2.0, PHP ha incorporado de forma nativa funciones para la conexión y uso de bases de datos. Al ser la rapidez una de las máximas de este lenguaje, y ante la ventaja de que proporciona mecanismos para la carga de librerías externas, se crearon unas librerías para cada motor de base de datos, que contenían las funciones necesarias para trabajar con él.

Estas API nativas son diferentes para cada SGBD, tanto en los nombres de las funciones que se utilizan para crear una conexión a la base de datos, lanzar una consulta, etc., como en el tratamiento de errores, resultados, etc.

Aunque se puede argumentar que al usar la API del SGBD concreto que utilizemos, dispondremos de operadores o funcionalidades específicas de ese motor que una librería estándar no puede proporcionar, con el paso del tiempo se ha visto que la utilización de estas API sólo está indicada (y aun así, no es recomendable) para aplicaciones que sepamos seguro que no van a cambiar el



Como actualmente hay aplicaciones web desarrolladas en PHP que usan la API concreta del SGBD para el que fueron pensadas, las revisaremos en este apartado.

SGBD con el que trabajan, ya que la revisión del código PHP, cuando hay un cambio de SGBD, es muy costosa y proclive a errores.

1.2. API nativa en MySQL

Para trabajar con la API nativa de MySQL en PHP, deberemos haber compilado el intérprete con soporte para este SGBD, o bien disponer ya del binario de PHP precompilado con el soporte incorporado.

En el caso de tenerlo que compilar, únicamente deberemos indicar como opción `--with-mysql`. Posteriormente, o en el caso de que ya dispongamos del binario, podemos validar que el soporte para MySQL está incluido correctamente en el intérprete con la ejecución del siguiente comando:

```
$ php -i | grep MySQL
supported databases => MySQL
MySQL Support => enabled
$
```

A partir de aquí, PHP proporciona unos parámetros de configuración que nos permitirán controlar algunos aspectos del funcionamiento de las conexiones con el SGBD, y las propias funciones de trabajo con la base de datos.

En cuanto a los parámetros, deberán situarse en el fichero `php.ini`, o bien configurarse para nuestra aplicación en concreto desde el servidor web. Destacan los siguientes:

`mysql.allow_persistent`: indica si vamos a permitir conexiones persistentes a MySQL. Los valores posibles son `true` o `false`.

`mysql.max_persistent`: número máximo de conexiones persistentes permitidas por proceso.

`mysql.max_links`: número máximo de conexiones permitidas por proceso, incluyendo las persistentes.

`mysql.connect_timeout`: tiempo que ha de transcurrir, en segundos, antes de que PHP abandone el intento de conexión al servidor.

Las conexiones persistentes son conexiones a la base de datos que se mantienen abiertas para evitar el tiempo de latencia que se pierde en conectar y desconectar. El intérprete, al ejecutar la sentencia de conexión a la base de datos, examina si hay alguna otra conexión abierta sin usar, y devuelve ésta en lugar de abrir una nueva. Lo mismo sucede al desconectar, el intérprete puede realizar la desconexión si hay suficientes conexiones aún abiertas, o bien mantener la conexión abierta para futuras consultas.

Por lo que respecta a la utilización de la API para la conexión y consulta de bases de datos, empezaremos con un ejemplo:

```
1 <?php
2 // Conectando y eligiendo la base de datos con que vamos a trabajar
3 $link = mysql_connect('host_mysql', 'usuario_mysql', 'password_mysql') or die
('No puedo conectarme: ' . mysql_error());
4 echo 'Conexión establecida';
5 mysql_select_db('mi_database',$link) or die('No he podido acceder a la base de
datos');
6
7 // Realizando una consulta SQL
8 $query = 'SELECT * FROM mi_tabla';
9 $result = mysql_query($query,$link) or die('Consulta errónea: ' . mysql_error());
10
11 // Mostramos los resultados en HTML
12 echo "<table>\n";
13 while ($line = mysql_fetch_array($result, MYSQL_ASSOC)) {
14     echo "\t<tr>\n";
15     foreach ($line as $col_value) {
16         echo "\t\t<td>$col_value</td>\n";
17     }
18     echo "\t</tr>\n";
19 }
20 echo "</table>\n";
21
22 // Liberamos el resultset
23 mysql_free_result($result);
24
25 // Cerramos la conexión
26 mysql_close($link);
27 ?>
```

En las cinco primeras líneas se establece la conexión y se selecciona la base de datos con que se va a trabajar. El código es bastante explícito y la mayoría de errores al respecto suelen deberse a una mala configuración de los permisos del usuario sobre la base de datos con la que debe trabajar.

Conviene estar muy atento, sobre todo a las direcciones de origen de la conexión, ya que, aunque podemos usar `localhost` como nombre de equipo, si el intérprete y el SGBD están en el mismo servidor, suele ocurrir que PHP resuelve `localhost` al nombre real del equipo e intenta conectarse con esta identificación. Así pues, debemos examinar cuidadosamente los archivos de registro de MySQL y los usuarios y privilegios del mismo si falla la conexión.

Para establecer una conexión persistente, debemos utilizar la función `mysql_pconnect()` con los mismos parámetros.

A continuación, se utiliza la función `mysql_query()` para lanzar la consulta a la base de datos.

La función `mysql_error()` es universal, y devuelve el último error ocurrido en el SGBD con nuestra conexión, o con la conexión `$link` que le indiquemos como parámetro.

La función `mysql_query()` puede devolver los siguientes resultados:

- FALSE si ha habido un error.
- Una referencia a una estructura si la sentencia es de consulta y ha tenido éxito.
- TRUE si la sentencia es de actualización, borrado o inserción y ha tenido éxito.

La función `mysql_affected_rows()` nos permite conocer el número de filas que se han visto afectadas por sentencias de actualización, borrado o inserción.

La función `mysql_num_rows()` nos permite conocer el número de filas devuelto por sentencias de consulta.

Una vez obtenido el recurso a partir de los resultados de la consulta, PHP proporciona multitud de formas de iterar sobre sus resultados o de acceder a uno de ellos directamente. Comentamos las más destacadas:

- `$fila = mysql_fetch_array($recurso, <tipo_de_array>)`

Esta función va iterando sobre el recurso, devolviendo una fila cada vez, hasta que no quedan más filas, y devuelve FALSE. La forma del *array* devuelto dependerá del parámetro `<tipo_de_array>` que puede tomar estos valores:

- `MYSQL_NUM`: devuelve un *array* con índices numéricos para los campos. Es decir, en `$fila[0]` tendremos el primer campo del SELECT, en `$fila[1]`, el segundo, etc.
- `MYSQL_ASSOC`: devuelve un *array* asociativo en el que los índices son los nombres de campo o alias que hayamos indicado en la sentencia SQL.
- `MYSQL_BOTH`: devuelve un *array* con los dos métodos de acceso.

Utilidad de la sentencia de conexión

Hemos proporcionado la sentencia SQL a la función y el enlace nos ha devuelto la sentencia de conexión. Esta funcionalidad es absolutamente necesaria si se trabaja con varias conexiones simultáneamente, si únicamente hay una conexión establecida en el *script*, este parámetro es opcional.

Recuperando el ejemplo inicial, entre las líneas 7 y 21:

```

7 // Realizando una consulta SQL
8 $query = 'SELECT * FROM mi_tabla';
9 $result = mysql_query($query,$link) or die('Consulta errónea: ' . mysql_error());
10
11 // Mostramos los resultados en HTML
12 echo "<table>\n";
13 while ($line = mysql_fetch_array($result, MYSQL_BOTH)) {
14     echo "\t<tr>\n";
15     for($i=0;$i<sizeof($line);$i++) {
16         echo "\t\t<td>$line[$i]</td>\n";
17     }
18     echo "\t\t<td>Nombre: $line['nombre']</td>";
19     echo "\t</tr>\n";
20 }
21 echo "</table>\n";

```

- `$objeto = mysql_fetch_object($recurso)`

Esta función va iterando sobre los resultados, devolviendo un objeto cada vez, de manera que el acceso a los datos de cada campo se realiza a través de las propiedades del objeto. Al igual que en el *array* asociativo, hay que vigilar con los nombres de los campos en consulta, evitando que devuelva campos con el mismo nombre fruto de combinaciones de varias tablas, ya que sólo podremos acceder al último de ellos:

Volvemos sobre el ejemplo inicial

```

7 // Realizando una consulta SQL
8 $query = 'SELECT nombre,apellidos FROM mi_tabla';
9 $result = mysql_query($query,$link) or die('Consulta errónea: ' . mysql_error());
10
11 // Mostramos los resultados en HTML
12 echo "<table>\n";
13 while ($object = mysql_fetch_array($result, MYSQL_BOTH)) {
14     echo "\t<tr>\n";
15     echo "\t\t<td>Nombre: " . $object->nombre . "</td>";
16     echo "\t\t<td>Apellidos: " . $object->apellidos . "</td>";
17     echo "\t</tr>\n";
18 }
19 echo "</table>\n";

```

- `$valor = mysql_result($recurso,$numero_de_fila,$numero_de_campo)`

Esta función consulta directamente un valor de un campo de una fila especificada. Puede ser útil si queremos conocer un resultado sin necesidad de realizar bucles innecesarios. Evidentemente, hemos de saber exactamente dónde se encuentra.

- `$exito = mysql_data_seek($recurso,$fila)`

Esta función permite mover el puntero dentro de la hoja de resultados representada por `$recurso`, hasta la fila que deseemos. Puede ser útil para

avanzar o para retroceder y volver a recorrer la hoja de resultados sin tener que ejecutar la sentencia SQL de nuevo. Si la fila solicitada no existe en la hoja de resultados, el resultado será FALSE, al igual que si la hoja no contiene ningún resultado. Es decir, el valor de `$fila` debe estar entre 0 (la primera fila) y `mysql_num_rows()-1`, excepto cuando no hay ningún resultado, en cuyo caso devolverá FALSE.

Finalmente, comentaremos las funciones de liberación y desconexión. En el primer caso, PHP realiza un excelente trabajo liberando recursos de memoria cuando la ejecución en curso ya no los va a utilizar más. Aun así, si la consulta devuelve una hoja de datos muy grande, puede ser conveniente liberar el recurso cuando no lo necesitamos.

Por lo que respecta al cierre de la conexión, tampoco suele ser necesario, ya que PHP cierra todas las conexiones al finalizar la ejecución y, además, el cierre siempre está condicionado a la configuración de las conexiones persistentes. Tal como ya hemos comentado, si activamos las conexiones persistentes (o bien hemos conectado con `mysql_pconnect`), esta función no tiene ningún efecto y, en todo caso, será PHP quien decida cuándo se va a cerrar cada conexión.

Ya hemos comentado que las API específicas para cada motor incluyen un conjunto de funciones que podía ayudar a trabajar con sus aspectos particulares, a continuación, enumeramos las más importantes:

- `mysql_field_flags`, `mysql_field_name`, `mysql_field_table`, `mysql_field_type`: estas funciones reciben como parámetro un `$recurso` y un índice de campo dentro de la consulta ejecutada y devuelven información sobre el campo; en concreto, sus restricciones, nombre, tabla a la que corresponden y tipo de campo. Pueden ser muy útiles para trabajar con consultas genéricas sobre bases de datos y/o campos que no conocemos al realizar el *script*.
- `mysql_insert_id`: esta función devuelve el último identificador obtenido de una inserción en un campo autoincremental.
- `mysql_list_dbs`, `mysql_list_tables`, `mysql_list_fields`: con distintos parámetros, estas funciones permiten consultar datos de administración del motor de la base de datos.

1.3. API nativa en PostgreSQL

Para trabajar con la API nativa de PostgreSQL en PHP, deberemos haber compilado el intérprete con soporte para este SGBD, o bien disponer ya del binario de PHP precompilado con el soporte incorporado.

Bibliografía

Hay otras funciones más específicas para obtener información sobre el cliente que origina la conexión, o sobre el propio servidor donde se está ejecutando. Conviene consultar la documentación para obtener información sobre usos más avanzados de esta API.

En el caso de tenerlo que compilar, únicamente debemos indicar como opción `--with-pgsql`. Posteriormente, o en el caso de que ya dispongamos del binario, podemos validar que el soporte para PostgreSQL está incluido correctamente en el intérprete con la ejecución del siguiente comando:

```
$ php -i | grep PostgreSQL
PostgreSQL
PostgreSQL Support => enabled
PostgreSQL(libpq) Version => 7.4.6
$
```

A partir de aquí, PHP proporciona unos parámetros de configuración que nos permitirán controlar algunos aspectos del funcionamiento de las conexiones con el SGBD, y las propias funciones de trabajo con la base de datos.

En cuanto a los parámetros, deberán situarse en el fichero `php.ini` o bien configurarse para nuestra aplicación en concreto desde el servidor web. Destacan los siguientes:

- `pgsql.allow_persistent`: indica si vamos a permitir el uso de conexiones persistentes. Los valores son `true` o `false`.
- `pgsql.max_persistent`: número máximo de conexiones persistentes permitidas por proceso.
- `pgsql.max_links`: número máximo de conexiones permitidas por proceso, incluyendo las persistentes.
- `pgsql.auto_reset_persistent`: detecta automáticamente conexiones persistentes cerradas y las elimina.

Este parámetro disminuye ligeramente el rendimiento del sistema.

Por lo que respecta a la utilización de la API para la conexión y consulta de bases de datos, reproduciremos el anterior ejemplo:

```
1 <?php
2 // Conectando y eligiendo la base de datos con que vamos a trabajar
3 $link = pg_connect("host=host_psql port=5432 dbname=mi_database user=user_psql
4 password=pass_psql");
5 $stat = pg_connection_status($link);
6 if ($stat === 0) {
7     echo 'Conexión establecida';
8 }
```

```

7 } else {
8     die 'No puedo conectarme';
9 }
10
11 // Realizando una consulta SQL
12 $query = 'SELECT * FROM mi_tabla';
13 $result = pg_query($link,$query) or die('Consulta errónea: ' . pg_last_error());
14
15 // Mostramos los resultados en HTML
16 echo "<table>\n";
17 while ($line = pg_fetch_array($result, PGSQL_ASSOC)) {
18     echo "\t<tr>\n";
19     foreach ($line as $col_value) {
20         echo "\t\t<td>$col_value</td>\n";
21     }
22     echo "\t</tr>\n";
23 }
24 echo "</table>\n";
25
26 // Liberamos el resultset
27 pg_free_result($result);
28
29 // Cerramos la conexión
30 pg_close($link);
31 ?>

```

En las diez primeras líneas, establecemos la conexión y comprobamos que se ha realizado correctamente.

A diferencia de MySQL, la selección de la base de datos se hace en el momento de la conexión. En cambio, la comprobación de la conexión es un poco más complicada.

Para establecer una conexión persistente, debemos utilizar la función `pg_pconnect()` con los mismos parámetros.

A continuación, se utiliza la función `pg_query()` para lanzar la consulta a la base de datos.

Para comprobar errores, la API de PostgreSQL distingue entre un error de conexión, y errores sobre los recursos devueltos. En el primer caso, deberemos usar `pg_connection_status()`, mientras que en el segundo podemos optar por `pg_last_error()` o bien `pg_result_error($recurso)` para obtener el mensaje de error que pueda haber devuelto un recurso en concreto.

Recordad

La mayoría de los errores en este aspecto se originan por una mala configuración de los permisos de los usuarios en el SGBD. Una forma de probarlo que suele ofrecer más información es intentar la conexión con el cliente de la base de datos desde la línea de comandos, especificando el mismo usuario, base de datos y contraseña que estamos utilizando en el *script*.

Utilidad de la sentencia de conexión

Aquí se aplican los mismos consejos que dábamos en el apartado anterior, y las mismas consideraciones en cuanto al parámetro `$link` y su opcionalidad si únicamente tenemos una conexión establecida con el mismo usuario y contraseña.

La función `pg_query()` puede devolver los siguientes resultados:

- FALSE si ha habido un error.
- Una referencia a una estructura si la sentencia ha tenido éxito.

La función `pg_affected_rows($recurso)` nos permite conocer el número de filas que se han visto afectadas por sentencias de actualización, borrado o inserción. Esta función deberá recibir como parámetro el recurso devuelto por la función `pg_query()`.

La función `pg_num_rows($recurso)` nos permite conocer el número de filas devuelto por sentencias de consulta.

Una vez obtenido el recurso a partir de los resultados de la consulta, PHP proporciona multitud de formas de iterar sobre sus resultados o de acceder a uno de ellos directamente. Comentamos las más destacadas:

- `$fila = pg_fetch_array($recurso, <tipo_de_array>)`

Esta función va iterando sobre el recurso, devolviendo una fila cada vez, hasta que no quedan más filas y devuelve FALSE. La forma del *array* devuelto, dependerá del parámetro `<tipo_de_array>` que puede tomar estos valores:

- PG_NUM: devuelve un *array* con índices numéricos para los campos. Es decir, en `$fila[0]` tendremos el primer campo del SELECT, en `$fila[1]`, el segundo, etc.
- PG_ASSOC: devuelve un *array* asociativo donde los índices son los nombres de campo o alias que hayamos indicado en la sentencia SQL.
- PG_BOTH: devuelve un *array* con los dos métodos de acceso.

Recuperando el ejemplo inicial, entre las líneas 11 y 24:

```
11 // Realizando una consulta SQL
12 $query = 'SELECT * FROM mi_tabla';
13 $result = pg_query($query,$link) or die('Consulta errónea: ' . pg_last_error());
14// Mostramos los resultados en HTML
15 echo "<table>\n";
16 while ($line = pg_fetch_array($result, PGSQL_BOTH)) {
17     echo "\t<tr>\n";
18     for($i=0;$i<sizeof($line);$i++) {
19         echo "\t\t<td>$line[$i]</td>\n";
20     }
21     echo "\t\t<td>Nombre: $line['nombre']</td>";
22     echo "\t</tr>\n";
23 }
24 echo "</table>\n";
```

- `$objeto = pg_fetch_object($recurso)`

Esta función va iterando sobre los resultados, devolviendo un objeto cada vez, de forma que el acceso a los datos de cada campo se realiza por medio de las propiedades del objeto. Al igual que en el *array* asociativo, hay que vigilar con los nombres de los campos en consulta, evitando que devuelva campos con el mismo nombre fruto de combinaciones de varias tablas, ya que sólo podremos acceder al último de ellos.

Volvemos sobre el ejemplo inicial

```
11 // Realizando una consulta SQL
12 $query = 'SELECT nombre,apellidos FROM mi_tabla';
13 $result = pg_query($query,$link) or die('Consulta errónea: ' . pg_last_error());
14 // Mostramos los resultados en HTML
15 echo "<table>\n";
16 while ($object = pg_fetch_array($result, PGSQL_BOTH)) {
17     echo "\t<tr>\n";
18     echo "<td>Nombre: " . $object->nombre . "</td>";
19     echo "<td>Apellidos: " . $object->apellidos . "</td>";
20     echo "\t</tr>\n";
21 }
22 echo "</table>\n";
```

Podemos pasar a la función `pg_fetch_object()` un segundo parámetro para indicar la fila concreta que queremos obtener:

```
$resultado = pg_fetch_all($recurso)
```

Esta función devuelve toda la hoja de datos correspondiente a `$recurso`; es decir, una *array* con todas las filas y columnas que forman el resultado de la consulta.

- `$exito = pg_result_seek($recurso,$fila)`

Esta función permite mover el puntero dentro de la hoja de resultados representada por `$recurso` hasta la fila que deseemos. Deben tomarse las mismas consideraciones que en la función `mysql_data_seek()`.

En cuanto a la liberación de recursos y la desconexión de la base de datos, es totalmente aplicable lo explicado para MySQL, incluyendo los aspectos relacionados con las conexiones persistentes.

Al igual que en MySQL, PHP también proporciona funciones específicas para trabajar con algunos aspectos particulares de PostgreSQL. Al tener éste más funcionalidad que se aleja de lo estándar debido a su soporte a objetos, estas

funciones cobrarán más importancia. A continuación comentamos las más destacadas:

- `pg_field_name`, `pg_field_num`, `pg_field_size`, `pg_field_type`: estas funciones proporcionan información sobre los campos que integran una consulta. Sus nombres son suficientemente explícitos acerca de su cometido.
- `pg_last_oid`: esta función nos devuelve el OID obtenido por la inserción de una tupla si el recurso que recibe como parámetro es el correspondiente a una sentencia INSERT. En caso contrario devuelve FALSE.
- `pg_lo_create`, `pg_lo_open`, `pg_lo_export`, `pg_lo_import`, `pg_lo_read`, `pg_lo_write`: estas funciones (entre otras) facilitan el trabajo con objetos grandes (LOB) en PostgreSQL.

```
<?php
  $database = pg_connect("dbname=jacarta");
  pg_query($database, "begin");
  $oid = pg_lo_create($database);
  echo "$oid\n";
  $handle = pg_lo_open($database, $oid, "w");
  echo "$handle\n";
  pg_lo_write($handle, "large object data");
  pg_lo_close($handle);
  pg_query($database, "commit");
?>
```

Las funciones `pg_lo_import` y `pg_lo_export` pueden tomar ficheros como parámetros, facilitando la inserción de objetos binarios en la base de datos.

1.4. Capa de abstracción PEAR::DB

El PEAR (PHP *extension and application repository*) se define como un marco de trabajo y un sistema de distribución de librerías reutilizables para PHP. Es similar en concepto al CPAN (*comprehensive perl archive network*) del lenguaje Perl o al PyPI (*Python package index*) de Python.

El PEAR pretende proporcionar una librería estructurada de código y librerías reutilizables, mantener un sistema para proporcionar a la comunidad herramientas para compartir sus desarrollos y fomentar un estilo de codificación estándar en PHP.

PEAR, debe ser el primer recurso para solventar cualquier carencia detectada en las funciones nativas de PHP. Como buena práctica general en el mundo del software libre, siempre es mejor usar, aprender o mejorar a partir de lo que

Nota

Hay otras funciones que tienen el mismo cometido que combinaciones de algunas de las funciones comentadas anteriormente (por ejemplo, `pg_select` o `pg_insert`, `pg_copy_from`), pero que no se comentan en este material por su extensión y por su poco uso.

PEAR

Si nuestra instalación de PHP es reciente, ya dispondremos de PEAR instalado (a no ser que lo hayamos compilado con la opción `--without-pear`).

han hecho otros, que proponemos reinventar la rueda. Además, si hacemos mejoras a las librerías que usemos de PEAR, siempre podemos contribuir a esos cambios mediante las herramientas que nos proporciona.

En cierta forma, PEAR se comporta como un gestor de paquetes más de los que pueden incorporar las distribuciones GNU/Linux más recientes (como apt, yum o YOU). Este gestor de paquetes se compone del ejecutable 'pear' al que podemos proporcionar un conjunto de parámetros según las acciones que deseemos realizar:

```
$ pear list
Installed packages:
=====
Package      Version  State
Archive_Tar  1.2      stable
Console_Getopt 1.2      stable
DB           1.6.8    stable
http        1.3.3    stable
Mail        1.1.4    stable
Net_SMTP    1.2.6    stable
Net_Socket  1.0.5    stable
PEAR        1.3.4    stable
PhpDocumentor 1.3.0RC3 beta
XML_Beautifier 1.1      stable
XML_Parser  1.2.2    stable
XML_RPC     1.1.0    stable
XML_Util    1.1.1    stable
```

PEAR (y PHP) ya viene con un conjunto de paquetes instalados, lo que se denomina el PFC (PHP *foundation classes*). Estos paquetes proporcionan a PHP la mínima funcionalidad necesaria para que PEAR funcione y para que dispongamos de las librerías básicas de PHP.

A continuación presentamos las opciones más habituales de PEAR:

Comando	Resultado
pear list	Lista de los paquetes instalados.
pear list-all	Lista de todos los paquetes disponibles en PEAR.
pear list-upgrades	Lista de los paquetes instalados con actualización disponible.
pear info <paquete>	Proporciona información sobre el paquete.
pear install <paquete>	Descarga e instala el paquete.
pear search <texto>	Busca paquetes en el repositorio PEAR.
pear upgrade <paquete>	Actualiza el paquete si es necesario.
pear upgrade-all	Actualiza todos los paquetes instalados con actualización disponible.
pear uninstall <paquete>	Desinstala el paquete.

1.4.1. Capa de abstracción del motor de la base de datos

Parece evidente que, para la inmensa mayoría de aplicaciones basadas en PHP, el uso de su librería nativa de acceso a bases de datos va a condicionar el SGBD a usar con la aplicación. En aplicaciones comerciales, o que no pueden ni desean estar cerradas a un único motor, no será imprescindible disponer de unas funciones que encapsulen la comunicación con el SGBD y que sean independientes de éste en las interfaces que ofrecen, mientras que internamente llamarán a las funciones nativas del SGBD concreto con que se esté trabajando en cada momento.

Así pues, y buscando en PEAR, encontramos el módulo 'DB', una capa de abstracción y encapsulamiento de la comunicación con el SGBD. Al tener que incorporar todas las funcionalidades de los motores que soporta, el resultado será siempre el mínimo conjunto de prestaciones comunes a todos los SGBD. Las prestaciones más destacadas que ofrece la versión actual 1.6.8 son las siguientes:

Versiones

La versión 1.6.8. era la más actualizada en el momento de elaboración de este material (finales de 2004).

- Interfaz orientada a objetos.
- Una sintaxis común para identificar SGBD y cadenas de conexión.
- Emulación de "sentencias preparadas" en los motores que no las soportan.
- Códigos de errores comunes.
- Emulación de secuencias o autoincrementos en SGBD que no los soportan.
- Soporte para transacciones.
- Interfaz para obtener información del metadato (información sobre la tabla o la base de datos).
- Compatible con PHP4 y PHP5.
- Motores soportados: dbase, fbsql, interbase, informix, msql, mssql, mysql, mysqli, oci8, odbc, pgsql, sqlite y sybase.

```
1 <?php
2 // Incluimos la librería una vez instalada mediante PEAR
3 require_once 'DB.php';
4
5 // Creamos la conexión a la base de datos, en este caso PostgreSQL
6 $db =& DB::connect('pgsql://usuario:password@servidor/basededatos');
7
8 // Comprobamos error en la conexión
9 if (DB::isError($db)) {
10 die($db->getMessage());
11 }
12
```

```
13 // Realizamos la consulta:
14 $res =& $db->query('SELECT * FROM clients');
15
16 // Comprobamos que la consulta se ha realizado correctamente
17 if (DB::isError($res)) {
18 die($res->getMessage());
19 }
20
21 // Iteramos sobre los resultados
22 while ($row =& $res->fetchRow()) {
23 echo $row[0] . "\n";
24 }
25
26 // Liberamos la hoja de resultados
27 $res->free();
28
29 // Desconectamos de la base de datos
30 $db->disconnect();
31 ?>
```

La estructura del código y hasta la sintaxis de las sentencias es similar a los ejemplos nativos vistos anteriormente, exceptuando las partes de las sentencias que hacían referencia al motor de base de datos en particular.

A continuación, vamos a avanzar por el código ampliando la información sobre cada paso.

La conexión se especifica mediante una sintaxis de tipo DSN (*data source name*). Los DSN admiten multitud de variantes, dependiendo del motor al que nos conectemos, pero en casi todos los casos, tienen la forma siguiente:

```
motorphp://usuario:contraseña@servidor/basededatos?opcion=valor
```

- Conexión a MySQL

```
mysql://usuario:password@servidor/basededatos
```

- Conexión a MySQL a través de un *socket* UNIX:

```
mysql://usuario:password@unix(/camino/al/socket)/basededatos
```

- Conexión a PostgreSQL

```
pgsql://usuario:password@servidor/basededatos
```

- Conexión a PostgreSQL en un puerto específico:

```
pgsql://usuario:password@tcp(servidor:1234)/basededatos
```

En cualquier llamada a un método del paquete DB, éste puede devolver el objeto que le corresponde (una hoja de resultados, un objeto representando la conexión, etc.) o bien un objeto que represente el error que ha tenido la llamada. De ésta manera, para comprobar los errores que puede originar cada sentencia o intento de conexión, bastará con comprobar el tipo del objeto devuelto:

```
8 // Comprobamos error en la conexión
9 if (DB::isError($db)) {
10     die($db->getMessage());
11 }
```

La clase `DB_Error` ofrece varios métodos. A pesar de que el más utilizado es `getMessage()`, `getDebugInfo()` o `getCode()` pueden ampliar la información sobre el error.

Para realizar consultas, disponemos de dos mecanismos diferentes:

- Enviar la consulta directamente al SGBD.
- Indicar al gestor que prepare la ejecución de una sentencia SQL y posteriormente indicarle que la ejecute una o más veces.

En la mayoría de los casos, optaremos por el primer mecanismo y podremos proceder como sigue:

```
13 // Realizamos la consulta:
14 $res =& $db->query('SELECT * FROM clients');
```

En ocasiones nos podemos encontrar ejecutando la misma consulta varias veces, con cambios en los datos o en el valor de las condiciones. En estos casos, es más indicado utilizar el segundo mecanismo.

Supongamos que tenemos que insertar un conjunto de clientes en nuestra base de datos. Las sentencias que ejecutaríamos serían parecidas a las siguientes:

```
INSERT INTO Clientes (nombre, nif) VALUES ('José Antonio Ramírez', '29078922Z');
INSERT INTO Clientes (nombre, nif) VALUES ('Miriam Rodríguez', '45725248T');
...
```

En lugar de esto, podemos indicarle al motor de la base de datos que prepare la sentencia, del modo siguiente:

```
$sth = $db->prepare('INSERT INTO Clientes (nombre,nif) VALUES (?,?)');
```

Utilizaremos la variable `$sth` que nos ha devuelto la sentencia `prepare` cada vez que ejecutemos el *query*:

```
$db->execute($sth, 'José Antonio Ramírez', '29078922Z');
$db->execute($sth, 'Miriam Rodriguez', '45725248T');
```

Ejemplo

Pensemos en un conjunto de actualizaciones o inserciones seguidas o en un conjunto de consultas donde vamos cambiando un intervalo de fechas o el identificador del cliente sobre el que se realizan.

También podemos pasar una *array* con todos los valores:

```
$datos = array('Miriam Rodriguez', '45725248T');
$db->execute($sth, $datos);
```

Y tenemos la opción de pasar una *array* de dos dimensiones con todas las sentencias a ejecutar, mediante el método `executeMultiple()`:

```
$todosDatos = array(array('José Antonio Ramírez', '29078922Z'),
                    array('Miriam Rodriguez', '45725248T'));
$sth = $db->prepare('INSERT INTO Clientes (nombre,nif) VALUES (?, ?)');
$db->executeMultiple($sth, $todosDatos);
```

A continuación, examinaremos los métodos de iterar sobre los resultados. En el ejemplo inicial hemos utilizado la función `fetchRow()` de la manera siguiente:

```
21 // Iteramos sobre los resultados
22 while ($row =& $res->fetchRow()) {
23     echo $row[0] . "\n";
24 }
```

Pero también disponemos de la función `fetchInto()`, que recibe como parámetro el *array* donde queremos que se almacene el resultado:

```
21 // Iteramos sobre los resultados
22 while ($res->fetchRow()) {
23     echo $row[0] . "\n";
24 }
```

Tanto `fetchRow()` como `fetchInto()` aceptan otro parámetro para indicar el tipo de estructura de datos que va a almacenar en `$row`:

- `DB_FETCHMODE_ORDERED`: es la opción por defecto. Almacena el resultado en una *array* con índice numérico.
- `DB_FETCHMODE_ASSOC`: almacena el resultado en una *array* asociativa en el que las claves son el nombre del campo.
- `DB_FETCHMODE_OBJECT`: almacena el resultado en un objeto donde dispondremos de atributos con el nombre de cada campo para obtener el valor en cada iteración.

```
21 // Iteramos sobre los resultados en modo asociativo
22 while ($res->fetchInto (($row,DB_FETCHMODE_ASSOC)) {
23     echo $row['nombre'] . "\n";
24     echo $row['nif'] . "\n";
25 }
```

o bien:

```
21 // Iteramos sobre los resultados en modo objeto
22 while ($res->fetchInto (($row,DB_FETCHMODE_OBJECT)) {
23     echo $row->nombre . "\n";
24     echo $row->nif . "\n";
25 }
```

La función `fetchInto()` acepta un tercer parámetro para indicar el número de fila que queremos obtener, en caso de que no deseemos iterar sobre la propia función:

```
21 // Obtenemos la tercera fila de la consulta
22 $res->fetchInto ($row,DB_FETCHMODE_ASSOC,3); {
23     echo $row->nombre . "\n";
24     echo $row->nif . "\n";
```

Hay otros métodos para obtener diferentes vistas del resultado como los siguientes:

- `getAll()`: obtiene una *array* de dos dimensiones con toda la hoja de resultados. Tendría una forma como la siguiente:

```
Array
(
    [0] => Array
        (
            [cf] => Juan
            [nf] => 5
            [df] => 1991-01-11 21:31:41
        )
    [1] => Array
        (
            [cf] => Kyu
            [nf] => 10
            [df] => 1992-02-12 22:32:42
        )
)
```

- `getRow()`: devuelve sólo la primera fila de la hoja de resultados.
- `getCol()`: devuelve sólo la columna indicada de la hoja de resultados.

De la misma manera que en las librerías nativas, hay métodos que proporcionan información sobre la consulta en sí:

- `numRows()`: número de filas de la hoja de resultados.
- `numCols()`: número de columnas de la hoja de resultados.
- `affectedRows()`: número de filas de la tabla afectadas por la sentencia de actualización, inserción o borrado.

1.4.2. Transacciones

PEAR::DB proporciona mecanismos para tratar las transacciones independientemente del SGBD con que trabajemos.

Como ya hemos comentado, la operativa con las transacciones está relacionada con las sentencias `begin`, `commit` y `rollback` de SQL. PEAR::DB envuelve estas sentencias en llamadas a métodos suyos, del modo siguiente:

```
// Desactivamos el comportamiento de COMMIT automático.
$db->autocommit(false);
..
..
if (...) {
    $db->commit();
} else {
    $db->rollback();
}
```

1.4.3. Secuencias

PEAR::DB incorpora un mecanismo propio de secuencias (AUTO_INCREMENT en MySQL), que es independiente de la base de datos utilizada y que puede ser de gran utilidad en identificadores, claves primarias, etc. El único requisito es que se usen sus métodos de trabajo con secuencias, siempre que se esté trabajando con esa base de datos; es decir, no se debe crear la secuencia en el SGBD y, después, trabajar con ella con los métodos que ofrece PEAR::DB. Si la secuencia la creamos mediante la base de datos, entonces deberemos trabajar con ella con las funciones extendidas de SQL que nos proporcione ese SGBD (en PostgreSQL la función `nextval()` o en MySQL la inserción del valor 0 en un campo AUTO_INCREMENT).

Atención

En MySQL sólo funcionará el soporte de transacciones si la base de datos está almacenada con el mecanismo InnoDB.

En PostgreSQL no hay restricción alguna.

En ningún sitio...

... se hace un `begin`. Al desactivar el `autocommit` (que está activado por defecto) todas las sentencias pasarán a formar parte de una transacción, que se registrará como definitiva en la base de datos al llamar al método `commit()` o bien se desechará al llamar al método `rollback()`, volviendo la base de datos al estado en el que estaba después del último `commit()`.

Disponemos de las siguientes funciones para trabajar con secuencias:

- `createSequence($nombre_de_secuencia)`: crea la secuencia o devuelve un objeto `DB_Error` en caso contrario.
- `nextId($nombre_de_secuencia)`: devuelve el siguiente identificador de la secuencia.
- `dropSequence($nombre_de_secuencia)`: borra la secuencia.

```
// Creamos la secuencia:
$tmp = $db->createSequence('miSecuencia');
if (DB::isError($tmp)) {
    die($tmp->getMessage());
}

// Obtenemos el siguiente identificador
$id = $db->nextId('mySequence');
if (DB::isError($id)) {
    die($id->getMessage());
}

// Usamos el identificador en una sentencia
$res =& $db->query("INSERT INTO miTabla (id, texto) VALUES ($id, 'Hola')");

// Borramos la secuencia
$tmp = $db->dropSequence('mySequence');

if (DB::isError($tmp)) {
    die($tmp->getMessage());
}
```

Finalmente, en el aspecto relacionado con los metadatos de las tablas, `PEAR::DB` ofrece la función `tableInfo()`, que proporciona información detallada sobre una tabla o sobre las columnas de una hoja de resultados obtenida de una consulta.

```
$info = $db->tableInfo('nombretabla');
print_r($info);
```

O bien:

```
$res =& $db->query('SELECT * FROM nombretabla');
$info = $db->tableInfo($res);
print_r($info);
```

El resultado será similar al siguiente:

```
[0] => Array (
    [table] => nombretabla
    [name] => nombre
```

```
[type] => string
[len] => 255
[flags] =>
)
[1] => Array (
[table] => nombretabla
[name] => nif
[type] => string
[len] => 20
[flags] => primary key not null
)
```

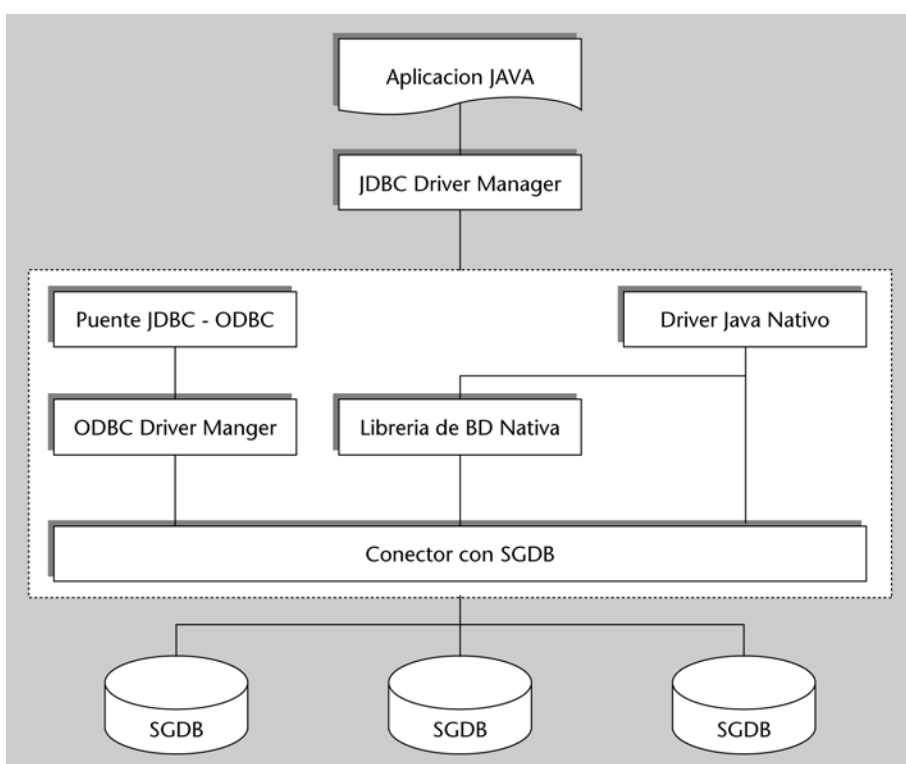
Funcionalidades

Hay funcionalidades más avanzadas de esta librería que aumentan continuamente. De todas formas, con las presentadas basta para identificar las ventajas de trabajar con una capa de abstracción del motor de base de datos donde se almacenan los datos de nuestra aplicación.

2. Conexión y uso de bases de datos en lenguaje Java

El acceso a bases de datos desde Java se realiza mediante el estándar JDBC (*Java data base connectivity*), que permite un acceso uniforme a las bases de datos independientemente del SGBD. De esta manera, las aplicaciones escritas en Java no necesitan conocer las especificaciones de un SGBD en particular, basta con comprender el funcionamiento de JDBC. Cada SGBD que quiera utilizarse con JDBC debe contar con un adaptador o controlador.

La estructura de JDBC se puede expresar gráficamente como sigue:



Hay *drivers* para la mayoría de SGBD, tanto de software libre como de código abierto. Además, hay *drivers* para trabajar con otros tipos de datos (hojas de cálculo, ficheros de texto, etc.) como si fueran SGBD sobre los que podemos realizar consultas SQL.

Para usar la API JDBC con un SGBD en particular, necesitaremos el *driver* concreto del motor de base de datos, que media entre la tecnología JDBC y la base de datos. Dependiendo de múltiples factores, el *driver* puede estar escrito completamente en Java, o bien haber usado métodos JNI (*Java native interface*) para interactuar con otros lenguajes o sistemas.

La última versión de desarrollo de la API JDBC proporciona también un puente para conectarse a SGBD que dispongan de *drivers* ODBC (*open database*

connectivity). Este estándar es muy común sobre todo en entornos Microsoft y sólo debería usarse si no disponemos del *driver* nativo para nuestro SGBD.

En el caso concreto de MySQL y PostgreSQL, no tendremos ningún problema en encontrar los drivers JDBC:

- MySQL Connector/J: es el *driver* oficial para MySQL y se distribuye bajo licencia GPL. Es un *driver* nativo escrito completamente en Java.
- JDBC para PostgreSQL: es el *driver* oficial para PostgreSQL y se distribuye bajo licencia BSD. Es un *driver* nativo escrito completamente en Java.

Tanto uno como otro, en su distribución en formato binario, consisten en un fichero .jar (*Java archive*) que debemos situar en el CLASSPATH de nuestro programa para poder incluir sus clases.

Java incluye la posibilidad de cargar clases de forma dinámica. Éste es el caso de los controladores de bases de datos: antes de realizar cualquier interacción con las clases de JDBC, es preciso registrar el controlador. Esta tarea se realiza con el siguiente código:

```
String controlador = "com.mysql.jdbc.Driver"  
Class.forName(controlador).newInstance();
```

o bien:

```
Class.forName("org.postgresql.Driver");
```

A partir de este momento, JDBC está capacitado para interactuar con MySQL o PostgreSQL.

2.1. Acceder al SGBD con JDBC

La interfaz JDBC está definida en la librería `java.sql`. Vamos a importar a nuestra aplicación Java todas las clases definidas en ella.

```
import java.sql.*;
```

Puesto que JDBC puede realizar conexiones con múltiples SGBD, la clase `DriverManager` configura los detalles de la interacción con cada uno en particular. Esta clase es la responsable de realizar la conexión, entregando un objeto de la clase `Connection`.

```
String url="jdbc:mysql://localhost/demo";
String usuario="yo"
String contrasenia="contraseña"
Connection conexion = DriverManager.getConnection (url,usuario,contrasenia);
```

El destino de la conexión se especifica mediante un URL de JDBC con la sintaxis siguiente:

```
jdbc:<protocolo_sgbd>:<subnombre>
```

La parte `protocolo_sgbd` de la URL especifica el tipo de SGBD con el que se realizará la conexión, la clase `DriverManager` cargará el módulo correspondiente a tal efecto.

El `subnombre` tiene una sintaxis específica para cada SGDB que tanto para MySQL como para PostgreSQL es `//servidor/base_de_datos`.

Las sentencias en JDBC también son objetos que deberemos crear a partir de una conexión:

```
Statement sentenciaSQL = conexion.createStatement();
```

Al ejecutar una sentencia, el SGBD entrega unos resultados que JDBC también representa en forma de objeto, en este caso de la clase `ResultSet`:

La variable `res` contiene el resultado de la ejecución de la sentencia, y proporciona un cursor que permite leer las filas una a una.

```
ResultSet res = sentencia.executeQuery("SELECT * FROM tabla");
```

Para acceder a los datos de cada columna de la hoja de resultados, la clase `ResultSet` dispone de varios métodos según el tipo de la información de la columna:

<code>getArray()</code>	<code>getInt()</code>
<code>getClob()</code>	<code>getBoolean()</code>
<code>getString()</code>	<code>getLong()</code>
<code>getAsciiStream()</code>	<code>getByte()</code>
<code>getDate()</code>	<code>getObject()</code>
<code>getTime()</code>	<code>getObject()</code>
<code>getBigDecimal()</code>	<code>getBytes()</code>
<code>getDouble()</code>	<code>getBytes()</code>
<code>getTimestamp()</code>	<code>getRef()</code>
<code>getBinaryStream()</code>	<code>getRef()</code>
<code>getFloat()</code>	<code>getCharacterStream()</code>
<code>getURL()</code>	<code>getShort()</code>
<code>getBlob()</code>	

```
import java.sql.*;
// Atención, no debe importarse com.mysql.jdbc ya que se carga dinámicamente!!

public static void main(String[] args) {

    // Cargamos el driver JDBC para MySQL
    String controlador = "com.mysql.jdbc.Driver"
    Class.forName(controlador).newInstance();

    // Conectamos con la BD
    String url="jdbc:mysql://localhost/uoc";
    String usuario="yo"
    String contrasenia="contraseña"
    Connection conexion = DriverManager.getConnection (url,usuario,contrasenia);

    // Creamos una sentencia SQL
    Statement sentenciaSQL = conexion.createStatement();
    // Ejecutamos la sentencia
    ResultSet res = sentencia.executeQuery("SELECT * FROM tabla");

    // Iteramos sobre la hoja de resultados
    while (res.next()) {
        // Obtenemos el campo 'nombre' en forma de String
        System.out.println(res.getString("nombre") );
    }

    // Finalmente, se liberan los recursos utilizados.
    res.close();
    sentencia.close();
    conexion.close();
}
```

En el ejemplo anterior no se ha previsto nada para tratar con los errores que puedan producirse, porque en Java el tratamiento de errores se realiza a través de Exceptions. En JDBC se han previsto excepciones para los errores que pueden producirse a lo largo de todo el uso de la API: conexión, ejecución de la sentencia, etc.

Revisemos el ejemplo, utilizando excepciones para tratar los errores.

```
import java.sql.*;
// Atención, no debe importarse com.mysql.jdbc ya que se carga
// dinámicamente!!

public static void main(String[] args) {

    try {
        // Cargamos el driver JDBC para MySQL
        String controlador = "com.mysql.jdbc.Driver"
        Class.forName(controlador).newInstance();
    } catch (Exception e) {
        System.err.println("No puedo cargar el controlador de MySQL ...");
        e.printStackTrace();
    }

    try {
        // Conectamos con la BD
        String url="jdbc:mysql://localhost/uoc";
        String usuario="yo"
        String contrasenia="contraseña"
        Connection conexion = DriverManager.getConnection (url,usuario,contrasenia);
    }
```

```

// Creamos una sentencia SQL
Statement sentenciaSQL = conexion.createStatement();
// Ejecutamos la sentencia
ResultSet res = sentencia.executeQuery("SELECT * FROM tabla");

// Iteramos sobre la hoja de resultados
while (res.next()) {
    // Obtenemos el campo 'nombre' en forma de String
    System.out.println(res.getString("nombre" ));
}

// Finalmente, se liberan los recursos utilizados.
res.close();
sentencia.close();
conexion.close();
} catch (SQLException e) {
System.out.println("Excepción del SQL: " + e.getMessage());
System.out.println("Estado del SQL: " + e.getSQLState());
System.out.println("Error del Proveedor: " + e.getErrorCode());
}
}

```

Mientras que la operación `executeQuery()` de la clase `Statement` devuelve un objeto `ResultSet`, la operación `executeUpdate()` sólo devuelve su éxito o fracaso. Las sentencias SQL que se utilizan con `executeUpdate()` son **insert**, **update**, o **delete**, porque no devuelven ningún resultado.

```

public void Insertar_persona(String nombre, direccion, telefono){
Statement sentencia = conexion.createStatement();
sentencia.executeUpdate( "insert into personas values("
+ nombre + ","
+ domicilio + ","
+ telefono + ")" );
}

```

Errores

Todos los errores de JDBC se informan a través de `SQLException`.

`SQLWarning` presenta las advertencias de acceso a las bases de datos.

2.2. Sentencias preparadas

Las sentencias preparadas de JDBC permiten la “precompilación” del código SQL antes de ser ejecutado, permitiendo consultas o actualizaciones más eficientes. En el momento de compilar la sentencia SQL, se analiza cuál es la estrategia adecuada según las tablas, las columnas, los índices y las condiciones de búsqueda implicados. Este proceso, obviamente, consume tiempo de procesador, pero al realizar la compilación una sola vez, se logra mejorar el rendimiento en siguientes consultas iguales con valores diferentes.

Otra ventaja de las sentencias preparadas es que permiten la parametrización: la sentencia SQL se escribe una vez, indicando las posiciones de los datos que van a cambiar y, cada vez que se utilice, le proporcionaremos los argumentos necesarios que serán sustituidos en los lugares correspondientes. Los parámetros se especifican con el carácter ‘?’.

```
public class Actualizacion{
    private PreparedStatement sentencia;

    public void prepararInsercion(){
        String sql = "insert into personas values ( ?, ? ,? )";
        sentencia = conexion.prepareStatement(sql);
    }

    public void insertarPersona(String nombre, dirección, telefono)
    {
        sentencia.setString(1, nombre);
        sentencia.setString(2, direccion);
        sentencia.setString(3, telefono);
        sentencia.executeUpdate();
    }
}
```

Al utilizar esta clase, obviamente, deberemos llamar primero al método que prepara la inserción, y posteriormente llamar tantas veces como sea necesario al método `insertarPersona`.

Se definen tres parámetros en la sentencia SQL, a los cuales se hace referencia mediante números enteros consecutivos:

```
String sql = "insert into personas values ( ?, ? ,? )";
```

La clase `PreparedStatement` incluye un conjunto de operaciones de la forma `setXXXX()`, donde `XXXX` es el tipo de dato para los campos de la tabla. Una de esas operaciones es precisamente `setString()` que inserta la variable en un campo de tipo cadena.

Ejemplo

El segundo de los tres parámetros se especifica con `sentencia.setString(2, direccion);`

2.3. Transacciones

La API JDBC incluye soporte para transacciones, de forma que se pueda deshacer un conjunto de operaciones relacionadas en caso necesario. Este comportamiento es responsabilidad de la clase `Connection`.

Por omisión, cada sentencia se ejecuta en el momento en que se solicita y no se puede deshacer. Podemos cambiar este comportamiento con la operación siguiente:

```
conexion.setAutoCommit(false);
```

Después de esta operación, es necesario llamar a `commit()` para que todas las sentencias SQL pendientes se hagan definitivas:


```
sentencia.executeUpdate(...);  
...  
sentencia.executeUpdate(...);  
...  
conexion.commit(); // Se hacen permanentes las dos actualizaciones anteriores
```

En caso contrario, desharemos todas las actualizaciones después del último
commit():

```
sentencia.executeUpdate(...);  
...  
sentencia.executeUpdate(...);  
...  
sentencia.executeUpdate(...);  
...  
conexion.rollback(); // Cancela las tres últimas actualizaciones
```

Resumen

Hemos presentado algunas de las formas más habituales de conectarse a los SGBD que hemos visto en módulos anteriores desde PHP y Java.

Hemos podido comprobar que no existen demasiadas variaciones ni restricciones entre MySQL y PostgreSQL en cuanto a su acceso desde lenguajes de programación, sino al contrario, los esfuerzos se encaminan en homogeneizar el desarrollo e independizarlo del SGBD con el que trabajamos.

En PHP, hemos repasado los métodos nativos y visto sus particularidades. Hemos comprobado que, a no ser que necesitemos características propias y muy avanzadas de un SGBD, no es aconsejable usar esos métodos por los problemas que nos puede ocasionar un cambio de gestor de base de datos en el futuro. Aun así, es interesante revisarlos porque encontraremos muchas aplicaciones de software libre desarrolladas en PHP que los utilizan.

PEAR::DB es un ejemplo de librería de abstracción (no es la única) bien hecha y con el soporte de la fundación que mantiene PHP. Tiene todo lo que podemos desear y actualmente es completamente estable y usable en entornos empresariales.

En Java, hemos visto JDBC. Aunque la API da mucho más de sí, creemos que hemos cumplido los objetivos de este capítulo, sin entrar en conceptos que sólo programadores expertos en Java podrían apreciar.

Así pues, se han proporcionado los elementos de referencia y los ejemplos necesarios para trabajar con bases de datos en nuestras aplicaciones.

Bibliografía

Documentación de PHP: <http://www.php.net/docs.php>

PEAR :: The PHP Extension and Application Repository: <http://pear.php.net/>

Pear::DB Database Abstraction Layer: <http://pear.php.net/package/DB>

DBC Technology: <http://java.sun.com/products/jdbc/>

MySQL Connector/J: <http://dev.mysql.com/downloads/connector/j>

PostgreSQL JDBC Driver: <http://jdbc.postgresql.org/>

