

Con forma de Pera

# Cobertura Total

Si se escribe código en un lenguaje de alto nivel, como PHP, se asegura su funcionalidad en cualquier plataforma. Entonces, ¿Por qué limitarlo a usar un único servidor de bases de datos? ¿Qué pasa con las plataformas que no soportan MySQL? **POR STEVEN GOODWIN**

La respuesta a estas preguntas está contenida en las palabras "API Genérica". Haciendo uso de una API genérica, nuestro código puede funcionar con diferentes servidores de bases de datos sin necesidad de tener que modificar los archivos de código fuente. Los desarrolladores de PHP son muy conscientes de esta ventaja y, en los últimos años se han dirigido esfuerzos a solventar la carencia de la que adolecía PHP en este área. Este mes, en nuestra sección dedicada a PHP, Steven Goodwin presenta *PEAR::DB*.

## Arte Abstracto

*PEAR::DB* es un módulo de PHP que permite el control de una base de datos sin necesidad de especificar ningún servidor de bases de datos en concreto. Esto significa que el mismo código se puede utilizar para acceder a MySQL y Oracle, por poner un ejemplo. Así pues, ¿cómo funciona? Gracias a la abstracción, que permite hacer una generalización de un sistema, o de una API, de modo que los detalles queden ocultos. La mayoría de nosotros usamos abstracciones (a menudo sin darnos cuenta de ello). Incluso los programadores de C, supuestamente los tipos más duros del mundo del desarrollo software, hacen que sus

vidas sean más fáciles gracias a las abstracciones. Cada función, expresión y sentencia abstrae algún detalle específico del hardware del procesador al programador por medio del lenguaje C. Esto se conoce como abstracción de bajo nivel.

Otro ejemplo de abstracción se produce cada vez que interactuamos con una base de datos a través de cualquier motor SQL. La programación de bases de datos por medio de SQL es una abstracción de alto nivel. La base de datos (ya sea MySQL, PostgreSQL u Oracle) puede funcionar de cualquier forma, ajustarse a cualquier algoritmo y hacer uso de cualquier archivo. Pero usando un lenguaje genérico (SQL) con el que comunicarnos, no necesitamos preocuparnos más por los aspectos específicos. Al contrario, podemos dedicar nuestro tiempo a tareas más importantes, como crear consultas *select* e *inner joins* eficientes.

Esta abstracción no se extiende a la forma en que programamos las bases de datos, ya que cada una de ellas tiene su propia API. Esto se ve en PHP, donde se puede empezar un diálogo con una base de datos MySQL llamada *fredbloggs* usando,

```
$db =
mysql_connect("localhost",
"myuser", "mypass");
mysql_select_db("fredbloggs");
```

mientras que PostgreSQL necesita,

```
$db =
pg_connect("host=localhost
dbname=fredbloggs user=myuser
password=mypass");
```



Aunque la mayoría de estos parámetros son opcionales, la migración de código entre bases de datos es aún más ardua. Toda conexión, consulta y manejo de errores debe ser reescrito para ajustarse a la nueva base de datos cada vez que se desee utilizar un sistema de almacenamiento y su correspondiente motor diferentes. Esto no sólo supone una modificación en lo que se refiere al nombre de la función sino que también cambia la estructura y el formato de los argumentos. Otro problema es que los códigos de error devueltos por el motor varían de sistema a sistema, lo que hace que sean difíciles de interpretar. La solución a todos estos dilemas, ya la hemos mencionado, consiste en abstraer la función *database\_connect* de la base de datos específica y usar una llamada a una API genérica. Esto debe hacerse para cada función específica de la base de datos. Desafortunadamente, si pretendemos crear la API nosotros mismos, esta tarea supone mucho trabajo

Pero, al fin y al cabo, estamos hablando de código libre, con una comunidad de desarrolladores muy activa, y las buenas noticias son que alguien ya lo ha hecho por nosotros. Hay un par de APIs de bases de datos para PHP. El objetivo de este artículo es *PEAR::DB*, uno de los muchos módulos PEAR (ver cuadro Acerca de PEAR) disponible en [1]. Está bajo desarrollo constante e incluye a algunos miembros del equipo del núcleo

## Acerca de Pear

PEAR son las siglas de PHP Extension and Application Repository (aunque algunos prefieren sustituir Application por Add-on) y es una biblioteca de funciones de código PHP, similar a CPAN de Perl. Además del módulo para el manejo de bases de datos (DB), también hay código para el manejo de HTML, autenticación y criptografía.

de PHP. La versión actual considerada estable es la 1.6.0. Para aquellos que quieran comparar alternativas están también ADOdb [2], Metabase [3] y PHPLib [4].

La adopción de MySQL ha sido, a la vez que una bendición, una maldición para los desarrolladores de PHP. En el lado positivo podemos decir que se integra bien con la instalación por defecto, significando que cualquiera puede desarrollar buenos sitios webs con bases de datos con el mínimo esfuerzo. Desafortunadamente, esto lleva a crear a muchos desarrolladores que no hay otros sistemas de bases de datos. O que no están bien soportadas. Nada más lejos de la realidad, como puede verse en la tabla 2, *Bases de datos soportadas*. Para mayor información se puede consultar el archivo *docs/STATUS*.

En este artículo utilizaremos PEAR::DB con MySQL para administrar una base de datos que contenga información sobre cadenas de televisión y su sintonía. Esto podrías servir, por ejemplo, para controlar programáticamente una tarjeta de televisión, permitiendo que se sintonizase cada canal a través de una aplicación.

## Manos a la obra

Pondremos las entradas (emisoras, canales y nombres) en una base de datos. De modo que pueda ser usado como parte de una aplicación para el control de una cadena de TV más grande, ver Listado 1.

Para empezar, hay que tomar este *sql-dump* e introducirlo en la base de datos de la forma habitual, permitiendo el acceso al usuario apropiado (*www-data*, por ejemplo). Esta base de datos puede ser accedida por MySQL con el Listado 2.

En esta sección de código (de la que se ha extraído el manejo de errores en aras

de la claridad) nos encontramos con no menos de 5 referencias separadas a MySQL. En vez de referenciar la tabla con *tv.channels*, algunos programadores prefieren especificar una base de datos por defecto usando:

```
mysql_select_db("tv");
```

Esta función es equivalente al comando *use* en el *prompt* de MySQL, pero añade otra llamada específica a MySQL.

En el momento de cambiar a un nuevo servidor de base de datos, nos encontramos con que cada referencia a *mysql* tendría que ser reescrita. Con más funciones, manipulando más bases de datos, la cantidad de código redundante crece de manera exponencial hasta el punto de ser inmanejable. Normalmente la única concesión a la hora de mantener el código al migrar a otro host consiste en un archivo independiente con el nombre del host, el usuario y la contraseña, por ejemplo, podemos tener un fichero como *dbase.inc*:

```
$dbhost = 'localhost';
$dbuser = 'www-data';
$dbpass = '';
$dbname = 'tv';
```

Pero normalmente esto suele ser muy insuficiente. Parte de lo anterior, necesitamos una capa de abstracción, como *PEAR::DB*. La mayoría de las instalaciones PHP incluyen la biblioteca *PEAR::DB* por defecto, y se puede encontrar los módulos que lo componen en */usr/share/pear*. Para confirmar que está instalado en el sistema y que se halla en la ruta de acceso a fichero de inclusión de PHP, podemos crear un fichero *test-pear.php* con el siguiente contenido:

```
<?php require_once 'DB.php'; ?>
```

Si lo anterior no funciona, se puede instalar o bien usando el *PEAR Packet Manager* (tecleando *pear install DB*), o manualmente con un *tarball*. Para más detalles de la instalación de

## DSN as array

```
01 $dsn = array(
02 'phptype' => "mysql",
03 'hostspec' => "localhost",
04 'database' => "tv",
05 'username' => "www-data",
06 'password' => ""
07 );
08 $db = DB::connect($dsn);
```

PEAR el manual on-line en [6] contiene toda la información necesaria para llevar a cabo la instalación y configuración del paquete. Otra opción consiste en copiar los archivos en el directorio *home* (*~/pear/*) y modificar el path de PHP. Esta alternativa será necesaria cuando el usuario no tenga los permisos de root en la máquina, como es el caso de muchos servicios de *hosting*. Por ejemplo:

```
<?php ini_set('include_path',
'~/pear/lib'.PATH_SEPARATOR.'ini'
_get('include_path'));?>
```

Ahora se dispondrá de acceso a nuevas funciones de bases de datos, todas ellas de acuerdo con las convenciones de nombrado de PEAR. ¿Dónde empezar? El lugar obvio es por las funciones básicas, *connect* y *close*. Habiéndole echado un vistazo previo a las versiones de MySQL y PostgreSQL, para empezar a utilizar *PEAR::DB* se requiere que el desarrollador adopte un ligero cambio de perspectiva.

Por ejemplo, para bases de datos que precisan más (o menos) parámetros que las dos mencionadas, una función simple que sustituya los parámetros no funcionará. En vez de esto, debemos especificar un Nombre de Fuente de Datos o DSN. Esto incluye todos los posibles argumentos dentro de una cadena única con formato. El formato completo es:

```
phptype(dbsyntax)://username:
password@protocol+hostspec/
databasename
```

El DSN tiene el aspecto de una URL. Describe donde conectarse, como conectarse y que base de datos y opciones utilizar una vez establecida la conexión. Estas líneas recogen dos partes. La primera parte describe los parámetros

**Tabla 1: Opciones de portabilidad**

Opción	Descripción
DB_PORTABILITY_LOWERCASE	Convierte los nombres de campos y tablas a minúsculas (en los <i>fetch</i> y <i>get</i> )
DB_PORTABILITY_RTRIM	Elimina los espacios en blanco de la derecha
DB_PORTABILITY_DELETE_COUNT	Informa siempre del número de filas borradas
DB_PORTABILITY_NUMROWS	Una copia del <i>numRows</i> de Oracle
DB_PORTABILITY_ERRORS	Convierte los números de error entre distintas bases de datos
DB_PORTABILITY_NULL_TO_EMPTY	Convierte los nulls en cadenas vacías (en los <i>fetch</i> y <i>get</i> ) porque Oracle no los diferencia

específicos del servidor de base de datos e incluye el tipo de base de datos (llamado *phptype*, ej. *mysql*) y cualquier opción específica dada por *dbsyntax*. Una lista de *phptypes* se muestra en la Tabla 2. El ejemplo citado de *dbsyntax* es el nombre de un driver específico cuando se usa una conexión ODBC (ej. *access*, *db2*, *mssql*). Esto no es difícil de determinar, pero afecta más a los usuarios de Windows que a nosotros, por ello no tenemos que profundizar más en este tema.

La segunda parte del DSN contiene todo aquello que es dependiente de la base de datos, como el nombre del host, el puerto, el nombre del usuario y la contraseña. Como en la típica función *mysql\_connect*, no todos los parámetros son obligatorios y pueden ser omitidos si no son necesarios. Por ejemplo:

```
mysql://www-data@localhost/tv
```

Naturalmente, nuestro código final almacenará todos estos parámetros en un archivo genérico *dbase.inc*, como se mostró antes. El DSN no tiene por qué ser especificado por una cadena. También puede ser definido por un array (como se muestra en el cuadro *DSN como Array*), haciendo ligeramente más rápida la inicialización ya que no se ha de analizar la cadena.

El DSN nos permite especificar las opciones de inicialización usando un método inspirado en una URL *?opcion1=valor1&opcion2=valor2*.

Hay varias opciones disponibles, incluyendo tanto características prácticas de conexión (uso de SSL) como ayu-

**Tabla 2: Bases de Datos soportadas**

Nombre	Palabra reservada
dBase	dbase
FrontBase	fbsql
InterBase	ibase
Informix	ifx
Mini SQL	msql
Microsoft SQL Server	mssql
MySQL	mysql
MySQL >=4.1	mysql4
Oracle 7/8/9	oci8
ODBC	odbc
PostgreSQL	pgsql
SQLite	sqlite
Sybase	sybase

dantes de desarrollo (para controlar la cantidad de mensajes de depuración producidos). Como estas opciones pueden variar entre consultas específicas, no las incluiremos dentro del DSN. Por el contrario crearemos un array detallando las opciones y lo pasaremos como argumento a la función *DB::connect* de forma separada.

```
// Recordar que estas variables
// han de ser declaradas de forma
global
$dbsn = "$dbbackend://$dbuser
@$dbhost/$dbname";
$options = array('debug' => 2);
$db =& DB::connect($dbsn,
$options);
```

Tenemos la posibilidad de realizar modificaciones en estos parámetros en cualquier momento con la utilización de la siguiente función:

```
$db->setOption('debug', 0);
```

En el caso de que la conexión no falle por algún motivo (echaremos un vistazo a las posibilidades del manejo de errores más adelante), tendremos un objeto de base de datos llamado *\$db* que se utilizará en todas las llamadas a esta base de datos en concreto. Por ejemplo, para cerrar la conexión después de su uso, haremos:

```
$db->disconnect();
```

**Tabla 3: Soporte de provides**

Cadena	Funcionalidad
prepare	Pre-comprueba la consulta SQL
pconnect	Conexiones persistentes
transactions	Activa el soporte de transacciones
limit	Limita las consultas select

**Tabla 4: Diferencias entre versiones de SQL**

Base de Datos	Sintaxis SQL
DB2	select * from table fetch first 10 rows only
Informix	select first 10 * from table
Microsoft SQL Server	select top 10 * from table
MySQL	select * from table limit 10
Oracle 8i	select * from (select * from table) where rownum <= 10
PostgreSQL	select * from table limit 10

Entonces, tendremos que adaptar nuestras funciones existentes para usar el nuevo objeto y sus funciones miembros asociadas. Esto no es difícil, ya que tienen nombres muy parecidas a las versiones originales de MySQL pero siguiendo las convenciones de nombres de PEAR. Así pues, *mysql\_query* se convierte en *query*, por poner un ejemplo. Las funciones básicas aparecen en el Listado 3.

## Al grano

Como todas las peticiones a la base de datos pasan por el controlador *PEAR::DB*, el código tiene que tener la habilidad de cambiar y modificarse en las peticiones. Esto se hace en aras de una mayor portabilidad. Y podemos darle más o menos opciones haciendo uso del método *setOption* que ya hemos visto con anterioridad. Este método permite establecer un número diferente de opciones conforme a las necesidades que precisemos.

Estas opciones normalmente están para facilitar la portabilidad a costa del rendimiento, y dependen de una manera muy específica de cada aplicación. Otro uso al que se pueden aplicar es para permitir la coexistencia de código antiguo dentro de *PEAR::DB*. Dentro del desarrollo con bases de datos, la convención dicat que la mayoría de los nombres de tablas aparecen en minúsculas. Si una aplicación está intentando obtener filas usando una mezcla de mayúsculas y minúsculas, por ejemplo, entonces estos nombres se convertirán de forma automática a minúsculas.

```
$db->setOption('portability',
DB_PORTABILITY_LOWERCASE);
```

Esto elimina las sorpresas que pueden surgir cuando un trozo de código desconocido se ejecuta y da un error de ejecución. Otras opciones pueden verse en la Tabla 1.

Los valores por defecto para estas opciones tienen como objetivo mejorar el rendimiento, pero está a la elección de uno determinar cuales se deben establecer en la aplicación. Hay también definiciones para *DB\_PORTABILITY\_ALL* y *DB\_PORTABILITY\_NONE* que permiten el activar todas las opciones o desactivarlas respectivamente.

## Sigamos adelante

No toda la funcionalidad está dirigida a que el acceso a la base de datos sea más fácil. *fetchRow*, por ejemplo, hace más fácil obtener datos en un formato programático más amigable. Actualmente *PEAR\_DB* soporta tres formatos de este tipo. Por defecto se usa un array, indexado desde cero, como se muestra en el cuadro adjunto. Los parámetros opcionales de *DB\_FETCHMODE\_ORDERED* han sido omitidos en el ejemplo anterior. Esto es útil para manejar bases de datos genéricas o para mostrar tablas sin la necesidad de saber el nombre de sus campos.

En la mayoría de las situaciones, sin embargo, los índices numéricos no son lo suficientemente descriptivos y tenemos que solicitar que los datos nos sean devueltos en un array asociativo. De esta forma los resultados aparecen en el código de una forma más legible a costa de sacrificar generalidad.

```
while($row = $result->fetchRow(
    DB_FETCHMODE_ASSOC)) {
    print $row['station']."
-
    ".$row['name']." (".$row[
    'channel'].")<br>"; }
```

Por último, *fetchRow* proporciona un medio para usar las características de la orientación a objetos de PHP para devolver un objeto por cada fila de la tabla de resultados. Cada columna está etiquetada como un atributo. De nuevo, esto hace que el código sea más fácil de leer, pero hace que la aplicación sea menos genérica.

```
while($row = $result->fetchRow(
    DB_FETCHMODE_OBJECT)) {
    print $row->station." -
    ".$row->name." (".$row->
    channel.")<br>"; }
```

## Allanando el camino

Ningún programa está terminado sin un control de manejo de errores y la documentación correspondiente. Ninguno de los dos son atractivos desde el punto de vista del programador pero son muy necesarios. Las opciones de manejo de errores de *PEAR::DB* han sido unificadas (como los códigos de error) y hacen uso de las capacidades básicas de manejo de

errores de *PEAR\_Error*. Pero si una función de *PEAR::DB* produce un fallo, la mayoría devolverá una instancia de la clase error, desde *connect* hasta *getRow*. Esta clase no solo controla los errores de *PEAR*, sino que también proporciona información extra al depurador, lo cual es útil a la hora de detectar problemas.

```
$db=& DB::connect (&dsn);
//DB::Error es lo mismo que
PEAR::isError
if (DB::isError($db))
{
    print $db->getMessage();
    print $db->getDebugInfo();
}
```

Los errores también pueden ser capturados usando el manejo de errores estándar de *PEAR*. Esto se hace con unas funciones definidas por el usuario que serán llamadas cada vez que un módulo de *PEAR* (como *DB*) genere un error. Esta función puede usarse para generar una página HTML estándar para el usuario, al mismo tiempo que también puede alertar al administrador del problema.

El manejo de errores tradicionalmente está combinado con las características de escritura en el buffer de salida de PHP, permitiendo a cualquier página parcialmente generada sea eliminada del flujo de salida HTML. Como esto es una característica de *PEAR*, no de PHP, los errores tradicionales (como la división por cero) no serán capturados por este método.

```
01 // Preparación del handler
02 PEAR::setErrorHandling (
    PEAR_ERROR_CALLBACK,
    'error_function');
03 //Activamos el buffer de
    salida
04 ob_start();
05 // Realizamos alguna tarea
06 PearVersion();
07 // Volcamos el buffer a
    la salida (si todo ha ido
    bien)
08 ob_end_flush();
09 // Preparamos nuestro
    handler
10 function
    error_function($err)
11 {
```

```
12 ob_end_clean();
13 print "Se ha producido un
    ERROR (".$err->
    getMessage().")";
14 exit;
15 }
```

## Rizando el rizo

*PEAR::DB* realmente es tan fácil como parece. La complejidad viene del SQL. *PEAR::DB* sólo sirve para protegerte en cierta medida de esto. SQL existe desde hace muchos años. Debido a las guerras comerciales entre distintos vendedores, han surgido distintas variantes de SQL que son incompatibles entre sí.

Aunque ANSI ha intentado estandarizar algunas partes del lenguaje (las versiones básicas de *select*, *insert* y *update*, las cuales son bastante portables), existen aún muchas diferencias. Escribir SQL estándar es una tarea árdua en sí misma y hay que saber y adoptar varias reglas. La mayoría de los programadores expertos en bases de datos las conocen de forma instintiva, el resto de los mortales tienen que aprenderlas de tutoriales tales como [7].

En algunos casos, podríamos desear usar diferentes consultas dependiendo de como reaccione la base de datos. Esto requiere trabajo extra ya sea de nosotros o de *PEAR::DB*. Desafortunadamente, una vez que hemos sacado la base de datos de la ecuación, no tenemos forma alguna de saber si es capaz de hacer lo que pretendemos. *PEAR::DB* tiene conocimiento de este problema y utiliza un método llamado *provides*.

*provides* indica las características propias del servidor de base de datos que se esté usando, nos permite cambiar entre dos consultas 'afinadas a mano' para ayudarnos a mejorar el rendimiento en casos especiales. Usamos las capacidades (otra abstracción) en vez de funciones específicas de la base de datos porque las cosas cambian. Por ejemplo, una versión posterior de la base de datos podría soportar nuevas características o podría aparecer un nuevo tipo de servidor en el mercado. El método *PEAR::DB provides* suministra un medio para usar la consulta más óptima en la base de datos sin la necesidad de entender nada acerca de las nuevas bases de datos.

```
if ($db->provides
```

### Listado 1: La Base de Datos TV

```
01 CREATE DATABASE IF NOT EXISTS tv;
02 USE tv;
03 drop table IF EXISTS channels;
04 CREATE TABLE channels (
05 station smallint(2) NOT NULL default '0',
06 channel smallint(2) default NULL,
07 name varchar(10) default NULL,
08 PRIMARY KEY (station)
09 ) TYPE=MyISAM;
10 INSERT INTO channel VALUES (1,55,'TVE1');
11 INSERT INTO channel VALUES (2,62,'La 2');
12 INSERT INTO channel VALUES (3,59,'Telecinco');
13 INSERT INTO channel VALUES (4,65,'Antena 3');
14 INSERT INTO channel VALUES (5,37,'Canal +');
```

```
('transactions'))
print "OK! Están soportadas";
```

El abanico de características para las que podemos usar *provides* se muestran en la Tabla 3.

En cada caso es posible que la base de datos no soporte de forma nativa la característica. La palabra clave aquí es “nativa”, porque hay que diferenciar entre el servidor de la base de datos y el driver *PEAR::DB*. Por ejemplo, si la base de datos no soporta comandos *prepare/execute*, el driver soportará la apariencia del comando de forma simulada.

Se puede estar tentado a usar *provides* para crear consultas completamente diferentes para cada base de datos de forma manual. En la mayoría de los casos esto es innecesario y desde luego es una mala idea. La razón (algunos dirían excusa) para este comportamiento viene de las extensiones que aparecen en SQL. El ejemplo típico de este problema viene de las consultas *select* limitadas, que paran de producir resultados después de las primeras, digamos, 10 filas. La mayoría de las bases de datos actuales son capaces de tener este comportamiento, pero con cadenas de consultas SQL diferentes, como se muestra en la tabla 4. Codificar cada ejemplo

explícitamente provocaría un montón de trabajo extra.

También, como no podemos probar cada base de datos (incluidas las nuevas, incluso las no escritas aún) nuestro código se volvería no-portable de forma muy rápida. Este problema es fácil de resolver, sin embargo, empleando el mismo principio de abstracción. *PEAR::DB* proporciona un método llamado *limitQuery* que oculta la sintaxis precisa al usuario final y adapta de forma adecuada la consulta al servidor. Esto obviamente tiene más sentido que escribir consultas separadas para cada una de las bases de datos por nosotros mismos.

```
$query = "SELECT name FROM
channels";// sin referencias
a límites aquí!
$result = limitQuery
($query, 2, 1);
```

Este comando extrae una línea de resultado de la consulta, empezando por el

### Listado 2: Acceso a MySQL

```
01 function GetStationsList()
02 {
03 $db = mysql_connect("localhost","www-data","");
04 $query = "SELECT * FROM tv.channels";
05 $result = mysql_query($query);
06 while ($row =
mysql_fetch_array($result,MYSQL_NUM)) {
07 print "$row[0] - $row[2]($row[1])<br>";
08 }
09 mysql_free_result($result);
10 mysql_close($db);
11 }
```

### Listado 3: Convenciones para los nombres PEAR

```
01 function PearVersion()
02 {
03 global $dbname, $dbhost, $dbuser, $dbbackend;
04 $dsn = "$dbbackend://$dbuser@$dbhost/$dbname";
05 $db =& DB::connect($dsn);
06 $query = 'SELECT * FROM channels';
07 $result = $db->query($query);
08 while ($row = $result->fetchRow()) {
09 print "$row[0] - $row[2]($row[1])<br>";
10 }
11 $result->free();
12 $db->disconnect();
13 }
```

índice 2. Como empezamos a contar desde 0, esto significa la tercera entrada.

Si la consulta SQL *select* puede ser modificada para crear una cadena adecuada para la base de datos en cuestión, *\$db->provides("limit")*; devolverá *alter*, y la consulta será modificada por el controlador de *PEAR::DB* antes de ser pasada al servidor. Por otro lado, *provides* puede devolver *emulate*, porque el driver es capaz de capturar los resultados de las consultas fila a fila. O devolver *false*. Se debería siempre preguntar por las características de las bases de datos usando *provides* en vez de confiar en nuestra experiencia o en la memoria. Sin embargo, por comparación, el conjunto de controladores actuales proporcionan la funcionalidad que muestra en la Tabla 5.

En algunos casos raros, es necesario saber la base de datos que se está utilizando de forma precisa. Normalmente debido a bugs conocidos en la base de datos misma. Son un hecho de vida. Pero si no podemos eliminarlos, tendremos que saber donde están para al menos poder evitarlos.

```
print $db->phptype;
```

Utiliza los mismos identificadores que se muestran en la tabla 2. Los detalles referentes a los bugs conocidos en bases de datos concretas, van más allá del objetivo de este artículo.

### Sólo para Ti

Como complemento especial, *PEAR::DB* no sólo proporciona abstracciones de bases de datos. También incluye algunas

### Example Interface Files

```
01 $tableinfo =
    $result->tableInfo();
02 for($col=0; $col <
    $result->numCols(); $col++) {
03 print
    $tableinfo[$col]['name']." es
    un ".$tableinfo[$col]['type'];
04 }
```

ayudas para el manejo de datos, como son *numRows* y *numCols*.

```
print "La consulta select
produjo ".$result->numRows()
. " filas";
print "La consulta select
produjo ".$result->numCols()
. " columnas";
```

Los resultados de estas consultas son bastante autoexplicativas, y pueden ser deducidas de la propia consulta *select*. Cuando la consulta es generada automáticamente, podemos evitar tener que contar las columnas manualmente. Podríamos usar estos valores para iterar sobre los resultados.

Hay también un método llamado *table-*

### Listado 5 : assertExtension

```
01
    if(DB::assertExtension("oci8")
    )
02 print "Usamos Oracle si
    tenemos que...";
03 if
    (DB::assertExtension("mysql"))
04 print "Usamos MySQL porque lo
    entendemos mejor...";
```

### Listado 6: Preparando una consulta

```
01 $generic =
    $db->prepare("INSERT INTO
    channels (station, name,
    channel) VALUES (?, ?, ?)");
02 $data = array (
03 array(6, "Video", 0),
04 array(7, "PS2", 39)
05 );
06 $res =
    $db->executeMultiple($generic,
    $data);
```

*Info* que proporciona información sobre cada columna del resultado, como es su nombre y tipo. Esto no es sólo útil mientras estamos depurando el programa, sino incluso para crear aplicaciones de bases de datos generalizadas. Podemos colorear cada columna según su tipo, o resaltar algún campo clave. Para ver como funciona esto, ver listado 4.

Además del nombre y del tipo, se puede consultar la longitud de cada columna, flags (que indiquen si son claves primarias) y el nombre de la tabla, usando la sintaxis apropiada.

Aunque la mayoría de las API soportan las consultas *select*, hay aún algunos problemillas a solventar con otros tipos de consultas como los *insert*. Un método interesante es *affectedRows*. Este método, como podrá se adivinar, devuelve el número de filas que han sido afectadas por una consulta *insert*, *delete* o *update*. Hay también soporte para generar IDs únicos usando funciones de secuencia, *nextID*, que es útil para generar claves únicas, algo no soportado de forma nativa en MySQL. Por ejemplo:

```
// Obtención de un nuevo ID.
$id = $db->nextID('sequence');
// La secuencia será creada de
no existir
```

Otra característica útil es el método de clase (declarado *static*) *assertExtension*. Este método, siendo de clase, no requiere de una instancia de la clase base de datos para ser invocado e indicará que características están incluidas en la instalación actual. En su forma más útil, se puede determinar que bases de datos están instaladas en el sistema y seleccionar la más adecuada para usarla en la aplicación. Ver Listado 5.

A pesar de su nombre, esto no es una aserción tal y como se suele entender tradicionalmente en programación, ya que no imprime ningún error y no tiene que cumplirse ninguna condición. Simplemente indica si una extensión existe o no.

Antes de terminar, mencionaremos de pasada la pareja *prepare* y *execute*. *prepare* prepara (valga la redundancia) una consulta para una ejecución repetida precompilándola en tokens. La consulta estará preparada para admitir distinta información en unas variables espe-

ciales, de manera que se pueda ejecutar múltiples veces con los datos diferentes contenidos en dichas variables. Ver listado 6.

El método *executeMultiple* se evalúa a la consulta SQL,

```
INSERT INTO channels (station,
name, channel) VALUES (6,
"Video",
0);
INSERT INTO channels (station,
name, channel) VALUES (7,
"PS2", 39);
```

*executeMultiple* se parará al primer error que ocurra. Para evitar esto, se necesitará ejecutar cada consulta cada vez con el método *execute*. Por ejemplo,

```
foreach ($data as $row) {
    $db->execute($generic, $row);
}
```

Conviene siempre recordar que el array de datos empieza a indexarse desde cero. En la mayoría de los casos se podrá ver una mejora en la velocidad cuando varias entradas se introduzcan en la base de datos de una vez. Este beneficio presupone que la base de datos soporta esta característica de forma nativa, algo que no siempre sucede.

### Conclusión

Con PEAR::DB en el arsenal, se pueden atacar la mayoría de las bases de datos, sin tener que preocuparse de características no soportadas o de funciones no disponibles. Usando las funciones y el amplio rango de métodos disponibles, podemos escribir en PHP buenas aplicaciones para múltiples plataformas sin preocuparnos de las minucias de la batalla y así pues, podremos concentrarnos en pensar en las estrategias para ganar la guerra. ■

### INFO

- [1] PEAR: <http://www.pear.php.net> [2] ADOdb: <http://php.weblogs.com/adodb>
- [3] Metabase: <http://freshmeat.net/projects/metabase/> [4] PHPLib: <http://phplib.sourceforge.net/> [6] Manula de PEAR: <http://www.pear.php.net/manual/en/installation.ci.php> [7] Como Escribir Código SQL Portable: [http://php.weblogs.com/portable\\_sql](http://php.weblogs.com/portable_sql)