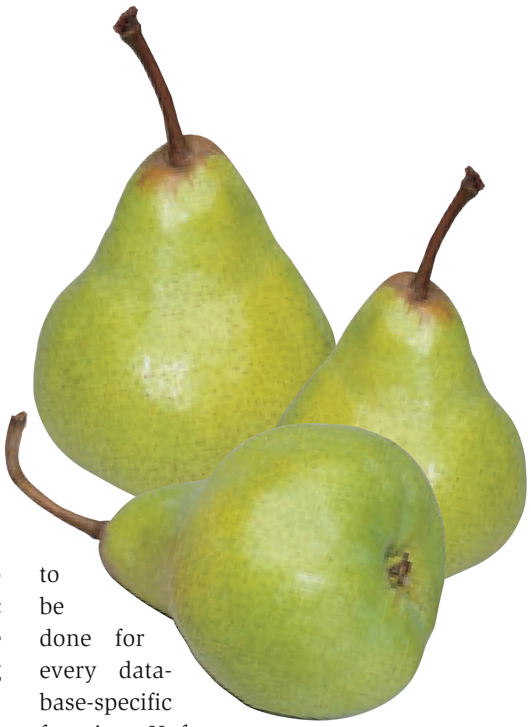
Going Pear-Shaped

Global Cover

Writing code in a high level language, such as PHP, ensures that it has a global audience. So why then limit that audience by imposing a specific choice of database? What about platforms that do not support MySQL?

BY STEVEN GOODWIN



By using a common API, our code can work with different databases without the trials of having to modify the source. This month, Steven Goodwin gets a bigger audience by using PEAR::DB.

Hey Matthew

PEAR::DB is a PHP module that provides control over a database, but without requiring any specific database. This means that the same code can be used to access MySQL and Oracle, for example. So how does it work? Abstraction! This is ability to generalize a system, or API, so that the details are hidden from view. Most of us use abstractions – often without realizing it. Even 'C' programmers, the supposedly hardened criminals of the development world, have their life made easier through abstractions. Every function, expression and statement abstracts specific hardware details of the processor away from the programmer through the 'C' language. This is a low level abstraction.

Database programming through SQL is a high level abstraction. The database (be it MySQL, PostgreSQL or Oracle) can work in any way it chooses, feature any algorithm, and use any files it likes. By choosing a common language (SQL) with which to communicate, we no longer have to worry about those specific details. Instead, we can devote our time to more important tasks, like creating efficient *inner joins* and *select* queries!

This abstraction doesn't extend to the way in which we program the database, as each has its own separate API. In PHP, you can initiate dialog with a MySQL database called *fredbloggs* using,

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

while PostgreSQL requires,

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

Although most of these parameters are optional, moving code between databases is still painful, as every connect, select and error handler needs to be re-written to accommodate the new database. Every time they're used. Not just a change to the function name, but also the structure and format of the arguments. As well as the error codes. The answer is to abstract the specific *database_connect* function away from a specific database, and use a common API call. This needs to be done for every database-specific function. Unfortunately, this is a lot of work!

The good news is that someone else has done it for us! There are a couple of database APIs for PHP. The focus of this article is on PEAR::DB, one of the many good PEAR modules (see box – About PEAR) available from [1]. It is under active development, and includes some of the core PHP team. The current version is 1.6.0, and considered stable. For those wanting to compare alternatives, there is also ADOdb [2], Metabase [3] and PHPlib [4].

The adoption of MySQL has been a blessing and a curse for the PHP developer. On the positive side, it integrates well with the default install and means that anyone can develop good database-driven web sites, with the minimum of

About Pear

PEAR stands for the PHP Extension and Application Repository (although some people seem intent on replacing *Application* with *Add-on*) and is a library for PHP code, similar to CPAN for Perl. Alongside the database (i.e. DB) modules, there is also code for handling HTML, authentications and encryption.

Listing 1: TV database

```
CREATE DATABASE IF NOT EXISTS tv;
USE tv;
drop table IF EXISTS channels;
CREATE TABLE channels (
  station smallint(2) NOT NULL default '0',
  channel smallint(2) default NULL,
  name varchar(10) default NULL,
  PRIMARY KEY  (station)
) TYPE=MyISAM;
INSERT INTO channel VALUES (1,55,'BBC 1');
INSERT INTO channel VALUES (2,62,'BBC 2');
INSERT INTO channel VALUES (3,59,'ITV');
INSERT INTO channel VALUES (4,65,'Channel 4');
INSERT INTO channel VALUES (5,37,'Five');
```

hassle. Unfortunately, it also blinkers many developers into believing that there are no other databases. Or that they are not well supported. Neither is true, as can be seen in Table 2, supported databases. Additional status information can be found in the file *docs/STATUS*.

With these considerations in mind, this article will follow the most common upgrade path to PEAR::DB, which is from MySQL, and look at the simple case of a TV channel database, on a PHP-generated web page.

Missionary Man

Since the utilization of PEAR::DB is, in essence, a software upgrade, we need to begin with an existing application. To fulfill that role we'll return to the theme of last month's article, and our email to video gateway [5]. Instead of storing the channels explicitly in a batch file, we'll place the mapping entries (stations, channels and names) into a database. This could be used as part of a larger TV control application, see Listing 1.

To start, take this sqldump and apply to the database in the usual way, granting access to the appropriate user (*www-data*, for example). This database can then be accessed through MySQL with Listing 2.

In this rather simple section of code (which has had all its error checking removed for clarity) we have no less than 5 separate references to MySQL. Instead of referencing the table with *tv.channels*, some people prefer to specify a default database using,

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

It works in the same way as the *use* command at the MySQL command prompt, but adds another specific MySQL call.

If we changed the backend database, every *mysql* reference would need to be re-written. With more functions, handling more databases, the amount of redundant code will grow. Normally, the only concession to maintenance coding involves moving the database host, user name and password into a separate file, like *dbase.inc*.

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

Table 2: Supported databases

Name	Keyword
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

That is not enough. We need an abstraction layer, like PEAR::DB. Most installations will include the PEAR::DB library by default, usually in */usr/share/pear*. You can confirm that a system-wide installation is included on your machine by writing,

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

If not, you can install it using either the PEAR Packet Manager (by typing *pear install DB*), or a manually with a tarball. Further details on the PEAR installation process can be found in the on-line manual, located at [6]. Alternatively, copy the files into your home directory (say *~/pear/*), and amend the PHP include path. This will be necessary where you do not have root on the machine, as is the case with many hosting packages. For example,

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

You will now have access to a host of new database functions, all conforming to PEAR's naming convention. Where to start? The obvious place are the basic house keeping functions, *connect* and *close*. Having previously looked at the MySQL and PostgreSQL versions, the PEAR::DB ones require a slight change in perspective.

Since other databases may require more (or less) parameters, a simple replacement function that renames the parameters will not work. Instead, we must specify a *data source name*. Or DSN. This encompasses all possible arguments into a single formatted string. The complete format of which is,

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

The DSN looks like a URL. It describes where to connect, how to connect, and what database and options to use once we have connected. This line comprises two parts. The first part describes the specifics of the database back-end, and includes the database type (named *phptype*, e.g. *mysql*), and any database-specific requirements given by *dbsyntax*. A list of *phptypes* is shown in Table 2. The oft-cited example of a *dbsyntax* string is the name of a specific driver when using an ODBC back-end (e.g. access, db2, mssql). This is not difficult to determine, but affects Windows users more than us, so we do not need to go any further.

The second part of the DSN contains everything that is database independent, such as hostname, port, username and password. As with the usual *mysql_connect* function, not all parameters are obligatory and can be omitted as necessary. For example:

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

Naturally, our final code will store these parameters in a common *dbase.inc* file for unity, as previously shown. The DSN need not be specified as a string. It can also be given as an array (as detailed in the DSN as an Array box), making it slightly faster to initialize since there is no string to parse.

The DSN allows us to specify initialization options using the URL-inspired *?option1=value1&option2=value2* method. There are several options available, including both practical connection based features (the use of SSL) and development helpers (to control the amount of debugging messages produced). Since these options can vary between specific queries, we shall not incorporate them into the DSN. Instead, we shall create an array detailing the options, and pass them to the *DB::connect* function separately.

```
// Remember these variables may need to be declared global
$dsn = "$dbbackend://$dbuser@$dbhost/$dbname";
$options = array('debug' => 2);
$db =& DB::connect($dsn, $options);
```

These options can be changed at any time, using the following function.

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

Should the connection work (we'll look at the error handling capabilities later), we will have a database object called *$db* that is used in all other calls to this particular database. For instance, closing the database after use.

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

We then have to set about adapting our existing functions to use the new object and its associated member functions. This is not difficult, since they have very similar names to the original MySQL versions and follow the PEAR naming convention. So *mysql_query* becomes *query*, for example. This makes our basic function as in Listing 3.

Glory Box

Because all requests to the database go through the PEAR::DB driver, the code within it has the ability to change, modify, and mess around with your request. It does this in the name of portability. You can give it more, or less, reason to do this with the portability options provided through the *setOption* method that we've already seen. This allows you to set up a number of different options so you can make informed decisions as to the trade-offs you want to allow.

These trade-offs usually exist on a performance versus portability playing field, and depend largely on your specific application. They can also be used to coax older code into PEAR::DB. By convention, most table names will be in lowercase. If an application is trying to fetch rows using mixed case, for example, then these names will be automatically converted to lowercase by,

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

This removes the surprise that can occur when an unknown code path gets executed and breaks the application. Other options are given in Table 1.

The default settings for all these options are aimed at improving performance, but it is up to you to determine which ones are set for your application. There are also definitions for DB_PORTABILITY_ALL and DB_PORTABILITY_NONE that fulfill their traditionally useful role of setting all, or none, of the flags respectively.

DSN as an Array

```
$dsn = array(
    'phptype'  => "mysql",
    'hostspec' => "localhost",
    'database' => "tv",
    'username' => "www-data",
    'password' => ""
);
$db = DB::connect($dsn);
```

Move the Crowd

Not all of the functionality is geared towards making the database easier to access. *fetchRow*, for example, makes it easier to retrieve the data in a programmer-friendly format. PEAR::DB currently supports three such formats. By default this will be an array, ordered from zero, as shown above. The optional parameter of *DB_FETCHMODE_ORDERED* has been omitted in the previous example. This is useful for handling generic databases, or displaying tables without the need to know, or reference, the field names.

Listing 2: Accessing MySQL

```
function GetStationsList()
{
  $db = mysql_connect("localhost", "www-data", "");
  $query = "SELECT * FROM tv.channels";
  $result = mysql_query($query);
  while ($row = mysql_fetch_array($result, MYSQL_NUM)) {
    print "$row[0] - $row[2] ($row[1])<br>";
  }
  mysql_free_result($result);
  mysql_close($db);
}
```

In most situations, however, numeric indices are not descriptive enough, and so we can request that the results are given to us in an associated array. This results in more easily readable code, but at the expensive of generality.

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

Finally, *fetchRow* provides a means of using the object-oriented features of PHP to return an object for each row of the results table. Each column is labeled as a property. Again, this makes the code easy to read, but it is of limited use for more general applications.

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

One for the Road

No code is ever complete without error checking and documentation. Neither are glamorous, but both are necessary. The error checking capabilities of PEAR::DB have been unified (as have the error codes), and stem from the basic error handling features of PEAR_Error. Whenever a PEAR::DB function fails, most will return an instance of an error class – from the grandest *connect*, to the lowliest *getRow*. This class not only holds the error from the PEAR function, but extra debug information which can be useful in tracking down problems.

```
$db =& DB::connect($dsn);
// DB::isError is the same as PEAR::isError
if (DB::isError($db))
```

Table 1:Portability Options

Option	Description
DB_PORTABILITY_LOWERCASE	Convert field and table names to lower case (on get and fetch)
DB_PORTABILITY_RTRIM	Trim output from right
DB_PORTABILITY_DELETE_COUNT	Always reports number of rows deleted
DB_PORTABILITY_NUMROWS	A hack for Oracle's numRows
DB_PORTABILITY_ERRORS	Map error messages between different data bases
DB_PORTABILITY_NULL_TO_EMPTY	Convert null's into empty strings (from get and fetch) because Oracle can't tell the difference between them

```
{
  print $db->getMessage();
  print $db->getDebugInfo();
}
```

Errors can also be trapped using the standard PEAR error handler. This is a user-defined function which will be called whenever a PEAR module (like DB) generates an error. This call-back function can be used in production code to give a standard HTML page to the user, while also alerting the administrator to the problem.

These error handlers are traditionally combined with the buffered output feature of PHP, enabling any partially-built page to be cleared from the HTML stream. Since this is a feature of PEAR, not PHP, traditional errors (such as divide by zero) will not get caught by this.

```
// Prepare the handler
PEAR::setErrorHandling(PEAR_ERROR_CALLBACK, 'error_function');
// Switch on output buffering
ob_start();
// Do normal stuff
PearVersion();
// Flush output buffer (not necessary, but tidy)
ob_end_flush();
// Prepare our handler
function error_function($err)
{
ob_end_clean();
print "An error (".$err->getMessage().") happened!");
```

Listing 3: PEAR naming convention

```
function PearVersion()
{
global $dbname, $dbhost, $dbuser, $dbbackend;
$dsn = "$dbbackend://$dbuser@$dbhost/$dbname";
$db =& DB::connect($dsn);
$query = 'SELECT * FROM channels';
$result = $db->query($query);
while ($row = $result->fetchRow()) {
  print "$row[0] - $row[2] ($row[1])<br>";
}
$result->free();
$db->disconnect();
}
```

1000MB ONLY £125*

Our Java enabled Z account is now even better than before. Nearly 10 times the space for half the price.

1000MB Space
Generous Bandwidth
200+ POP3 Mailboxes
MySQL Database
PHP, Perl, CGI and Java
3 Hour Support Promise
Emergency SMS Pager
Anti Spam Included
Anti Virus Option
100% Linux
Free Giraffe (As Shown)

Freephone 0800 316 1006
www.positive-internet.com

we're good

* + Vat per year

```
14 exit;
15 }
```

The Real Slim Shady

PEAR::DB really is as easy as it appears. The complexity comes instead from SQL itself. PEAR::DB only manages to shield some of this away from you. SQL has been in existence for many years. Within the usual rules of vendor warfare, special extensions were added to each specific variant of SQL making it incompatible with everyone else's.

Although ANSI managed to standardize several parts of the language (the basic versions of *select*, *insert* and *update* are fairly portable), there are still many holes. Some big enough to drive a planet through! Writing standard SQL is a task in itself, and there are several rules of thumb to know and adopt. Most hardcore database programmers know them instinctively. The rest of us need to refer to tutorials such as [7].

In some cases we may wish to use different queries, depending on how the database will react. This requires extra work, either by us, or PEAR::DB. Unfortunately, once we've abstracted the database out of the equation, we have no way of knowing if it is capable of doing what we require of it. PEAR::DB has acknowledged this problem, and features a method called *provides*.

provides indicates the capabilities of the current back-end database. It allows us to switch between two hand-tuned queries to help improve performance in special cases. We use the capabilities (another abstraction!), rather than a specific database, because things change. A later version of the database might support a new feature. Or a new database might come onto the market. The PEAR::DB *provides* method provides (no pun intended!) a means for us to use more optimal queries in our database without having to understand anything about the new database.

Table 3:Support for *provides*

string	Functionality
prepare	Does the database pre-check the SQL query
pconnect	Persistent connections
transactions	Does the database support transactions
limit	Limited select queries

Table 4: Limited SQL

Database	SQL Syntax
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

```
if ($db->provides
('transactions'))
 print "Hooray! It's supported";
```

The range of features for which we can test is shown in Table 3.

In each case it is possible that the database does not natively support the feature. The key word here is *natively*, because there is a distinction between back-end database and PEAR::DB driver. For example, if the database does not support prepare/execute commands, the driver will support the *appearance* of prepare/execute through emulation.

It can be tempting to use *provides* to create completely different, hand-tuned, queries for each database. In most cases, this is unnecessary and A Bad Thing^{TM}. The reason (some would say excuse) for this behavior stems from the extensions that are prominent within SQL. The typical example of this problem comes from limited select queries, which stop producing results after the first, say, 10 rows. The current crop of databases can all perform this task, but does so with different SQL query strings, as shown in Table 4. Coding each example explicitly will create a lot of extra work.

Also, because we cannot cater for every database (including new, currently unwritten, ones), our code will become non-portable very quickly, and suffer 'code rot'. This problem is easy to solve, however, as we can employ the same principle of abstraction. PEAR::DB provides a method called *limitQuery* that hides the precise syntax away from the end user, and adapts to whatever is compatible with the current database back end. This invariably makes more sense than writing separate queries for each database ourselves.

```
$query = "SELECT name FROM 
channels"; // no reference to 
limits here!
$result = limitQuery
($query, 2, 1);
```

This command retrieves 1 row of results from the query, starting at index 2. Because we're counting from 0, this means the 3rd entry.

If the SQL select query can be modified to create a suitable string for the database in question, *$db->provides('limit')* will return *alter*, and the query will be modified by the PEAR::DB driver before being passed on. Otherwise, *provides* might return *emulate*, because the driver is able to fetch query results on a row-by-row basis. Or it could return *false*. You should always query the database features using the results from *provides*, and not your memory or experience. However, for comparison, the current set of drivers provides the functionality as shown in Table 5.

In some rare cases, it is necessary to know the precise database being used. This is because of bugs in the database itself. They're a fact of life. But if we can't remove them, we have to know where they are so we can at least avoid them.

Table 5: What *provides* provides

Database	prepare	pconnect	transactions	limit
FrontBase	false	true	true	emulate
InterBase	true	true	true	false
Informix	false	true	true	emulate
Mini SQL	false	true	false	emulate
Microsoft SQL	false	true	true	emulate
MySQL	false	true	true	alter
Oracle 8i	false	true	true	alter
ODBC	true	true	false	emulate
PostgreSQL	false	true	true	alter
Sybase	false	true	false	emulate

Listing 4: tableInfo

```
$tableinfo = $result->tableInfo());
for($col=0; $col < $result->numCols(); $col++) {
  print $tableinfo[$col]['name']." is a ".$tableinfo[$col]['type'];
}
```

Listing 5: assertExtension

```
if (DB::assertExtension("oci8"))
  print "Use Oracle if we have to...";
if (DB::assertExtension("mysql"))
  print "Use MySQL because we understand it better...";
```

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

This yields the same identifiers as shown in Table 2. Details of specific database bugs are beyond the scope of this article.

Especially for You

As a special bonus, PEAR::DB doesn't only provide abstraction for databases. It also includes a few extra bells and whistles for handling this data, such as *numRows* and *numCols*.

```
print "Select query produced ".$result->numRows(). " rows";
print "Select query produced ".$result->numCols(). " cols";
```

The results of these functions are fairly self-explanatory, and could be deduced from the select query itself. When the query is generated automatically, we can avoid having to manually count the columns. We could also use these numbers to iterate through our results.

There is also a *tableInfo* method which can provide information about each column in our results, such as its name and type. This is not only very useful in debugging, but also good for creating generalized database applications. We could color different columns according to type, or highlight the key field. See Listing 4.

In addition to *name* and *type*, you can also query each column's *length*, *flags* (which indicates the primary key) and *table* name, using the syntax above.

Although much of the API supports the *select* query, there is still room for a little affection to be lavished upon other queries, such as *insert*. One such feature is the *affectedRows* method. This, as you might guess, returns the number of rows that were affected by a query like *insert*, *delete*, or *update*. There is also support to generate unique ID's using the sequence function, *nextID*, which is useful for generating unique keys – something not natively supported by MySQL. For example:

```
// Get a new ID.
$id = $db->nextID('sequence');
// (the sequence will be created if it doesn't exist)
```

Another useful feature is the statically declared *assertExtension* method. This, being static, doesn't require a database object, and will indicate what features are included with the current installation. In its most useful incarnation, it can determine which databases are installed on the system, and the best suited can then be selected for use within the application. See Listing 5.

Despite its name, this is not an assertion in the traditional programming sense, as no error is explicitly output, should the condition not be met. It simply maintains that such an extension does, or does not, exist.

Finally in this section, we shall briefly mention the pairing of *prepare* and *execute*. This is a feature that *prepare*s a query for repeated execution by effectively pre-compiling it into tokens. The query will generally contain place holders for other information, so the query can be *execute*d many times, with different data in place of the generalized markers. See Listing 6.

Listing 6: Preparing a query

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

The *executeMultiple* method would evaluate to the SQL queries,

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

executeMultiple will stop at the first error, should it occur. To circumvent this, you will need to execute each query in turn with the *execute* method instead. For example,

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

Incidentally, the data array must be numerically indexed from zero. In most cases you should see a speed improvement when several entries are inserted into the database at once. This benefit assumes that your database supports the feature natively which, alas, not all do.

Close to the Edit

With PEAR::DB in your armory you can attack most databases, without having to worry about missing functionality, or unsupported features. Using the many utility functions, and the extensive range of database methods, a good cross-platform database application can be written in PHP without worrying about the minor battles. So you can concentrate on the war. ■

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/*
[5] Linux Magazine, issue 41, April 2004, p66
[6] PEAR manual: *http://www.pear.php.net/manual/en/installation.cli.php*
[7] Writing Portable SQL Code: *http://php.weblogs.com/portable_sql*