Database Abstraction with php

By icarus

Table of Contents

Rebirth

I'm almost ashamed to admit it, but a long time ago, I used to be a Perl programmer.

Not a good one, mind you. A very bad one. One who would continually get confused by the difference between hashes and arrays, and hated every second of writing a regex. One who felt like curling up and crying for Mommy every time he had to use strict().

Then I discovered PHP, and my life suddenly took a turn for the better. Gone were those agonizingly−strange symbols and convoluted syntactical constructs. PHP was clean, user−friendly and came with a manual that actually seemed to be written by humans. I felt reborn.

There was one thing I missed about Perl, though − the DBI. The DBI provided a database−independent way to write database−driven Perl code, a necessity when developing dynamic Web sites. PHP, on the other hand, had specific functions to be invoked for each database type − which made code written for one database system hard to port over to other database systems.

Then I found PHP's database abstraction layer, and I felt like I'd come home.

Alphabet Soup

I'll start with the basics − what the heck is a database abstraction layer anyhow?

A database abstraction layer is a database−independent software interface. As the name suggests, it's a layer of abstraction over the actual database access methods and allows developers to deal with different databases without radically altering their code on a per−database basis.

By placing a layer of abstraction between the database and the developer, the database abstraction layer insulates the programmer from database implementation details. If you initially write a script to talk directly to, say, Oracle and later need to have it work with another database server, you will usually have to rewrite all the database−specific parts. If you use a database−independent API, you can port your script over with very little surgery required.

PHP's database abstraction layer comes courtesy of PEAR, the PHP Extension and Application Repository (http://pear.php.net). In case you didn't know, PEAR is an online repository of free PHP software, including classes and modules for everything from data archiving to XML parsing. When you install PHP, a whole bunch of PEAR modules get installed as well; the DB class is one of them.

Sultans Of Swing

Before we begin, you might want to take a quick look at the tables I'll be using throughout this article. Here they are:

```
mysql> SELECT * FROM cds;
+----+------------------+--------------+
| id | title | artist |
+----+------------------+--------------+
| 16 | On Every Street | Dire Straits |
| 15 | Fever | Kylie Minogue |
| 17 | Hell Freezes Over | The Eagles |
+----+------------------+--------------+
3 rows in set (0.06 sec)

mysql> SELECT * FROM tracks;
+----+---------------------+------+
| cd | track | indx |
+----+---------------------+------+
| 17 | I Can't Tell You Why | 9 |
| 15 | More More More | 1 |
| 17 | Hotel California | 6 |
| 15 | Love At First Sight | 2 |
| 16 | Sultans Of Swing | 1 |
| 16 | Lady Writer | 2 |
| 16 | Romeo And Juliet | 3 |
+----+---------------------+------+
7 rows in set (0.00 sec)
```

As you can see, I've got my data split up into two tables in order to avoid duplicating information unnecessarily (geeks like to call this "normalization", simply because it sounds way cooler than "I'm really lazy"). The "cd" table contains a list of all the CDs currently taking up shelf space in my living room, while the "tracks" table lists the tracks on each CD. The two tables are linked to each by a unique CD identifier (again, the geek term for this is "foreign key", but you can forget that one immediately).

Using SQL, it's possible to easily link these two tables together in order to obtain a complete picture of the data contained within them:

```
mysql> SELECT title, artist, indx, track FROM cds, tracks
WHERE cds.id =
tracks.cd ORDER BY cd, indx;
+------------------+--------------+------+---------------------+
| title | artist | indx | track |
+------------------+--------------+------+---------------------+
| Fever | Kylie Minogue | 1 | More More More |
```

```
| Fever | Kylie Minogue | 2 | Love At First Sight |
| On Every Street | Dire Straits | 1 | Sultans Of Swing |
| On Every Street | Dire Straits | 2 | Lady Writer |
| On Every Street | Dire Straits | 3 | Romeo And Juliet |
| Hell Freezes Over | The Eagles | 6 | Hotel California |
| Hell Freezes Over | The Eagles | 9 | I Can't Tell You Why |
+------------------+--------------+------+---------------------+
7 rows in set (0.00 sec)
```

You can do lots of other things with these two tables as well, but they're all completely useless so far as this article is concerned. So let's get started with some code − you can play with SQL on your own time.

Let's suppose I want to display a list of CDs I like on my personal Web page. The data's already there in my table; all I need to do is extract it and display it. Since PHP comes with out−of−the−box support for MySQL, accomplishing this is almost as simple as it sounds.

```
<?php
// uncomment this to see plaintext output in your browser
// header("Content−Type: text/plain");

// open connection to database
$connection = mysql_connect("localhost", "john", "doe") or die
("Unable to
connect!");

// select database
mysql_select_db("db278") or die ("Unable to select
database!");

// execute query
$query = "SELECT * FROM cds";
$result = mysql_query($query) or die ("Error in query: $query.
" .
mysql_error());

// iterate through rows and print column data
// in the form TITLE − ARTIST
while ($row = mysql_fetch_row($result))
{
echo "$row[1] − $row[2]\n";
}

// get and print number of rows in resultset
echo "\n[" . mysql_num_rows($result) . " rows returned]\n";

// close database connection
mysql_close($connection);
```

```
?>
```

Here's what the output looks like:

```
On Every Street − Dire Straits
Fever − Kylie Minogue
Hell Freezes Over − The Eagles

[3 rows returned]
```

The process here is fairly straightforward: connect to the database, execute a query, retrieve the result and iterate through it. The example above uses the mysql_fetch_row() function to retrieve each row as an integer−indexed array, with the array indices corresponding to the column numbers in the resultset; however, it's just as easy to retrieve each row as an associative array (whose keys correspond to the column names) with mysql_fetch_assoc(), or an object (whose properties correspond to the column names) with mysql_fetch_object().

The problem with this script? Since I've used MySQL−specific functions to interact with the database, it's going to crash and burn the second I switch my data over to PostgreSQL or Oracle. Which is where the database abstraction layer comes in.

Independence Day

In order to demonstrate how the abstraction layer works, I'll use it to rewrite the previous example − take a look:

```
<?php
// uncomment this to see plaintext output in your browser
// header("Content−Type: text/plain");

// include the DB abstraction layer
include("DB.php");

// connect to the database
$dbh = DB::connect("mysql://john:doe@localhost/db287");

// execute query
$query = "SELECT * FROM cds";
$result = $dbh−>query($query);

// iterate through rows and print column data
// in the form TITLE − ARTIST
while($row = $result−>fetchRow())
{
echo "$row[1] − $row[2]\n";
}

// get and print number of rows in resultset
echo "\n[" . $result−>numRows() . " rows returned]\n";

// close database connection
$dbh−>disconnect();

?>
```

This output of this snippet is equivalent to that of the previous one; however, since it uses database−independent functions to interact with the database server, it holds out the promise of continuing to work no matter which database I use. I'll show you how in a minute − but first, a quick explanation of the functions used above:

1. The first step is, obviously, to include the abstraction layer in your script. The class file can usually be found in the PEAR installation directory on your system; you can either provide include() with the full path, or (if your PEAR directory is included in PHP's "include_path" variable) just the class file name.

```
<?
// include the DB abstraction layer
```

```
include("DB.php");
?>
```

2. Next, open up a connection to the database. This is accomplished by statically invoking the DB class' connect() method, and passing it a string containing connection parameters.

```
<?
$dbh = DB::connect("mysql://john:doe@localhost/db287");
?>
[code]

Reading this may make your head hurt, but there *is* method to
the madness
- roughly translated, the line of code above attempts to open
up a
connection to the MySQL database named "db287", on the host
named
"localhost", with the username "john" and password "doe".

In case you're wondering, the format of the connection string
is:

[code]
type(syntax)://user:pass@protocol+host/database
```

Supported database types include "fbsql", "ifx", "mssql", "oci8", "pgsql", "ibase", "msql", "mysql", "odbc" and "sybase".

Here are a few examples of how this might be used:

```
mysql://john:doe@dbserver/mydb

mysql://john:doe@some.host.com:123/mydb

pgsql://john@dbserver/mydb

pgsql://dbserver/mydb

odbc://john:doe@some.host.com/mydb
```

If all goes well, the connect() method will return an object that can be used in subsequent database operations.

This also means that you can open up connections to several databases simultaneously, using different connect() statements, and store the returned handles in different variables. This is useful if you need to access

several databases at the same time; it's also useful if you just want to be a smart−ass and irritate the database administrators. Watch them run as the databases overheat! Watch them scurry as their disks begin to thrash! Watch them gibber and scream as they melt down!

3. Once the connect() method returns an object representing the database, the object's query() method can be used to execute SQL queries on that database.

```
<?
// execute query
$query = "SELECT * FROM cds";
$result = $dbh−>query($query);
?>
```

Successful query execution returns a new object containing the results of the query.

4. This object exposes methods and properties that can be used to extract specific fields or elements from the returned resultset.

```
<?
// iterate through rows and print column data
// in the form TITLE − ARTIST
while($row = $result−>fetchRow())
{
echo "$row[1] − $row[2]\n";
}
?>
```

In this case, the object's fetchRow() method is used, in combination with a "while" loop, to iterate through the returned resultset and access individual fields as array elements. These elements are then printed to the output device.

Once all the rows in the resultset have been processed, the object's numRows() method is used to print the number of rows in the resultset.

```
<?
// get and print number of rows in resultset
echo "\n[" . $result−>numRows() . " rows returned]\n";
?>
```

5. Finally, with all the heavy lifting done, the disconnect() method is used to gracefully close the database connection and disengage from the database.

```
<?
// close database connection
$dbh->disconnect();
?>
```

In the event that someone (maybe even me) decides to switch to a different database, the only change required in the script above would be to the line invoking connect() − a new connection string would need to be passed to the function with new connection parameters. Everything else would stay exactly the same, and my code would continue to work exactly as before.

This is the beauty of an abstraction layer − it exposes a generic API to developers, allowing them to write one piece of code that can be used in different situations, with all the ugly bits hidden away and handled internally. Which translates into simpler, cleaner code, better script maintainability, shorter development cycles and an overall Good Feeling. Simple, huh?

Different Strokes

Let's now look at a few of the other methods exposed by the PEAR abstraction layer. Consider the following variant of the example you just saw, this one retrieving the resultset as an associative, or string−indexed, array:

```
<?php
// uncomment this to see plaintext output in your browser
// header("Content−Type: text/plain");

// include the DB abstraction layer
include("DB.php");

// connect to the database
$dbh = DB::connect("mysql://john:doe@localhost/db287");

// execute query
$query = "SELECT * FROM cds";
$result = $dbh−>query($query);

// iterate through rows and print column data
// in the form TITLE − ARTIST

// in this case, each row is fetched as an associative array,
// whose keys are the column names
while($row = $result−>fetchRow(DB_FETCHMODE_ASSOC))
{
echo $row['title'] . " − " . $row['artist'] . "\n";
}

// get and print number of rows in resultset
echo "\n[" . $result−>numRows() . " rows returned]\n";

// close database connection
$dbh−>disconnect();

?>
```

In this case, the additional argument to the fetchRow() function requests that the data be retrieved as elements of an associative array (the default is a "regular" array, indexed by number rather than string). The keys of this array are the column names of the resultset. In the event that a column name is repeated − as might happen, for example, with a table join − the last one will be used.

It's also possible to retrieve a row as an object, with the fields within it exposed as object properties. Take a look:

```
<?php
// uncomment this to see plaintext output in your browser
// header("Content-Type: text/plain");

// include the DB abstraction layer
include("DB.php");

// connect to the database
$dbh = DB::connect("mysql://john:doe@localhost/db287");

// execute query
$query = "SELECT * FROM cds";
$result = $dbh->query($query);

// iterate through rows and print column data
// in the form TITLE - ARTIST

// in this case, each row is fetched as an object,
// whose properties correspond to the column names
while($row = $result->fetchRow(DB_FETCHMODE_OBJECT))
{
echo $row->title . " - " . $row->artist . "\n";
}

// get and print number of rows in resultset
echo "\n[" . $result->numRows() . " rows returned]\n";

// close database connection
$dbh->disconnect();

?>
```

You can play the tyrannical dictator and impose a default mode for resultset retrieval with the setFetchMode() method. The following example demonstrates:

```
<?php
// uncomment this to see plaintext output in your browser
// header("Content-Type: text/plain");

// include the DB abstraction layer
include("DB.php");

// connect to the database
$dbh = DB::connect("mysql://john:doe@localhost/db287");

// set default mode for all resultsets
```

```
$dbh->setFetchMode(DB_FETCHMODE_ASSOC);

// execute query
$query = "SELECT * FROM cds";
$result = $dbh->query($query);

// iterate through rows and print column data
// in the form TITLE - ARTIST
while($row = $result->fetchRow())
{
echo $row['title'] . " - " . $row['artist'] . "\n";
}

// get and print number of rows in resultset
echo "\n[" . $result->numRows() . " rows returned]\n";

// close database connection
$dbh->disconnect();

?>
```

In this case, the default mode of operation for the fetchRow() method is to structure each row into an associative array. Isn't it wonderful how PHP lets you be both lazy and efficient at the same time?

The Number Game

The numRows() and numCols() methods can be used to obtain the number of rows and columns in the returned resultset respectively:

```
<?php
// uncomment this to see plaintext output in your browser
// header("Content-Type: text/plain");

// include the DB abstraction layer
include("DB.php");

// connect to the database
$dbh = DB::connect("mysql://john:doe@localhost/db287");

// execute query
$query = "SELECT * FROM cds";
$result = $dbh->query($query);

// print number of rows and columns in resultset
echo "Query returned " . $result->numRows() . " rows of " .
$result->numCols() . " columns each";

// close database connection
$dbh->disconnect();

?>
```

The LimitQuery() method can be used to restrict the number of rows retrieved:

```
<?php
// uncomment this to see plaintext output in your browser
// header("Content-Type: text/plain");

// include the DB abstraction layer
include("DB.php");

// set some variables for limited query
$start = 2;
$num = 4;

// connect to the database
$dbh = DB::connect("mysql://john:doe@localhost/db287");

// execute limited query
```

```
$query = "SELECT track FROM tracks";
$result = $dbh->LimitQuery($query, $start, $num);

echo "[Retrieving rows $start through " . ($start+$num) . " of
resultset]\n";

// iterate through rows and print column data
// in the form TITLE - ARTIST
while($row = $result->fetchRow())
{
echo "$row[0]\n";
}

// close database connection
$dbh->disconnect();

?>
```

In this case, the LimitQuery() method can be used to obtain a subset of the complete resultset retrieved from the database. The first argument to the method is the query to execute, the second is the row offset from which to begin, and the third is the number of rows required.

If you're feeling voyeuristic, the tableInfo() method can be used to take a quick peek at the structure of the table(s) returned by your query. Consider the following example,

```
<?php
// uncomment this to see plaintext output in your browser
// header("Content-Type: text/plain");

// include the DB abstraction layer
include("DB.php");

// connect to the database
$dbh = DB::connect("mysql://john:doe@localhost/db287");

// execute query
$query = "SELECT title, artist, track FROM cds, tracks WHERE
cds.id =
tracks.cd";
$result = $dbh->query($query);

// get info on structure of tables used in query
// this is returned as an array of arrays
// dump it with print_r()!
print_r($result->tableInfo());

// close database connection
```

```
$dbh->disconnect();

?>
[code]

And then take a look at the output of the tableInfo() command,
as seen
through the print_r() function:

[output]
Array
(
[0] => Array
(
[table] => cds
[name] => title
[type] => string
[len] => 255
[flags] => not_null
)

[1] => Array
(
[table] => cds
[name] => artist
[type] => string
[len] => 255
[flags] => not_null
)

[2] => Array
(
[table] => tracks
[name] => track
[type] => string
[len] => 255
[flags] => not_null
)

)
```

As you can see, information on the table structure − the field names, data types, flags et al − is returned by tableInfo() as an array. Every element of this array corresponds to a column in the resultset, and is itself structured as an associative array. Note that this method only works if your query actually returns a valid resultset − so you can't use it with INSERT or UPDATE queries. Finally, the free() method is used to free the resources associated with a particular resultset. [code] <?php // uncomment this to see plaintext output in your browser // header("Content−Type: text/plain"); // include the DB abstraction layer include("DB.php"); // connect to the database $dbh = DB::connect("mysql://john:doe@localhost/db287"); // execute query $query =

"SELECT * FROM cds"; $result = $dbh−>query($query); // iterate through rows and print column data // in the form TITLE − ARTIST while($row = $result−>fetchRow()) { echo "$row[1] − $row[2]\n"; } // free resultset $result−>free(); // close database connection $dbh−>disconnect(); ?>

Just out of curiosity, look what happens if you use free() in the wrong place.

```
<?php
// uncomment this to see plaintext output in your browser
// header("Content−Type: text/plain");

// include the DB abstraction layer
include("DB.php");

// connect to the database
$dbh = DB::connect("mysql://john:doe@localhost/db287");

// execute query
$query = "SELECT * FROM cds";
$result = $dbh−>query($query);

// iterate through rows and print column data
// in the form TITLE − ARTIST
while($row = $result−>fetchRow())
{
echo "$row[1] − $row[2]\n";

// free resultset
$result−>free();
}

// close database connection
$dbh−>disconnect();

?>
```

In this case, the "while" loop will execute only once, since the resultset gets free()d on the first iteration of the loop. Consequently, only one record will be displayed as output.

This is kinda pointless − after all, if you only wanted to display a single record, you wouldn't need a "while" loop in the first place − but interesting to try out; it serves as both a warning to newbies and a source of amusement to more experienced geeks.

Preparing For The Long Haul

In the event that you need to execute a particular query multiple times with different values − for example, a series of INSERT statements − the DB class comes with two methods that can save you a huge amount of time (see, there's that lazy thing again!) and also reduce overhead. Consider the following example, which demonstrates:

```
<?php
// include the DB abstraction layer
include("DB.php");

// set up array of track titles
$tracks = array("Track A", "Track B", "Track C", "Track D");

// connect to the database
$dbh = DB::connect("mysql://john:doe@localhost/db287");

// prepare query
$query = $dbh->prepare("INSERT INTO tracks (cd, track) VALUES
(12, ?)");

// execute query
// run as many times as there are elements in $tracks
foreach($tracks as $t)
{
$result = $dbh->execute($query, $t);
}

// close database connection
$dbh->disconnect();

?>
```

The prepare() function, which takes an SQL query as parameter, readies a query for execution, but does not execute it (kinda like the priest that walks down the last mile with you to the electric chair). Instead, prepare() returns a handle to the prepared query, which is stored and then passed to the execute() method, which actually executes the query (bzzzt!).

Note the placeholder used in the query string passed to prepare() − this placeholder is replaced by an actual value each time execute() runs on the prepared statement. The second argument to execute() contains the values to be substituted in the query string.

In the event that you have multiple placeholders within the same query string, you can pass execute() an array of values instead of a single value − as demonstrated below:

```
<?php
// include the DB abstraction layer
include("DB.php");

// set up array of CD titles
$cds = array();

// each element of this array is itself an array
$cds[] = array("CD A", "Artist A");
$cds[] = array("CD B", "Artist B");
$cds[] = array("CD C", "Artist C");

// connect to the database
$dbh = DB::connect("mysql://john:doe@localhost/db287");

// prepare query
$query = $dbh->prepare("INSERT INTO cds (title, artist) VALUES
(?, ?)");

// execute query
// run as many times as there are elements in $cds
foreach($cds as $c)
{
$result = $dbh->execute($query, $c);
}

// close database connection
$dbh->disconnect();

?>
```

Although overkill for our simple needs, you should be aware that prepare() can also read data directly from a file, rather than an array. I'll leave this to you to experiment with.

Commitment Issues

If your database system supports transactions (MySQL doesn't, but quite a few others do), you'll be thrilled to hear that PHP's database abstraction layer comes with a bunch of methods that allow you to transparently use this feature in your scripts. What you may not be so thrilled about, though, is the fact that it also allows me the opportunity to make a few bad puns.

The following example demonstrates:

```
<?php
// uncomment this to see plaintext output in your browser
// header("Content-Type: text/plain");

// include the DB abstraction layer
include("DB.php");

// connect to the database (Oracle)
$dbh = DB::connect("oci8://john:doe@listener/db287");

// turn off autocommit
$dbh->autoCommit(false);

// execute one query
$dbh->query("INSERT INTO cds (id,title, artist)
VALUES(SEQ_CDS_ID.nextval,
'Weathered', 'Creed')");

// commit the data to database
$dbh->commit();

// execute one more query
$dbh->query("INSERT INTO cds (id,title, artist)
VALUES(SEQ_CDS_ID.nextval,
'The Joshua Tree', 'U2')");

// now rollback to previous stage
$dbh->rollback();

// now if you check the contents of the table
// only one entry will be visible
// as the second action has been rolled back

// close database connection
$dbh->disconnect();

?>
```

The first step here is to turn off auto−committal of data to the database, via the autoCommit() method. Once that's done, you can go ahead and execute as many queries as you like, secure in the knowledge that no changes have (yet) been made to the database.

Once your fear of commitment passes (and I don't mean just here, oh no!), you can save your data to the database via a call to the commit() method. In the event that you realize you made a mistake, you can uncommit yourself gracefully (notice how this never happens in real life?) with the rollback() function. It's simple, it's healthy, and the guy who manages to apply it to human behaviour will make a gazillion.

No News Is Good News

You may remember, from your misadventures with MySQL, that MySQL's API includes a function called mysql_error(), which returns information on the last error that occurred. While this is not really something you would miss − I mean, how can you love someone who always brings you bad news? − it's still useful to have lying around, in the event you need it within a script or application.

With the database abstraction layer, though, you're insulated from PHP's native functions − so how do you handle errors? Are your users doomed to spend eternity staring at a cryptic error message or even (joy!) a blank screen? Or is there a white knight who will gallop up and save them from this hell?

As it turns out, there is. The PEAR abstraction layer comes with a bunch of methods designed specifically to simplify the task of handling errors. Take a look at the next example, which demonstrates how they can be used in a PHP script:

```
<?php
// uncomment this to see plaintext output in your browser
// header("Content-Type: text/plain");

// include the DB abstraction layer
include("DB.php");

// connect to the database
$dbh = DB::connect("mysql://john:doe@localhost/db287");

// execute query (note the error!)
$query = "RE-ELECT * FROM cds";
$result = $dbh->query($query);

// if an error occurred
if ($dbh->isError($result))
{
// display an error message and exit
echo "Take cover. Something bad just happened.";
exit();
}
// or...
else
{
// iterate through rows and print column data
// in the form TITLE - ARTIST
while($row = $result->fetchRow())
{
echo "$row[1] - $row[2]\n";
}
}
```

```
// close database connection
$dbh->disconnect();

?>
```

The theory here is that the isError() method can be used to find out if an error occurred during query execution, and display an error message appropriately. You'll notice that I've deliberately introduced a syntax error into the query string above, just to see how true this really is. Here's the output:

```
Take cover. Something bad just happened.
```

Hmmm. Guess it works after all. But how about a more descriptive error message, one that actually tells you what went wrong?

```
<?php
// uncomment this to see plaintext output in your browser
// header("Content-Type: text/plain");

// include the DB abstraction layer
include("DB.php");

// connect to the database
$dbh = DB::connect("mysql://john:doe@localhost/db287");

// execute query
$query = "RE-ELECT * FROM cds";
$result = $dbh->query($query);

// if an error occurred
if ($dbh->isError($result))
{
// display an error message and exit
echo "Take cover. Something bad just happened.\n";
echo "You said: $query\n";
echo "The database said: " . DB::errorMessage($result) . "\n";
exit();
}
// or...
else
{
// iterate through rows and print column data
// in the form TITLE - ARTIST
while($row = $result->fetchRow())
{
echo "$row[1] - $row[2]\n";
```

```
}
}

// close database connection
$dbh->disconnect();

?>
```

In this case, the error message has been beefed up with a more descriptive error string, accessible via the errorMessage() method of the DB class. Here's what it looks like:

```
Take cover. Something bad just happened.
You said: RE-ELECT * FROM cds
The database said: syntax error
```

Catch Me If You Can

Now, while the error−handling technique described on the previous page works great so long as you only have a couple of queries running in each script, it can get tedious if you need to execute eight, ten or a hundred queries within a script. Which is why the abstraction layer also provides a generic way to trap and resolve errors, via a default error handler.

This default error handling mechanism can be defined through the setErrorHandling() method, and it accepts any one of the following five parameters:

PEAR_ERROR_RETURN − do nothing PEAR_ERROR_PRINT − print the error message but continue executing the script PEAR_ERROR_TRIGGER − trigger a PHP error PEAR_ERROR_DIE − terminate script execution PEAR_ERROR_CALLBACK − call another function to handle the error

Let's see this in action:

```
<?php
// uncomment this to see plaintext output in your browser
// header("Content−Type: text/plain");

// include the DB abstraction layer
include("DB.php");

// connect to the database
$dbh = DB::connect("mysql://john:doe@localhost/db287");

// set an error handler
$dbh−>setErrorHandling(PEAR_ERROR_PRINT, "The database said:
%s");

// run a sequence of queries
$dbh−>query("INSERT INTO cds (title, artist) VALUES ('Title
A', 'Artist
A')");
$dbh−>query("INSERT INTO cds (title, artist) VALUES ('Title
B', 'Artist
B')");
$dbh−>query("INSERT INTO cds (title, artist) VALUES ('Title
C', Artist
C')");
$dbh−>query("INSERT INTO cds (title, artist) VALUES ('Title
D', 'Artist
D')");

// close database connection
$dbh−>disconnect();
```

```
?>
```

In this case, the third query contains a syntax error (a missing quote). When this error is encountered, an exception is raised and the error handler is invoked to handle the error. Since I've specified

```
<?
// set an error handler
$dbh->setErrorHandling(PEAR_ERROR_PRINT, "The database said:
%s");
?>
```

an error message will appear, but the script will move on to execute the fourth query.

Here's the output:

```
The database said: DB Error: syntax error
```

In case you're wondering, the second argument to the setErrorHandling() method above is an optional string that can be used to apply basic formatting to the error message. You can omit it if you like − the output will look just as pretty without it.

You can also specify a callback function of your own to handle the error, in case you want to do something complicated with it − for example, insert it into a table, or log it to a file. In such a situation, the setErrorHandling() function must be passed two arguments: the constant PEAR_ERROR_CALLBACK, and the name of your user−defined error handler. If an error occurs, this error handler is invoked with an object as argument; this object represents the error, and its properties hold detailed information on what went wrong.

If all this sounds a little complicated, the next example might make it easier to understand:

```
<?php
// uncomment this to see plaintext output in your browser
// header("Content-Type: text/plain");

// include the DB abstraction layer
include("DB.php");

// connect to the database
$dbh = DB::connect("mysql://john:doe@localhost/db287");

// set an error handler
$dbh->setErrorHandling(PEAR_ERROR_CALLBACK, "catchMe");

// run a sequence of queries
```

```
$dbh->query("INSERT INTO cds (title, artist) VALUES ('Title
A', 'Artist
A')");
$dbh->query("INSERT INTO cds (title, artist) VALUES ('Title
B', 'Artist
B')");
$dbh->query("INSERT INTO cds (title, artist) VALUES ('Title
C', Artist
C')");
$dbh->query("INSERT INTO cds (title, artist) VALUES ('Title
D', 'Artist
D')");

// close database connection
$dbh->disconnect();

// user-defined error handler
// logs errors to a file
function catchMe(&$obj)
{
// uncomment next line to see object properties
// print_r($obj);

// create a string to hold the error
// access object properties to get different bits of
information about the
error
$str = "Timestamp: " . date("d-M-Y h:m", mktime()) . "\n";
$str .= "Error code: " . $obj->code . "\n";
$str .= "Error message: " . $obj->message . "\n";
$str .= "Debug string: " . $obj->userinfo . "\n\n";

// append error string to a log file
$fp = fopen("error.log", "a+") or die("Could not open log
file");
fwrite($fp, $str);
fclose($fp);
}

?>
```

In this case, I'm logging every error that occurs to a log file using my own little catchMe() function. This function receives an object when an error occurs; this object can be manipulated to expose detailed information about the error (including the raw error message returned by the server), and this information can then be written to a file.

Once Again, The Headlines

In case your short−term memory is on the fritz, here's a brief summary of everything discussed so far:

A database abstraction layer provides a database−independent interface to developers working on different databases. Perl has one, and now PHP has one too, courtesy of PEAR. Though an abstraction layer can take a little getting used to, it offers tremendous benefits in the long run, especially if you expect platform or database changes during the development cycle. Using an abstraction layer can significantly reduce the amount of time you spend porting your code over from one database to another.

The PEAR abstraction layer currently includes support for a number of different database types, including MySQL, PostgreSQL, Oracle, Sybase and ODBC. In addition to frequently−used methods for executing queries and fetching resultsets, the PEAR abstraction layer also supports pre−prepared queries, transactions and customized error handling. A number of miscellaneous utility functions are also included to assist in query execution and data retrieval.

That just about concludes this little tour of PHP's database abstraction layer. I hope you enjoyed it and found it useful, and that it demonstrated the flexibility and power of one of my favourite PHP widgets. Until next time...ciao!

Note: All examples in this article have been tested on Linux/i386 with Apache 1.3.12 and PHP 4.1.0. Examples are illustrative only, and are not meant for a production environment. Melonfire provides no warranties or support for the source code described in this article. YMMV!