melonfire!

Community

Trog

PHP Application Development With ADODB (part 1)

Make your PHP scripts portable across databases with the powerful ADODB database abstraction library.

Any Port In A Storm

As a developer, one of the most important things to consider when developing a Web application is portability. Given the rapid pace of change in the Web world, it doesn't do to bind your code too tightly to a specific operating system, RDBMS or programming language; if you do, you'll find yourself reinventing the wheel every time things change on you (and they will - take my word for it).

That's where this article comes in.

Over the course of this two-part tutorial, I'm going to be showing you how to make your code a little more portable, by using a database abstraction layer for all your RDBMS connectivity. This database abstraction layer allows you to easily switch between one RDBMS and another, without requiring either a code rewrite or a long, tortuous retest cycle. In the long run, this will save you time, save your customers money, and maybe make your life a little simpler.

Before we get started, one caveat: while portability is something you should strive for regardless of which language or platform you work on, it's impossible to cover every single possibility in this article...and I don't plan to. Instead, I'll be restricting myself to my favourite Web programming language, PHP, and the ADODB database abstraction library, also written in PHP. Similar libraries exist for most other programming languages, and you should have no trouble adapting the techniques in this article to other platforms.

Let's get started!

A Little Insulation

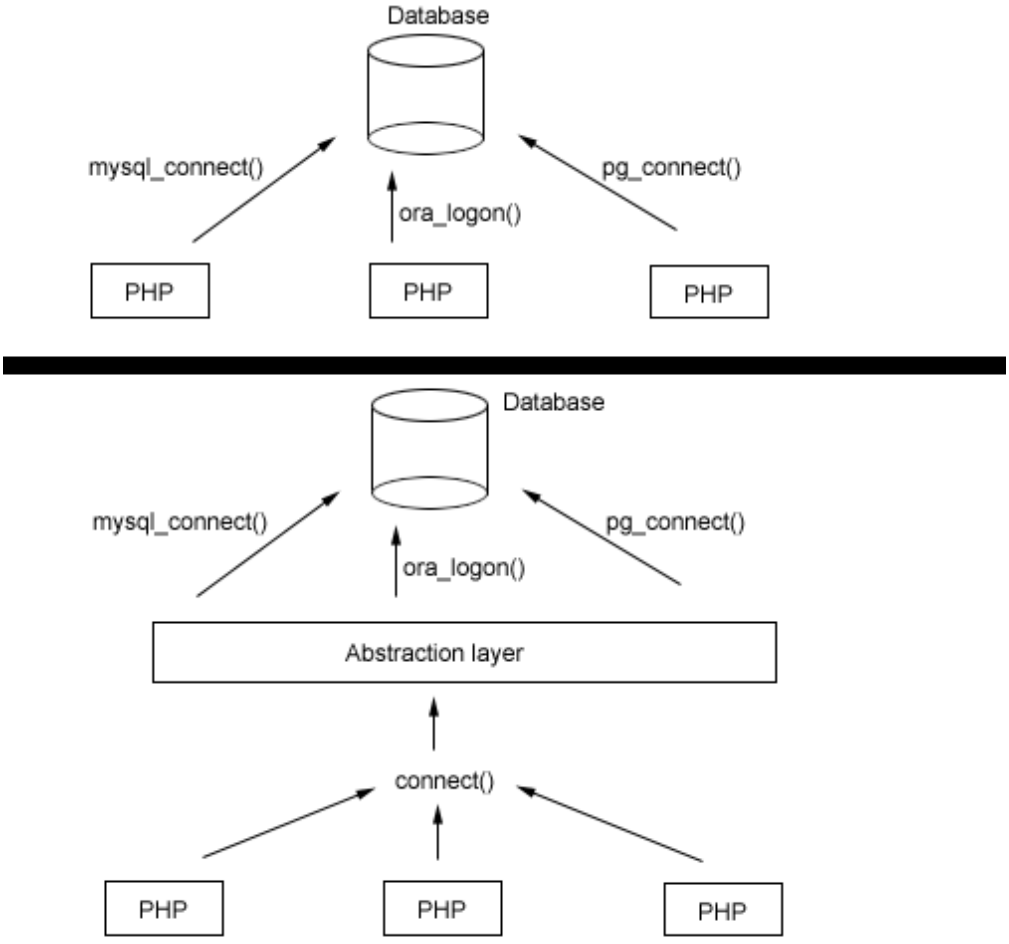
First up, let's get the lingo straight: what the heck is a database abstraction library anyway?

If you've worked with different databases, you've probably seen that each database operates in a slightly different manner from the others. The data types aren't always uniform, and many of them come with proprietary extensions (transactions, stored procedures et al) that aren't supported elsewhere. Additionally, the API to interact with these databases is not uniform; PHP itself comes with a different API for each supported database type,

For all these reasons, switching from one database to another is typically a complex process, one which usually involves porting data from one system to another (with the assorted datatyping complications), rewriting your code to use the new database API, and testing it to make sure it all works. And that's where a database abstraction layer can help.

Typically, a database abstraction layer functions as a wrapper around your code, exposing a set of generic methods to interact with a database server. These generic methods are internally mapped to the native API for each corresponding database, with the abstraction layer taking care of ensuring that the correct method is called for your selected database type. Additionally, most abstraction layers also incorporate a generic superset of datatypes, which get internally converted into datatypes native to the selected RDBMS.

In order to better understand the difference, consider the following diagram:



As you can see, without an abstraction layer in place, you need to use a different API call for each of the three database types. With an abstraction layer in place, however, you can transparently use a single, generic call, and have the abstraction layer convert it into the native API call.

A number of different abstraction layers are available for PHP, most notably the PEAR DBI, Metabase and PHPLib. The one I'm going to use in this article is named ADODB (Active Data Objects DataBase), and it's one of the most full-featured and efficient PHP abstraction libraries available today. Developed by John Lim, the library currently supports a wide variety of database systems, including MySQL, PostgreSQL, Oracle, Interbase, Microsoft SQL Server, Access, ODBC and others, and has been used in a number of well-known open-source PHP projects, including phpLens, PostNuke and Webodex.

You can download a copy of ADODB from http://php.weblogs.com/adodb - get your copy now, set it up,

and flip the page for an example of how it can be used.

The Bookworm Turns

Before we get into the code, you might want to take a quick look at the database table I'll be using throughout this article. Here it is:

```
mysql> SELECT * FROM library;
+----+-------------------+----------------+
| id | title             | author         |
+----+-------------------+----------------+
| 14 | Mystic River      | Dennis Lehane  |
| 15 | For Kicks         | Dick Francis   |
| 16 | XML and PHP       | Vikram Vaswani |
| 17 | Where Eagles Dare | Jack Higgins   |
+----+-------------------+----------------+
```

As you might have guessed, the "library" table contains a list of all the books currently taking up shelf space in my living room. Each record within the table is identified by a unique number (the geek term for this is "foreign key", but you can forget that one immediately).

Now, let's suppose I want to display a list of my favourite books on my personal Web site. Everything I need is stored in the table above; all yours truly has to do is write a script to pull it out and massage it into a readable format. 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 library";
$result = mysql_query($query) or die ("Error in query: $query. " . mysql_error());

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

```
Mystic River - Dennis Lehane
For Kicks - Dick Francis
XML and PHP - Vikram Vaswani
Where Eagles Dare - Jack Higgins

[4 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.

Anatomy Class

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 ADODB library
include("adodb.inc.php");

// create an object instance
// configure it for a MySQL connection
$db = NewADOConnection("mysql");

// open connection to database
$db->Connect("localhost", "john", "doe", "db278") or die("Unable to connect!");

// execute query
$query = "SELECT * FROM library";
$result = $db->Execute($query) or die("Error in query: $query. " . $db->ErrorMsg());

// iterate through resultset
// print column data in format TITLE - AUTHOR
while (!$result->EOF)
{
   echo $result->fields[1] . " - " . $result->fields[2] . "\n";
     $result->MoveNext();
}

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

// close database connection
$db->Close();

?>
```

This output of this snippet is equivalent to that of the previous one; however, since it uses the ADODB abstraction library, rather than PHP's native API, 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.

```
<?
// include the ADODB library
include("adodb.inc.php");
?>
```

Note that the ADODB library doesn't consist of just this file - in fact, there are over thirty different files included with the library, many of them drivers for different databases. You don't need to worry about including each and every one; simply include the main class file, as above, and it will invoke the appropriate drivers or additional classes as required.

2. Next, create an instance of the ADODB class.

```
<?
// create an object instance
// configure library for a MySQL connection
$db = NewADOConnection("mysql");
?>
```

The parameter passed to the object constructor tells ADODB which type of database you're trying to connect to. In this case, I've used the argument "mysql", since I'm going to be connecting to a MySQL database server; you could just as easily use "pgsql" or "oci8" or...

3. Next, it's time to open up a connection to the database. This is accomplished via the Connect() method, which must be passed a set of connection parameters.

```
<?
// open connection to database
$db->Connect("localhost", "john", "doe", "db278") or die("Unable to connect!");
?>
```

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 "db278", on the host named "localhost", with the username "john" and password "doe".

4. Once the Connect() method does its job, the object's Execute() method can be used to execute SQL queries on that database.

```
<?
// execute query
$query = "SELECT * FROM library";
$result = $db->Execute($query) or die("Error in query: $query. " . $db->ErrorMsg());
?>
```

Successful query execution returns a new object containing the results of the query. Note the special ErrorMsg() method, which can be used to obtain the last error message generated by the system.

5. The result object returned in the previous step exposes methods and properties that can be used to extract specific fields or elements from the returned resultset.

```
<?
// iterate through resultset
// print column data in format TITLE - AUTHOR
while (!$result->EOF)
{
   echo $result->fields[1] . " - " . $result->fields[2] . "\n";
     $result->MoveNext();
}
?>
```

In this case, the object's MoveNext() method is used, in combination with a "while" loop, to iterate through the returned resultset and display individual fields (these individual fields are accessed as array elements of the object's "fields" property). This data is then printed to the output device.

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

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

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

```
<?
// close database connection
$db->Close();
?>
```

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 instantiating the connection object - a new argument would need to be passed to the object constructor, with a new database type. 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

ADODB also offers a number of alternative methods to process a resultset. For example, you can retrieve the resultset as a string-indexed associative array, where the keys are field names and the values are the corresponding field values. Consider the following example, which demonstrates:

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

// include the ADODB library
include("adodb.inc.php");

// create an object instance
// configure library for a MySQL connection
$db = NewADOConnection("mysql");

// open connection to database
$db->Connect("localhost", "john", "doe", "db278") or die("Unable to connect!");

// get resultset as associative array
$ADODB_FETCH_MODE = ADODB_FETCH_ASSOC;

// execute query
$query = "SELECT * FROM library";
$result = $db->Execute($query) or die("Error in query: $query. " . $db->ErrorMsg());

// iterate through resultset
// print column data in format TITLE - AUTHOR
while (!$result->EOF)
{
   echo $result->fields['title'] . " - " . $result->fields['author'] . "\n";
     $result->MoveNext();
}

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

// close database connection
$db->Close();

?>
```

In this case, the value of the special $ADODB_FETCH_MODE variable tells ADODB how the resultset should be structured.

You can also fetch each row as an object, whose properties correspond to the field names, via ADODB's FetchNextObject() method.

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

// include the ADODB library
include("adodb.inc.php");

// create an object instance
// configure library for a MySQL connection
$db = NewADOConnection("mysql");

// open connection to database
$db->Connect("localhost", "john", "doe", "db278") or die("Unable to connect!");

// execute query
```

```
$query = "SELECT * FROM library";
$result = $db->Execute($query) or die("Error in query: $query. " . $db->ErrorMsg());

// iterate through resultset
// print column data in format TITLE - AUTHOR
while ($row = $result->FetchNextObject())
{
   echo $row->TITLE . " - " . $row->AUTHOR . "\n";
}

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

// close database connection
$db->Close();

?>
```

Note that this FetchNextObject() method automatically moves to the next row in the resultset, and does not require an explicit call to MoveNext(). Once the end of the resultset is reached, the method returns false.

Getting It All

You can replace the Execute() method with the GetAll() method, which returns the complete resultset as a two-dimensional array of field-value pairs. This array can then be processed with a simple "foreach" or "for" loop. Consider the following example, which demonstrates:

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

// include the ADODB library
include("adodb.inc.php");

// create an object instance
// configure library for a MySQL connection
$db = NewADOConnection("mysql");

// open connection to database
$db->Connect("localhost", "john", "doe", "db278") or die("Unable to connect!");

// execute query
$query = "SELECT * FROM library";
$result = $db->GetAll($query) or die("Error in query: $query. " . $db->ErrorMsg());

// clean up
$db->Close();

// uncomment the following line to see the returned array.
// print_r($result);

// iterate through resultset
// print column data in format TITLE - AUTHOR
foreach ($result as $row)
{
   echo $row[1] . " - " . $row[2] . "\n";
}
```

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

?>
```

In this case, the GetAll() method creates a two-dimensional array of result data, which looks something like this.

```
Array
(
    [0] => Array
        (
            [0] => 14
            [id] => 14
            [1] => Mystic River
            [title] => Mystic River
            [2] => Dennis Lehane
            [author] => Dennis Lehane
        )

    [1] => Array
        (
            [0] => 15
            [id] => 15
            [1] => For Kicks
            [title] => For Kicks
            [2] => Dick Francis
            [author] => Dick Francis
        )

    [2] => Array
        (
            [0] => 16
            [id] => 16
            [1] => XML and PHP
            [title] => XML and PHP
            [2] => Vikram Vaswani
            [author] => Vikram Vaswani
        )

       ... and so on ...
)
```

This array can then be iterated over by a "foreach" loop, and the required values accessed as regular array elements. The size of the array is equivalent to the total number of rows in the resultset.

This method provides a useful alternative to the Execute() method, especially in situations when you would prefer to have the entire recordset available at once, rather than one row at a time.

Playing The Field

ADODB comes with a number of utility functions that provide you with useful information on the query you just executed. The most useful of these are the RecordCount() and FieldCount() methods, which return the number of rows and columns in the recordset respectively. Here's a simple example, which demonstrates:

```
<?php
```

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

// include the ADODB library
include("adodb.inc.php");

// create an object instance
// configure library for a MySQL connection
$db = NewADOConnection("mysql");

// open connection to database
$db->Connect("localhost", "john", "doe", "db278") or die("Unable to connect!");

// execute query
$query = "SELECT * FROM library";
$result = $db->Execute($query) or die("Error in query: $query. " . $db->ErrorMsg());

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

// get and print number of fields in resultset
echo $result->FieldCount() . " fields returned\n";

// clean up
$db->Close();

?>
```

You can obtain further information on each field with the FetchField() method, which returns an object containing detailed information on the field properties, including its name and type. Consider the following variant of the example above, which might make this a little clearer:

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

// include the ADODB library
include("adodb.inc.php");

// create an object instance
// configure library for a MySQL connection
$db = NewADOConnection("mysql");

// open connection to database
$db->Connect("localhost", "john", "doe", "db278") or die("Unable to connect!");

// execute query
$query = "SELECT * FROM library";
$result = $db->Execute($query) or die("Error in query: $query. " . $db->ErrorMsg());

// get field information
for($x=0; $x<$result->FieldCount(); $x++)
{
   print_r($result->FetchField($x));
}

// clean up
$db->Close();

?>
```

Here's what the output might look like for the "id" field:

```
stdClass Object
(
    [name] => id
    [table] => library
    [def] =>
    [max_length] => 3
    [not_null] => 1
    [primary_key] => 1
    [multiple_key] => 0
    [unique_key] => 0
    [numeric] => 1
    [blob] => 0
    [type] => int
    [unsigned] => 1
    [zerofill] => 0
    [binary] =>
)
```

Strange Relationships

In case you're performing an INSERT query on a table containing an auto-increment primary key, you can obtain the last auto-increment ID generated via ADODB's Insert_ID() method.

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

// include the ADODB library
include("adodb.inc.php");

// create an object instance
// configure library for a MySQL connection
$db = NewADOConnection("mysql");

// open connection to database
$db->Connect("localhost", "john", "doe", "db278") or die("Unable to connect!");

// execute query
$title = $db->qstr("It's Not Me, It's You!");
$author = $db->qstr("J. Luser");
$query = "INSERT INTO library (title, author) VALUES ($title, $author)";
$result = $db->Execute($query) or die("Error in query: $query. " . $db->ErrorMsg());

// print auto-generated ID
if ($result)
{
   echo "Last inserted ID is " . $db->Insert_ID();
}

// clean up
$db->Close();
?>
```

Note also the qstr() method, which comes in handy when you need to escape special characters in your query string.

If you're executing a query that affects the table, either by adding, deleting or modifying rows, the Affected_Rows() method can come in handy to tell you the number of rows affected.

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

// include the ADODB library
include("adodb.inc.php");

// create an object instance
// configure library for a MySQL connection
$db = NewADOConnection("mysql");

// open connection to database
$db->Connect("localhost", "john", "doe", "db278") or die("Unable to connect!");

// execute query
$query = "DELETE FROM library WHERE author = 'J. Luser'";
$result = $db->Execute($query) or die("Error in query: $query. " . $db->ErrorMsg());

// return number of affected rows
if ($result)
{
   echo $db->Affected_Rows() . " rows deleted";
}

// clean up
$db->Close();
?>
```

Hitting The Limit

The SelectLimit() 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 ADODB library
include("adodb.inc.php");

// create an object instance
// configure library for a MySQL connection
$db = NewADOConnection("mysql");

// open connection to database
$db->Connect("localhost", "john", "doe", "db278") or die("Unable to connect!");

// execute query
// get 5 rows, starting from row 3
$query = "SELECT * FROM library";
$result = $db->SelectLimit($query, 5, 3) or die("Error in query: $query. " .
```

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

// iterate through resultset
while (!$result->EOF)
{
   echo $result->fields[1] . " - " . $result->fields[2] . "\n";
     $result->MoveNext();
}

// clean up
$db->Close();
?>
```

In this case, the SelectLimit() 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 number of rows required, and the third is the row offset from which to begin.

Finally, you can obtain a list of databases on the server via the MetaDatabases() method, and a list of tables within the current database via the MetaTables() method.

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

// include the ADODB library
include("adodb.inc.php");

// create an object instance
// configure library for a MySQL connection
$db = NewADOConnection("mysql");

// open connection to database
$db->Connect("localhost", "john", "doe", "db278") or die("Unable to connect!");

// get database list
echo "Databases:\n";
foreach($db->MetaDatabases() as $d)
{
   echo "* $d\n";
}

// get table list
echo "\nTables in current database:\n";
foreach($db->MetaTables() as $table)
{
   echo "* $table\n";
}

// clean up
$db->Close();
?>
```

Coming Soon, To A Screen Near You

And that's about it for this introductory tutorial. Over the preceding pages, I demonstrated the basics of the ADODB database abstraction library, explaining how it could be used to insulate your code from the impact of a database change and thereby add portability to your PHP application. I showed you the

different methods the ADODB library provides to iterate over a resultset, and to convert a resultset into native PHP arrays suitable for further processing. Finally, I demonstrated a number of useful utility functions, most notably functions related to counting rows and columns, retrieving database, table and field information, and escaping special characters prior to inserting them into a database.

All this, of course, constitutes just the tip of the ADODB iceberg - there's a lot more to this library than meets the eye. Tune in next week for the advanced course, when I'll be exploring things like transactions, cached queries, data typing and dynamic menus. Until then...be good!

Note: All examples in this article have been tested on Linux/i586 with PHP 4.2.0, Apache 1.3.12 and ADODB 2.2.0. 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!
