melonfire! **Community**

**Trog**

# PHP Application Development With ADODB (part 2)

**Find out how ADODB can be used to optimize multiple-run queries, commit and roll back transactions, and improve performance by caching query results.**

## Moving On

In the first part of this article, I introduced you to the ADODB database abstraction library, and showed you a little of how it works. I demonstrated how using it in your PHP application development could substantially reduce the time spent on code rewrites if your RDBMS decided to change shape, and also gave you a crash course in the basic functions built into the library.

Fortunately, that isn't all she wrote. ADODB comes with a whole bunch of bells and whistles, which allow you to do some fairly nifty new things in your PHP scripts. Over the next few pages, I'll be showing you some of them - so flip the page, and let's get started!

## Rapid Execution

In the event that you need to execute a particular query multiple times with different values - for example, a series of INSERT statements - the ADODB class comes with two methods that can save you a huge amount of time and also reduce overhead. 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!");

// prepare query
$query = $db->Prepare("INSERT INTO library (title, author) VALUES (?, ?)");
```

```
// read title-author list in from CSV file
$data = file("list.txt");

// iterate through each line in file
foreach ($data as $l)
{
   // split on comma
   $arr = explode(",", $l);
   // insert values into prepared query
   $result = $db->Execute($query, array($arr[0], $arr[1])) or die("Error in query: $query.
" . $db->ErrorMsg());
}

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

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 two placeholders used in the query string passed to Prepare() - these placeholders are replaced by actual values each time Execute() runs on the prepared statement. The second argument to Execute() is a PHP array containing the values to be substituted in the query string.

It should be noted that using Prepare() can provide performance benefits when you have a single query to be executed a large number of times with different values. However, this benefit is only available to you if your database system supports prepared queries (MySQL does not at this time, although Interbase and Oracle do); in all other cases, only simulated functionality is available and Prepare() becomes equivalent to a simple Execute(), with no inherent performance gain.

## A Fear Of Commitment

If your database system supports transactions (MySQL doesn't, but quite a few others do), you'll be pleased to hear that ADODB allows you to transparently use this feature in your scripts.

The following example 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!");

// turn off auto-commit
// begin transaction block
$db->BeginTrans();
```

```
// first query
$query = "INSERT INTO library (title, author) VALUES ('Title A', 'Author B')";
$result = $db->Execute($query) or die("Error in query: $query. " . $db->ErrorMsg());

// use ID from first query in second query
if ($result)
{
   $id = $db->Insert_ID();
   $query = "INSERT INTO purchase_info (id, price) VALUES ($id, 'USD 39.99')";
   $result = $db->Execute($query) or die("Error in query: $query. " . $db->ErrorMsg());
}

// if no failures
if ($result)
{
   // commit
   $db->CommitTrans();
}
// else rollback
else
{
   $db->RollbackTrans();
}

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

The first step here is to turn off auto-committal of data to the database, via the BeginTrans() method; this method also marks the beginning of a transaction block, one which can be ended by either CommitTrans() or RollbackTrans(). Once auto-commit has been turned off, 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.

Every call to Execute() within the transaction block returns either a true or false value, depending on whether or not the query was successful. These values can be tracked, and used to determine whether or not the entire transaction should be committed. Once you're sure that all is well, you can save your data to the database via a call to the CommitTrans() method. In the event that you realize you made a mistake, you can rewind gracefully with the RollbackTrans() function.

## Cache Cow

One of the coolest things about ADODB has to be its support for cached queries. Why? Because caching your queries can result in a fairly significant performance improvement, especially if you're executing the same tired old SELECT every time.

In order to illustrate the difference, let's take a look at how this normally works:

```
<?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 should be familiar to you by now - it's a very basic SQL SELECT operation with ADODB. If this was your personal Web site, and you were getting 5000 hits a minute, you'd be running the query above 30,000 times an hour. As you might imagine, this will have your database server scurrying around like a hamster on cocaine - not to mention affecting the performance of your Web site.

ADODB offers a better option - caching the results of the first SELECT query, and using this cached resultset in each subsequent run of the query. This reduces the load on the database server, and can also provide you with an incremental performance benefit.

Here's what the revised script looks like:

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

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

// set cache location
$ADODB_CACHE_DIR = '.';

// 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 = $db->CacheExecute(300,$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();

?>
```

The first argument to CacheExecute() is the number of seconds to cache the query results; the second is, obviously, the query string itself. The remainder of the script remains unchanged - a cached resultset is processed in exactly the same manner as a non-cached one.

You can also use the CacheFlush() method to flush all queries from the cache.

## What's On The Menu?

ADODB also comes with a couple of methods designed specifically for common Web development tasks. One of the most useful is the GetMenu() method, which retrieves and iterates over a resultset, and uses it to automatically build a form drop-down list containing the database records. This comes in very handy for dynamically-generated forms, when the items in the various form listboxes have to be dynamically built from a database.

Here's an example of how it works:

```
<html>
<head></head>
<body>
<?php
// 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 title, id FROM library";
$result = $db->Execute($query) or die("Error in query: $query. " . $db->ErrorMsg());

// print HTML menu
print $result->GetMenu("library", '', false);

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

?>
</body>
</html>
```

The GetMenu() method takes a number of arguments, which can be used to control the behaviour of the generated list box. The first argument is the name for the list ("library", in this case); the second is the default value for the list; the third lets you specify whether the first item in the list should be empty; and the fourth lets you control whether or not the list allows multiple selection.

Here's the HTML code generated by the script above:

```
<select name="library" >
<option value="15">Mystic River</option>
<option value="16">Where Eagles Dare</option>
<option value="17">XML and PHP</option>
</select>
```

As you can see, the contents of the list box are built from the resultset returned by the query; the first column of the resultset becomes the label for each list item, while the second is the corresponding value.

The GetMenu() method can simplify the task of developing a Web form substantially, significantly reducing the amount of code you have to write - consider using it the next time you need to build a list box from the records in a database.

## A Rose By Any Other Name...

ADODB also allows you to export a resultset into a variety of different formats - comma-separated text, tab-separated text, or even an HTML table. These functions are not part of the ADODB class per se; rather, they are packaged as ancillary functions in a separate file, which needs to include()-d in your scripts. The following example demonstrates:

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

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

// include conversion functions
include("toexport.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 title, id FROM library";
$result = $db->Execute($query) or die("Error in query: $query. " . $db->ErrorMsg());

// return a CSV string
echo rs2csv($result);

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

?>
```

Here's the output:

```
title,id
Mystic River,15
Where Eagles Dare,16
XML and PHP,17
```

You can suppress the first line - the column list - by adding an extra argument to the call to rs2csv(), like this:

```
<?php

// snip

// return a CSV string
echo rs2csv($result, false);

?>
```

And here's the revised output:

```
Mystic River,15
Where Eagles Dare,16
XML and PHP,17
```

You can format the data as a tab-separated string with the rs2tab() function,

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

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

// include conversion functions
include("toexport.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 title, id FROM library";
$result = $db->Execute($query) or die("Error in query: $query. " . $db->ErrorMsg());

// return a tab-separated string
echo rs2tab($result);

// close database connection
```

```
$db->Close();

?>
```

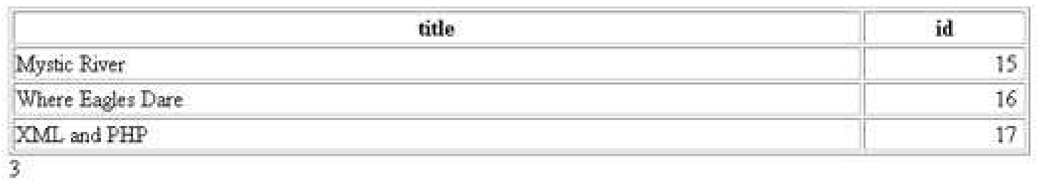
which returns the following output:

```
title	id
Mystic River	15
Where Eagles Dare	16
XML and PHP	17
```

or as an HTML table with the rs2html() function,

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

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

// include conversion functions
include("tohtml.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 title, id FROM library";
$result = $db->Execute($query) or die("Error in query: $query. " . $db->ErrorMsg());

// return a table
echo rs2html($result);

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

?>
</body>
</html>
```

which looks like this:



| title | id |
|---|---|
| Mystic River | 15 |
| Where Eagles Dare | 16 |
| XML and PHP | 17 |

3

A number of other interesting conversion functions are also shipped with ADODB - take a look at the documentation for more information.

## The Final Countdown

And that's about it for the moment. Over the course of this two-part article, I introduced you to the ADODB database abstraction class and demonstrated how it could be used in your Web development efforts. I showed you the fundamentals - executing queries, iterating over resultsets, obtaining table and row information - and then moved on to more advanced material, illustrating how ADODB could be used to optimize multiple-run queries, commit and roll back transactions, improve performance by caching query results, and automatically write HTML (or text) files.

That isn't all, though - ADODB comes with a wealth of features, and is constantly being improved by its author, and the PHP community at large. For more information on what it can do, and how you can use it in your own PHP projects, take a look at the following links:

The ADODB home page, at http://php.weblogs.com/ADODB

The ADODB manual, at http://php.weblogs.com/ADOdb_manual

MySQL and ADODB, at http://php.weblogs.com/adodb_tutorial

Tips on writing portable SQL, at http://php.weblogs.com/portable_sql

Web services with ADODB, at http://php.weblogs.com/adodb_csv

As for me, I'm outta here. See you soon!

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!