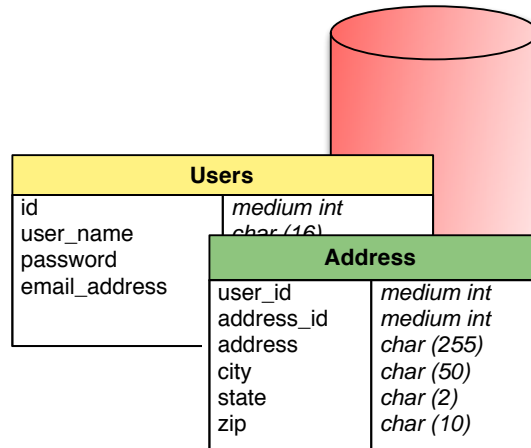


DBClass

An Object Oriented Class For PHP To Ease Database Access



Version 4.2 / 1 January 2006
Michael Heuss

Foreward

It was necessity that drove me to start developing DBClass. The year was 2000, and I was knee deep in my first major web application. I had to create an event management program that would allow an event management company to interface with attendees, airlines, the corporate client and hotels. It would run with on Linux with a MySQL database, but the in the first year was expected to run on Windows box connected to separate Microsoft SQL Server .

The deadline for delivery the product was a scant 6 months away.

I decided on PHP - it was enough like C that I felt comfortable getting started with it. However, the database functions would have to go. I foresaw problems in the future when we switched to a different database platform. The functions themselves decreased code readability. The differences in function formatting increased the risk of errors. Too much work would have to be done managing and working with the result sets in each page of code. I needed a better way.

So I set about creating a more object oriented approach to database access.

DBClass is the result of that effort. I have endeavored to increase code readability, to reduce errors, to improve debugging, and to speed development. I believe this class does all of that.

I know that most programmers using PHP have their own collection of functions or classes for accomplishing the same thing. I humbly submit this class to you in hopes that you find it useful.

Purpose Of This Overview

This overview is meant to simply expose you to the DBClass. It does not cover every method available to the programmer, and does not even cover all of the optional arguments available to each method.

In other words - This is not the reference documentation - That can be found on the mikeheuss.com website or with the download distribution file you have in your possession. If, after having read through this, you understand how to start using DBClass in your application, I will have hit my target.

Getting Started

At its most basic, connecting to a database is as simple as deriving the object with a few arguments. In the example below, we connect to a database and print out the user name of the users whose username starts with a 'm'.

```
$db = new DBClass (0, "user", "password", "database",  
    "localhost"); //Connect to database  
  
$db->saveQuery("select username from users where username like 'm%'");  
  
while ($db->getData())  
    print ($db->Get('username'));
```

At this point we have simplified connecting to the database, but not much else. We're still writing SQL.

By just adding a few more arguments to the constructor, we open up a little more possibilities.

```
$db = new DBClass (0, "user", "password", "database",  
    "localhost", "users", "id");  
  
$db->autoGet("username", "username like 'm%'");  
  
while ($db->GetData())  
    print ($db->Get('username'));
```

The sql is beginning to disappear, right? The code looks a bit cleaner, it is a bit easier to read. Now, lets create class that kind of groups together all sql relating interaction with the users table. We'll call it users class.

```
class UsersClass extends DBClass {  
  
function UsersClass ($arg_db=0)  
{  
  
    //Connect to the database  
    $this->DBClass ($arg_db, "user_name", "password", "database",  
        "localhost", "users", "id");  
  
    //Turn on SQL Tracing  
    $this->enableSQLTrace (true);  
  
    //Define a relationship it has to the addresses table  
    $this->autoDefineRelationships ("id", "address", "user_id",  
        DBCLASS_ONE_TO_MANY);  
  
}  
  
} //End of UsersClass  
  
$users_db = new UsersClass();
```

With the functionality of DBClass behind it, right now users_db has methods to add, insert, edit and delete data from the users table. It can perform cascading deletes to the addresses table when a users row is deleted. It can import data from a text file. It can aid debugging by printing out every SQL executed, and the time it took to do each one. It has a few more tricks up its sleeve, as well.

Sound like anything you are interested in? If so, read on.

Selecting Data

Let's stick to the user's table example we were working with in the last section. Here is what our users table looks like:

Users	
id	<i>medium int</i>
user_name	<i>char (16)</i>
password	<i>char (16)</i>
email_address	<i>char (50)</i>

Let's start with accessing the row of data for user 100. We want to find out what his email address is.

Here is how you can do that with the UsersClass we derived from DBClass earlier.

```
if( $users_db->autoLoadByKey (100))
  echo $users_db->get('email_address');
```

autoLoadByKey() is a function, that when given a value, attempts to load the row of data whose primary key corresponds to that value.

get() is used to retrieve data from the current row of data loaded into the _props array of the class. This loading was accomplished by the autoLoadByKey. If we wanted to also retrieve the password for user 100, we just have to add another get().

```
if( $users_db->autoLoadByKey (100))
  echo $users_db->get('email_address')." ".$users_db->get('password');
```

But what if you wanted to get more than one row of data? Let's say we want to select every user who has an AOL email address, so we can target them with spam mocking their choice of ISP. Let's sort that result set by email_address. Here's how we'd do that:

```

$users_db->autoGet("*", "email_address like '%aol.com'", "email_address");
while ($users_db->getData()) {
    echo $users_db->Get('email_address');
} //End of loop

```

autoGet() selects data from the database, but does not load that data into the `_props` array.

getData() is used to bring a record from the recordset into the `_props` array, and at that point get() can be used to retrieve it. Each subsequent call to getData erases the old information in the `_props` array, and loads the next row in the recordset.

Note: Member functions that contain the word Load are used to bring a row of data into the `_props` array, thus allowing immediate access using get(). Functions containing Get, however, don't. They are intended for multi row record set, and wait for getData to load the `_props` array.

Updating Data

Let's change the password for user 100. His wants the password changed to "raven", because he loved Edgar Allan Poe.

Here we go:

```

if ($users_db->autoLoadByKey(100)) {
    $users_db->put('password', 'raven');
    $users_db->autoSave();
}

```

put() populates the column specified in the first argument with the value specified in the second.

autoSave() determines whether data is being inserted or updated, then creates & executes the appropriate SQL

Inserting Data

What if we want to add two new records? That also is pretty simplistic:

```

$users_db->Put ('user_name', 'bobjones');
$users_db->Put ('password', 'bobathy');
$users_db->Put ('email_address', 'bobjones@example.com');
$insert_id[] = $users_db->autoSave();
$users_db->blank(); //Erases the current contents in the _props array
$users_db->Put ('user_name', 'randy');

```

```
$users_db->Put ('password', 'ballw');  
$users_db->Put ('email_address', 'randy@example.com');  
$insert_id[] = $users_db->autoSave();
```

Now, we have created two new users, bobjones and randy.

`insert_id[0]` contains the primary key for the first insert. `insert_id[1]` contains the primary key value for the second user.

`blank()` clears the `_props` array of data from the first insert, ensuring none of bobjones info gets saved to randy's record.

Deleting Data

Deleting data is also fairly simple. Here are three quick deletes:

```
$rows = $users_db->autoPrimaryDelete (100);  
$rows = $users_db->autoDelete ("email_address", "mike@rapidstability.com");  
$rows = $users_db->autoCascadingDelete ("users.id=100");
```

All three functions return the number of rows deleted.

`autoPrimaryDelete()` gets rid of the record whose primary key is equal to 100.

`autoDelete()` gets rid of all records who have a column matching a given value - in this case, an `email_address` of mike@rapidstability.com.

`autoCascadingDelete()` gets rid of users whose id is 100, as well as all address records that belong to user 100. The relationship was defined in the constructor for the users class. (It's alright. Go back and look,) More on those relationships later.

Debugging

There are two useful debugging concepts in `DBClass`. The first, and simplest, can be executed at any time. It simply echoes the query being executed to the screen.

```
$users_db->echoQuery(true);  
$users_db->autoPrimaryDelete(100);  
$users_db->echoQuery(false);
```

When executed, you'll see the following on your screen:

```
delete from users where id='100';  
Rows Affected: 1  
By: ::autoPrimaryDelete
```

Every query executed by this object between the points where `echoQuery` is turned on and turned off will be echoed to the screen.

The other one is a little more complex. It actually captures the time it took for the SQL to execute. It was enabled in the users class in the constructor, when we executed `enableSQLTrace()`;

Here we go:

```
//Lets execute a few sql statements - each using different methods
$users_db->doQuery("select nickname from users where id=1001");
$users_db->saveQuery("select id from users where user_name='mike'");
$data = $users_db->quickRow("select id from users where ".
    "user_name like 'mike%'");
//Now, let's retrieve the results
$results = $db->retrieveAllSQL();

//And lastly, we'll display the results
for ($cnt=1; $cnt<=count($results); $cnt++) {
    echo "<p>sql: ".$results[$cnt]['sql']."</p>";
    echo "<p>ts: ".$results[$cnt]['ts']."</p>";
    echo "<p>affected_rows: ".$results[$cnt]['affected_rows']."</p>";
    echo "<p>exec_method: ".$results[$cnt]['exec_method']."</p>";
    echo "<p>microtime_start: ".$results[$cnt]['microtime_start']."</p>";
    echo "<p>microtime_finish: ".$results[$cnt]['microtime_finish']."</p>";
    echo "<hr>";
}
}
```

`doQuery()` is a quick and dirty way to specify sql to be executed. No result set is saved, it is just run.

`saveQuery()` is another quick and dirty way to specify sql, but the result set is saved, and `getData` can be used to retrieve results.

`quickRow()` is a simple query that when executed immediately returns the results as an array. In the example above `$row[0]` contains the id for the first user whose name is like mike. The result set is not retained, and calls to `getData` will not return the following rows.

`retrieveAllSQL()` returns an array containing the attributes of every SQL executed through the object. The attributes returned are: the sql, the timestamp of when the sql was executed, the number of rows affected by the sql, the method causing the sql to be executed, the microtime the sql started, and the microtime it finished.

The above example's output would be as follows:

```
sql: select nickname from users where id=1001
ts: 1136588373
affected_rows: 1
exec_method: ::DoQuery
microtime_start: 0.19002300 1136588373
microtime_finish: 0.19143900 1136588373

sql: select id from users where nickname='mike'
ts: 1136588373
affected_rows: 1
exec_method: ::saveQuery
microtime_start: 0.19156000 1136588373
microtime_finish: 0.19300700 1136588373

sql: select id from users where nickname like 'mike%'
ts: 1136588373
affected_rows: 290
exec_method: ::quickRow
microtime_start: 0.19312000 1136588373
microtime_finish: 0.19933400 1136588373
```

Joins

Remember that relationship we defined in the constructor? We can use that to quickly and simply pull data from both tables.

Here is what the address table looks like:

Address	
user_id	medium int
address_id	medium int
address	char (255)
city	char (50)
state	char (2)
zip	char (10)

The following example pulls the user name from the users table, and the address information from the address table.

```
$users_db->autoGetJoinedTables ("users.user_name, address.address, address.city,
address.state, address.zip", "address");

echo "We are retrieving ".$users_db->getNumberAffectedRows()." rows";

while ($users_db->getData()) {

    echo $users_db->Get('user_name')."<br>".
        $users_db->Get('address')."<br>".
```

```

        $users_db->Get('city')." ". $users_db->Get('state').
        " ". $users_db->Get('zip');
    }

```

`autoGetJoined()` retrieves data based upon the relationship defined with `autoDefineRelationships()`.

`getNumberAffectedRows()` returns the number of rows affected by the last sql query.

The Constructor

The constructor has eight parameters, of which only four need to be specified. Lets take a closer look at them. Here is the definition from `DBClass`:

```

function DBClass ($arg_db,
                 $arg_user,
                 $arg_pwd,
                 $arg_dbname,
                 $arg_host="localhost",
                 $arg_tablename="",
                 $arg_primarykey="",
                 $arg_dbtype="mysql")

```

The first argument, `$arg_db`, is the database connector id that is used by another `DBClass` object. The idea behind this is that multiple `DBClass` objects can share the same connection to the database. Let me give you an example of this parameter in use:

```

$db = new DBClass (0, "user", 'password', 'database');
$second_db = new DBClass ($db->getDB(), 'user', 'password', 'database');

```

The `$second_db` object makes use of `$db`'s connection to the database.

The `getDB()` function is used to get `$db`'s database connection.

The `user`, `password`, `dbname` and `host` arguments should be self explanatory. The `tablename` argument is used to define the table this object is going to be concerned with, and the `primarykey` id is the primary key of the table. Yes, `primarykey` could have been found programmatically, but it seemed just as simple to me to let the programmer specify it.

The last parameter, the `dbtype`, is used to tell `PEAR::DB` which database we are trying to access.

One handy function to place in your derived classes constructor is `setSequenceName`. This tells the `PEAR::DB` structure what the table name for your sequence is. This may be omitted, and `PEAR::DB` will automatically use `table_name_seq`.

Once the constructor has executed, it is possible to switch to a different database and table. The caveat is that the same username, password and host must be maintained.

```

$db->switchDB("second_database");
$db->defineTable ("different_table", "different_primary_key");

```

switchDB() is the function that actually does the work of switching.

defineTable() can be used to specify the table and primary key.

More Ways To Get Data

Quick and dirty way to load a single record

I want to load the user whose email address is mike@rapidstability.com.

```
$users_db = new UsersClass();  
  
if ($users_db->autoLoad ("*", "email_address='mike@rapidstability.com'"))  
    echo $users_db->Get('user_name').  
        " has the email address I am looking for";
```

autoLoad() selects a record from the table defined in the constructor or with the defineTable() command and loads it into the props array.

This also could have been done with autoLoadByColumn().

```
if ($users_db->autoLoadByColumn ('email_address', 'mike@rapidstability.com'))  
    echo $users_db->Get('user_name').  
        " has the email address I am looking for";
```

The last way to skin this cat is to use autoQuickRow(). Here we go again:

```
$row = $users_db->autoLoadByColumn ('email_address',  
'email_address='mike@rapidstability.com'))  
  
if (strlen($row[0]))  
    echo $row[0]. " has the email address I am looking for";
```

The purpose of the QuickRow & autoQuickRow functions is that the data is not loaded into the props array, it is directly returned.

This is very useful if you are trying to get a value from the table while in the middle of processing a record set.

Getting data into an array

Let's say that you want to get the name and abbreviation of every state into an array called \$states, for use in creating a multitude of select boxes on a form. You have a table called states, and you have even derived a StatesClass for interfacing with that table.

Here is what is necessary to accomplish this:

```
$states = new StatesClass();  
  
//Get the states into the array  
  
$states->autoGetArray ("state_name", "abbreviation");  
  
//Print out the states in a select  
  
echo "<select name=states>";
```

```

while (list ($key, $value) = each ($results)) {
    echo "<option value=\""$value\"">$key</option>";
}
echo "</select>";

```

autoGetArray() returns portions of recordset in an array.

Getting random records from a database

I had to add a quote of the day interface in a clients website. She wanted the quote to rotate on a daily basis.

autoGetRandom accomplishes this.

```

$quote_db = new QuoteClass(); //Derived from DBClass
$quote_db->autoGetRandom ("daily");
echo $quote_db->Get('quote')."<br><b>".$quote_db->Get('source')."</b>";

```

Monthly, hourly, and random are also possible values.

More Ways To Delete Data

More Control

autoComplexDelete allows you to specify the where criteria for the delete directly. For instance, let's delete all the aol users out of the users table.

```

$users_db = new UsersClass();
$users_db->autoComplexDelete ("email_address like '%aol.com'");
echo "Deleted ".$users_db->getNumberAffectedRows()." rows";

```

Caching

Current Data

If you ever need to cache the data in the _props array to be retrieved at a later time, then cacheData/ restoreData is the method to use.

These functions don't cache the result set, but simply what is in the _props array.

Here is an example of their use.

```

$users_db = new UsersClass();
$users_db->autoLoadByKey (100);
echo $users_db->get('user_name'); //outputs 'mike'
$users_db->cacheData();

```

```

$users_db->autoLoadByKey(200);

echo $users_db->get('user_name'); //outputs 'betty'

$users_db->restoreData();

echo $users_db->get('user_name'); //outputs 'mike'

```

Result Sets

If you need to interrupt the current result set with another one, but want to resume the current one later, `cacheResults/restoreResults` is for you.

```

$db = new DBClass(0, 'user', 'password', 'database', 'table');

$db->saveQuery ("select * from users");

while ($db->getData()) {

    echo $db->Get('first_name')." ".$db->Get('last_name');

    if ($db->Get('paychecks')>0) {

        $db->cacheResults();

        $db->saveQuery ("select sum(amount) as tl from paychecks where ".
            " user_id='". $db->Get('id')."");

        $db->getData();

        echo " earned ".$db->Get('tl')." this year";

        $db->restoreResults();

    }

}

```

In this example, we get all of the users. Any who have more than one paycheck, we then go get the sum of their paychecks. `cacheResults` and `restoreResults` allows us to stop and resume the first recordset as necessary.

Inserting/Updating Data Roundup

`autoInsert()` is a quick function call that will take the values specified by the `put()` method and attempt to insert a record into the table. It has a parameter that will, if set to true, cause the SQL to be returned as a string instead of being executed.

The same goes for `autoUpdate()`. The only difference between the two is `autoUpdate` will attempt to update existing records.

Extended inserts are faster. So we have two functions that will get your extended inserts going. Here is an example of them in use.

```

require_once "DBClass.php";

$db = new DBClass (0, "test_user", "test_password", "test_database",
                  "localhost", "users", "id");

$db->put ('username', 'mikeheuss');

$db->putHash ('password', 'my_password');

$db->autoInsertExtended();

$db->put ('username', 'next_user');

$db->putHash ('password', 'his_password');

$db->autoInsertExtended();

//This command will execute the cached sql statements

$num_rows = $db->autoInsertExecute();

print ("<P>We just inserted ".
      $num_rows.
      " into the table 'users'</p>");

```

`autoInsertExtended()` caches the SQL to be executed at a later time.

`autoInsertExecute()` causes that cache SQL to be executed.

The last insert function to cover is `importDataFromArray()`. This function takes an array and attempts to match it up to the table. It then attempts to insert the matches columns as a row of data. This makes importing data from text files fairly easy. The `arg_array` parameter simply need to follow the layout of the file.

Data Manipulation Functions Roundup

There are a multitude of little functions useful for operating on the data retrieved from a recordset.

`convertTimestamp()` takes a timestamp in YYYYMMDDHHIISS format and returns a unix timestamp. This is handy when working with PHP date functions.

`prettyDate()` takes a PHP DATE format string and returns a specified columns date in that format.

`prettyDateTimestamp()` does the same thing for Timestamp columns.

`nonBreakGet()` returns a column with all spaces replaced with ` `;

`getStrLength()` returns the length of the data in that column.

If `loadTableStructure()` has been called, `getColumnInformation` will return all kinds of great information about the makeup of the specified column.

`putHash()` uses the `md5()` PHP function to hash the data before putting it into the props array for the specified column. Handy for passwords.

Of course, if you have a `putHash`, you need a `compareHash()`. This will tell you if the data provided as an argument matches the hash already in the column specified.