A Database Wrapper Mechanism for Server-Side HTML-Embedded Scripting

William J. McIver, Jr.
University at Albany, State University of New York
135 Western Avenue
Albany, New York 12222 USA
(1) 518.442.5123

McIver@albany.edu

ABSTRACT
This paper outlines techniques for implementing database wrapper mechanisms using PHP and MySQL. These techniques solve complexity and redundancy problems associated with the development of HTML form-based applications that are database-driven. The techniques demonstrated here are generalizable to other PHP database API bindings.

Categories and Subject Descriptors
H.2. [**Database Management**]: Languages, Database Application H.5.3 [**Group and Organizational Interfaces**]: Web-based interaction

General Terms
Management, Design, Languages

Keywords
PHP, MySQL, database wrappers

1. INTRODUCTION
A common approach to developing forms-based, database-centric applications for the Web is to use PHP in combination with MySQL [2]. PHP is a general-purpose scripting language that can be embedded in HTML for server-side processing [1] and MySQL is arguably the world s most commonly used open source database management system [3]. Both technologies are highly complementary and provide a relatively convenient development platform.

While these technologies are complementary, PHP/MySQL implementations can easily become tedious and redundant as system requirements become more complex. At a conceptual level, PHP s processing model makes it convenient to transfer data between HTML forms and the database application logic. For robust applications, however, the tedium arises in that each data element captured from an HTML form must be validated for content and format, confirmed with the user, mapped and inserted into the appropriate underlying database table, and possibly returned to the user interface (i.e. browser) as a result. The redundancy arises in that these processes are generally the same for most form-to-database interactions. This tedium and redundancy brings with them the risk of software errors. A PHP-based approach is, therefore, necessary to automate these tasks.

This paper describes a PHP class implementation that provides convenient and effective solutions to these problems and, thereby, enables rapid development of database-supported applications with HTML-client interfaces.

2. BACKGROUND
PHP and MySQL are highly complementary technologies. PHP s processing model and bindings handle all of the fundamental technical problems inherent in forms-based, database-centric applications for the Web: the capture of data entered using HTML forms, interaction with underlying database management systems (DBMS) (e.g. MySQL), and the generation of dynamic markup.

PHP s processing model establishes convenient associative relationships between <INPUT> element *name* attributes in HTML forms and PHP variables. The HTML form in Figure 1 below

```
<FORM METHOD= POST
        ACTION= <?php echo $PHP_SELF?> >
<INPUT TYPE= text  NAME= first_name >
<INPUT TYPE= submit
        VALUE= Click here to Submit >
</FORM>
```

Figure 1. Submitting input from an HTML form to a PHP script.

will result in the any value that is entered into the text INPUT element named first_name to be transferred (i.e. posted) to the PHP variable $first_name when the document containing this form is re-processed. Any such value can then be used to construct any SQL command, such as INSERT, SELECT, or DELETE. This is done using PHP s string-processing features and the MySQL API that is standard in PHP. For example, the input from form above might be processed as follows:

```
<?php  if ($submit){ // The submit button was clicked.
    $db_link = mysql_connect( ) or
    die("<br>ERROR connect failed.\n");
     $status = mysql_select_db("<db_name>",$db_link);
    $query = "INSERT INTO students_table
            VALUES('$first_name')";
     $result = mysql_query($query,$db_link);

    // Generate HTML output
?>
```

Figure 2. Processing the input from an HTML form.

In Figure 2 above, the data entered into the INPUT element name= first_name in Figure 1 is captured in the PHP variable $first_name, inserted into an SQL INSERT command and submitted to a MySQL database server.

Figures 1 and 2 make clear the power of HTML-based PHP/MySQL applications. What is also clear, however, is that as the complexity of the underlying database grows so too will the complexity of both the HTML form specifications and the PHP code to construct database interactions. If, for example, we wish to add more database attributes the *students* table, so too must we add code to capture these values from HTML form INPUT elements and, correspondingly, perform additional string processing to insert them into SQL commands.

The real complexity comes when attempting to implement robust applications where data must be validated and confirmed with the user prior to entry into the underlying database. In addition, if an application is to interact with multiple database servers or tables, then these operations must generally be implemented in a redundant fashion using static code.

The specific requirements for PHP/MySQL implementations in developing robust HTML form-based applications include following tasks:

1. Checking that data entered into HTML INPUT elements are the appropriate lengths for their associated attributes in the associated underlying database tables;
2. Validating the data entered into HTML INPUT elements according to the desired formats, such as numeric-only data, telephone numbers or postal codes;
3. Ensuring that values for attributes database tables that are declared NOT NULL are present prior to insertion;
4. Providing a convenient processing model for correcting data that fail any of the tests above; and
5. Providing a parameterizable approach to constructing SQL commands based on data input into HTML INPUT elements.

3. IMPLEMENTATION

This implementation is modeled in part on the concept of extending PHP *echo* function to provide specialized and context-dependent functions for generating HTML. As will be shown, these include functions for echoing HTML INPUT element specifications for HTML FORM elements. In addition, the echo extensions construct INPUT element names in a way that corresponds directly with underlying database attribute names and in a way that allows all of the values in a database table to be captured automatically by a PHP object method once the form is submitted. Finally, the implementation allows the parameterized generation of SQL commands such as INSERT and SELECT based on INPUT elements configurations of any complexity. This implementation centers on a PHP class called Table.

WWW 2003, May 20-24, 2003, Budapest, Hungary.
ACM xxx.

3.1 The *Table* Class

The Table class is implemented using PHP s object model. Using the *Table* class, objects are instantiated for each MySQL table to be used in a PHP/MySQL application. Each *Table* object encapsulates the following information:

6. The hostname, user id, password, database name, and table name;
7. Names and data types for each attribute in the database table;
8. Data constraint specifications; and
9. Labels, messages, and colors to be associated with the attribute in an HTML form.

The hostname, user id, password, database name and table name allow the automatic construction of code for necessary for connecting with a database associated with the HTML form representation of a table. The data type and constraint specifications allow the generation of PHP code needed to validate data that are input into HTML forms to minimize database errors and inconsistencies. The labels, messages and colors specifications allow for the convenient generation of HTML forms. The general model is, thus, that of wrapping each MySQL table with a PHP object. Each aspect of the class is discussed below. In addition, the processing model required for use of the *Table* class is discussed.

3.2 Wrapping MySQL Tables

One Table object is instantiated for each MySQL table to be manipulated in a PHP/MySQL application. Our running example assumes the existence of the following MySQL tables:

```
CREATE TABLE students (
    student_id INTEGER NOT NULL AUTO_INCREMENT
                PRIMARY KEY,
    first_name VARCHAR(16) NOT NULL,
    last_name  VARCHAR(16) NOT NULL,
    phone      VARCHAR(10),
    class      VARCHAR(16),
    credits    integer,
    comments   VARCHAR(64));
CREATE TABLE class_values (class VARCHAR(16));
```

In our application, we will use the table class_values only for referential integrity and not for direct interaction from the browser. The *class_values* table contains the values *{ freshman , sophomore , junior , senior , masters , phd , non-degree }*. Thus, we instantiate a table object for the *students* table only. This is done as follows:

```
$students = new  Table("students","localhost",
                       "<uid>","<passwd>",
                       "<databasename>");
```

This instantiation establishes the class variables for the table. To this, specifications for each attribute in the table are then added to the object. These specifications are used for two main purposes: to direct the generation of HTML INPUT element specification and to guide the validation and processing of

data that are received from the INPUT elements once an HTML form is submitted. For example, the data will be tested to ensure that their lengths satisfy the length constraints specified in the object and that they satisfy both basic data type format constraints (i.e. numeric only or alpha numeric) and higher-level data format constraints (if any) (e.g. telephone number). Attributes are added to the students table object as follows:

```
$students->add_attribute("student_id",$dt_numeric,
                $auto_increment, $null, 4,
                 "Student Id",$black,$null,$red,$it_textfield,
                $constraint_num);
$students->add_attribute("first_name",$dt_string,
                $required,$null,64,
                 "First name",$red,$null,$red,$it_textfield,
                $constraint_none);
$students->add_attribute("last_name",$dt_string,
                $required,$null,64,
                 "Last name",$red,$null,$red,$it_textfield,
                 $constraint_none);
$students->add_attribute("phone",$dt_string,
                $not_required,$null,10,
                 "phone",$black,$null,$red,$it_textfield,
                $constraint_phone);
$students->add_attribute("class",$dt_numeric,
                $required,$null,10,
                 "class",$black,$null,$red,$it_select,
                $constraint_none,
               "class_values");
$students->add_attribute("credits",$dt_numeric,
                 $not_required,$null,10,
                  "credits",$black,$null,$red,$it_textfield,
                 $constraint_num);
$students->add_attribute("comments",$dt_string,
                  $not_required,$null,255,
                  "comments",$black,$null,$red,$it_textarea,
                 $constraint_none);
```

Each *add_attribute()* invocation also specifies the type of HTML form INPUT element to be used to enter data for the associated database attribute. Each attribute may also, optionally, be associated with a referential integrity constraint. For example, the class attribute in the *students* table will be entered using an INPUT element of type= select drawn from the values stored in the class_values table. This is indicated by the last argument in the *add_attribute()* invocation for the class attribute above.

3.3 Generating HTML INPUT elements.

Once the Table objects are instantiated and the attribute specifications have been added, the *echo_inputfield()* provided by the Table class can be used to automatically generate HTML INPUT elements. Examples of such invocations are as follows:

```
<form method=POST action="<?php echo $PHP_SELF?>">
<?php   $students->echo_inputfield("student_id");       ?>
<br>
<?php   $students->echo_inputfield("first_name");       ?>
<br>
<?php   $students->echo_inputfield("last_name");        ?>
<br>
<?php   $students->echo_inputfield("phone");            ?>
<br>
<?php   $students->echo_inputfield("class");            ?>
<br>
<?php   $students->echo_inputfield("credits");          ?>
<br>
<?php   $students->echo_inputfield("comments");         ?>

</form>
```

The resulting HTML form would appear as follows:

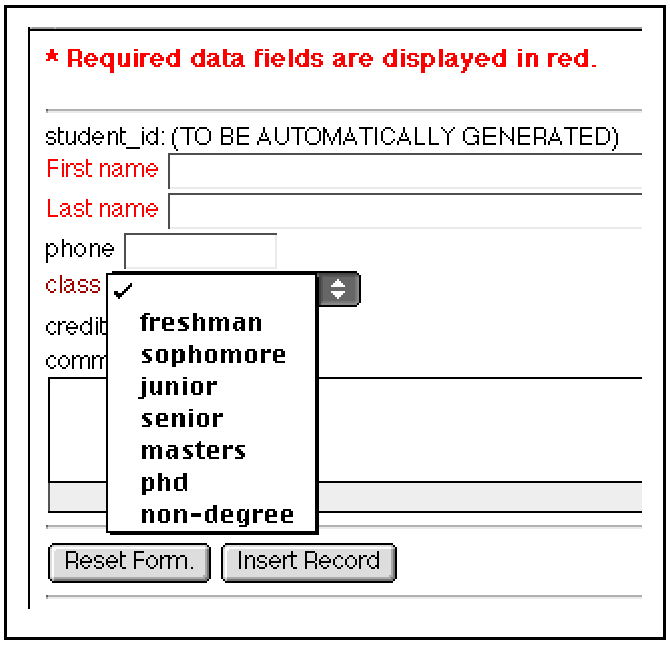

Figure 3. Generation of HTML form INPUT elements using the echo_inputfield() method.

3.4 Handling Input

The major issue is automating the assignment of INPUT element names with the corresponding attributes in the Table object after an HTML form has been submitted. This is done by:

1. Deriving the name of the INPUT element from the table attribute name in such a way that the corresponding

variables that are posted can be automatically matched to the attributes specified in the associated *Table* object.

2. Providing a method in the *Table* class that iterates through each database table attribute automatically capturing their associated INPUT element values.

Since a given PHP/MySQL application might be associated with multiple tables or multiple database servers, INPUT element name attributes must be assigned in a way that insures their uniqueness across all tables and MySQL server hosts. INPUT element attribute names are, thus, assigned within the *echo_inputfield()* method as follows:

```
$input_var_name =
   $table_name . "-in_" . $attr_name[$_attr_name];
```

The resulting HTML source appears as follows:

```
<input type=text
    name= students-in_first_name
   value=""
   size=64
   maxlength=64>
```

Once an HTML form is submitted all values that are posted are captured using the set_all_attributes() method. An example of this is as follows:

```
<?php
if ($submit) {
      $students->set_all_attributes();

?>
```

The effect of this invocation is to iterate through all of the database attributes that were declared for Table object students and make assignments to PHP variables constructed using the same technique as the *echo_inputfield()* method shown above. Thus, a PHP/MySQL developer is assured of an automated approach of capturing any and all INPUT element values corresponding to the database attributes they have declared.

3.5 Handling Constraints

The *set_all_attributes()* method has the additional responsibility of checking the various constraints declared for each attribute when they were inserted into a Table object using the *add_attribute()* method. If a database attribute was declared as NON NULL it should also be declared using the add_attribute() method using the $required constant. Additionally, if the database attribute was declared as a numeric values such as INTEGER, BIGINT, DOUBLE or FLOAT should be declared using the add_attribute() method using the $dt_numeric constant. Finally, the developer can specify specialized constraints such as telephone numbers using constants such as $constraint_phone. The latter type of constraints may need to implemented by the developer, but can be conveniently added to the *Table* class definition.

3.6 Confirming data that are entered

A critical aspect of forms-based applications is the validation of data that are entered. This is sometimes overlooked to the detriment of users and organizations alike. The *Table* class in this implementation allows this to be done conveniently by providing a PHP variable *$confirm* that serves as a flag that can be tested after an HTML form has been submitted. This flag is generated as part of the *echo_inputfield()* method invocation. More specialized flags are also set depending on context such as *$confirming_input* if a database INSERT might be attempted. Typical document processing flow of control logic might be as follows using these flags:

```
<?php
    // Document processing routing section.
   // Decide which part of this HTML document
   // should be processed.
   if ($insert) {
        $status = $confirming_insert;
        if (!$students->set_all_attributes()){
            $status = $data_entry_error;
       }// end if
   } else if ($confirm){
        if (!$students->set_all_attributes()){
            $status = $data_entry_error;
       } else {
            $insert_id = $students->db_insert();
           if (!$insert_id){  ?>
                <h3>Student Record Insert Failed.</h3>
               <hr>
                </body>
                </html>
<?php       // Stop parsing the script.
                die("");
           } else {
                $status = $inserted;
                // Go display the record.
           }// end if
       }// end if
   }// end if
?>
```

3.7 Handling Database INSERTS

As shown in the code above, the *Table* class also provides the method *db_insert()*. This method is responsible for automating the invocation of MySQL INSERT SQL commands. This is done within the *Table* class by iterating through all of the attributes declared for an object using the add_attribute() method and constructing an INSERT command as follows:

```
$insert_clause = "INSERT INTO $this->table_name ( ";
 $value_clause = " ) VALUES ( ";

  $count = 0;
  reset($this->attr_name);
  while (list($_attr_name) = each($this->attr_name)){
       if ($count > 0){
```

```
            $insert_clause .= ", ";
            $value_clause .= ", ";
        }// end if
        $insert_clause .= $_attr_name;
          $attr_value = $this->attr_value[$_attr_name];
        $value_clause .= "'$attr_value'";

        ++$count;
    }// end while
```

3.8 Handling AUTO_INCREMENT Attributes

In many cases, the developer will want to insert a tuple into a table and have one attribute value be generated by MySQL to ensure their uniqueness. This is done using the standard SQL AUTO_INCREMENT modifier for an attribute type when a table is created. See section 3.2 above.

Such AUTO_INCREMENT attributes are handled by the db_insert() method by assigning NULL values to the attributes in the VALUES clause of the INSERT command that it generates. Once the value is assigned by MySQL, it is then assigned by the script to the Table object for use in subsequent operations. This is done using the PHP/MySQL built-in function mysql_insert_id(), which returns the value assigned to an AUTO_INCREMENT attribute, if any, on the last SQL command.

The current version of the *Table* class handles only tables with single AUTO_INCREMENT attributes. MyISAM and BDB tables used in MySQL do allow multiple AUTO_INCREMENT attributes (See [MySQL]).

The db_insert() method returns the LAST INSERT ID for utility purposes nevertheless. It is required in many cases to pass the value of a PRIMARY KEY to a foreign key attribute managed by another PHP object and its underlying MySQL table.

3.9 Handling Referential Integrity Constraints

It is often the case, that a developer wants to ensure that only values from a fixed set are entered into particular database table attributes. This is done conceptually in the relational model using referential integrity constraints which specify value sources for data values. The sources are attributes in other tables. As in our example above, we intend for the class attribute in the students table to come from the class_values table from the attribute class. This is done using the *Table* class by specifying an optional *ref_table_name* in the *add_attribute()* method invocation and a corresponding $it_select INPUT element type.

The result is that an HTML drop down menu is produced when the corresponding echo_inputfield() is invoked for this type of attribute. The select INPUT element generated is automatically populated by values from the specified *ref_table_name*. One issue here is ensuring that no value from the referential integrity table is assumed to be the default. Thus, users are constrained to explicitly specifying only the values from the specified table, ensuring that the referential integrity constraint is met. This is done as follows in the Table class:

```
echo "<select name=$input_var_name>\n";
// This prevents the first value from the database
// from being a default value.
echo "<option>$null\n";

// Get results for the pulldown menu.
$result = mysql_query("SELECT * FROM $ref_table_name[$_attr_name]",$db_link);
while ($tuple = mysql_fetch_row($result)) {
        echo "<option value=\"$tuple[0]\"";

          // if inst_level was selected on the
          // previous pass, auto select it now.
        if (strcmp($attr_value[$_attr_name],$tuple[0]) == 0){
            echo " selected ";
        }
        echo ">$tuple[0]\n";
}
echo "</select>\n";
```

3.10 Debugging Generated Code

Each method in the *Table* class that generates HTML does so in a way that makes the HTML readable and indicating where method calls begin and end. This is critical for developing PHP/MySQL in an efficient and reliable way. Many HTML editors, such as Netscape Composer, and similar applications do not generate code that formatted in a way that is easy to read. For example, method invocations that generate HTML indicate in HTML comments the starting and ending points of their invocations. The following is an example:

```
<!-- Table::echo_inputfield -->
   [ .GENERATED HTML CODE HERE ]
<!-- end Table::echo_inputfield -->
```

4. EXAMPLE

The solution discussed in this paper addresses each of these tasks. The source code for this implementation is available using instructions at the URL http://www.albany.edu/~mciver/WWW2003.

The example demonstrates the basic approach to using the *Table* class. This example includes the browser interface within *students.php*, the class instantiation and configuration in *students.inc*, and the *Table* class definition in *dbwrapper.inc*.

Each application should define a PHP include file, which includes the file containing the *Table* class definition and constants, called *dbwrapper.inc*. It is necessary, as in C or C++, to insure that files are included only once. This is done in each include file using the following types of tests:

```
// Include the file which contains the definition
```

```
// for class "Table."
    if (!$dbwrapper_inc){
        require("dbwrapper.inc");
        $dbwrapper_inc = true;
}// end if
```

This application-specific object instantiation files is in-turn included into the file to be retrieved from the browser, in this case, students.php. Thus, students.php includes the following code:

```
<?php
    // Include the object instantiation and configuration code.
    include("students.inc");
?>
```

5. CONCLUSIONS

This paper demonstrates techniques for automating many of the tasks involved in developing PHP/MySQL applications that utilize HTML forms. The implementation makes use of a special PHP class, which is used to wrap each MySQL database table. The wrapper providing by this class handles the generation of HTML INPUT element fields, the automatic capture by PHP scripts of data transferred when HTML forms are submitted, and the generation of MySQL SQL commands, such as INSERT commands. The approaches demonstrated in this implementation enables rapid application development while reducing risks of implementation errors.

Future work in this area will include the development of a parser for the automatic generation of Table object instantiation code corresponding to CREATE TABLE statements for specified database implementations.

6. ACKNOWLEDGMENTS

This work was supported by the Scholarly Technology Group at Brown University. The author thanks, in particular, Elli Mylonas and Carole Mah.

7. REFERENCES

[1] Bakken. S.S. Introduction to PHP. April 17, 2000. http://www.zend.com/zend/art/intro.php

[2] Merrall, G. PHP/MySQL Tutorial. May 19, 1999. http://hotwired.lycos.com/webmonkey/programming/php/tutorials/tutorial4.html

[3] MySQL. http://www.mysql.com/