Perl

The Perl programming language has gone from a tool primary used by Unix systems administrators to the most widely used development platform for the World Wide Web. Perl was not designed for the web, but its ease of use and powerful text handling abilities have made it a natural for Web application development. Similarly MySQL, with its small footprint, speed and large feature set, has been very attractive to web developments that need to serve thousands of transactions a day. Therefore, it was only a natural that a Perl interface to MySQL was developed that allowed for the best of both worlds.

Note: At the time of this writing Perl has standardized on the DBI suite of modules for all database interaction, including MySQL. However, many legacy systems still use an older interface to MySQL called MySQL.pm. This module is not compatible with the DBI standard and is no longer actively developed. All new development should certainly use the standard DBI modules, and any sites using MySQL.pm should consider upgrading to DBI for any future development.

DBI

The recommended method for accessing MySQL databases from Perl is the DBD/DBI interface. DBD/DBI stands for DataBase Driver/DataBase Interface. The name arises from the two-layer implementation of the interface. At the bottom is the database driver layer. Here, modules exist for each type of database accessible from Perl. On top of these database dependent driver modules lies a database independent interface layer. This is the interface that you use to access the database. The advantage of this scheme is that the programmer only has to learn one API, the database interface layer. Every time a new database comes along, someone needs only to write a DBD module for it and it will be accessible to all DBD/DBI programmers.

As with all Perl modules, you must use the DBI to get access:

```
#!/usr/bin/perl

use strict;
use warnings;
use DBI;
```

When running and MySQL Perl programs, you should always include the 'use warnings' statement early in the script. With this present, DBI

will redirect all MySQL specific error messages to STDERR so that you can see any database errors without checking for them explicity in your program.

All interactions between Perl and MySQL are conducted through what is known as a database handle. The database handle is an object—represented as a scalar reference in Perl—that implements all of the methods used to communicate with the database. You may have as many database handles open at once as you wish. You are limited only by your system resources. The connect() method used a connection format of DBI:servertype:database:hostname:post (hostname and port are optional), with additional arguments of username and password to create a handle:

```
# We will use the variable name 'dbh' to indicate a database handle.
# This is a very common idiom among DBI users.
my $dbh = DBI->connect('DBI:mysql:mydata', undef, undef);
my $dbh = DBI->connect('DBI:mysql:mydata:myserver', undef, undef);
my $dbh = DBI->connect('DBI:mysql:mydata', 'me', 'mypass');
```

The servertype attribute is the name of the DBD database-specific module, which in our case will be 'mysql' (note capitalization). The database is the name of a database within the server, and the hostname and port determine the location of the server. If connection via a Unix socket on the local machine, the path of the socket can be used instead of a numerical port.

The first version used above creates a connection to the MySQL server on the local machine via a Unix-style socket. This is the most efficient way to communicate with the database and should be used if you are connecting to a local server. If the hostname is supplied it will connect to the server on that host using the standard port unless the port is supplied as well. If you do not provide a username and password when connecting to a MySQL server, the user executing the program must have sufficient privileges within the MySQL database.

Note: Perl 5 has two difference calling conventions for modules. With the object-oriented syntax, the arrow symbol "->" is used to reference a method in a particular class or object (as in DBI->connect). Another method is the indirect syntax, in which the method name is followed by the class name, then the arguments. The las connect method above would be written as connect DBI 'DBI:mysql:mydata', 'me', 'mypass'. Because of conventions used in early versions of the MySQL Perl modules, a lot of older Perl code that interfaces with MySQL will have lines in it like SelectDB $dbh 'test' wher a simple $dbh->selectdb('test') would do. If you haven't guess, we are partial to the object-oriented syntax, if only because the arrow makes the relationship between class/object and method clear.

Once you have connected to the MySQL server, the database handle -- $dbh in all of the examples in this section – is the gateway to the database server. For instance to prepare a SQL query:

```
$dbh->prepare($query);
```

MySQL allows clients to use any number of different databases during a session, and different databases can even be accessed simultaneously during a query. Each connection also has a default database, which can

be changed at any time. This is the database that is used if no specific database name is given. However, sometimes it is necessary to access two databases that reside on entire separate servers at the same time. To enable this, DBI allows a program to open any number of simultaneous database handles and use them side-by-side.

Chapter XX, Perl Reference, describes the full range of method and variables supplied by DBI.

As an example of the use of DBI consider the following simple programs. In example XX-1, datashow.cgi is a CGI program which accepts a hostname as a parameter--"localhost" is assumed if no parameter is present. The program then displays all of the databases available on that host.

Example 10-1. The CGI program database.cgi shows all of the databases on a MySQL server.

```
#!/usr/bin/perl

use warnings;
use strict;
use CGI qw(:standard);
use CGI::Carp;

# Use the DBI module
use DBI;

my ($server, $sock, $host);

$server = param('server') || '';

# Prepare the MySQL DBD driver
my $driver = DBI->install_driver('mysql');

my @databases = $driver->func($server, '_ListDBs');

# If @databases is undefined we assume that means that
# the host does not have a running MySQL server. However, there
# could be other reasons for the failer. You can find a complete
# error message by checking $DBI::errmsg.
if (not @databases) {
         print header, start_html('title'=>"Information on $server",
               'bgcolor' => 'white' );
         print <<END_OF_HTML;
<h1>$server</h1>
$server does not appear to have a running MySQL server.
</body></html>
END_OF_HTML
        exit(0);
}

        print header, start_html('title'=>"Information on $host",
               'bgcolor'=>'white');
        print <<END_OF_HTML;
<h1>$host</h1>
<p>
$host\'s connection is on socket $sock.
<p>
```

```
Databases:<br>
<ul>
END_OF_HTML
foreach (@databases) {
   print "<li>$_\n";
}
print <<END_OF_HTML;
</ul>
</body></html>
END_OF_HTML
exit(0);
```

You probably noticed in this example that we never created a database handle. This is because we never required a connection to a specific database on the server. Instead, we only needed a list of the available databases. To do this, we first loaded the DBD driver for MySQL. This step is normally performed automatically when you make an explicit connection to the server. However, since we are not making a connection, we need to explicitly load the driver in order to use any of its functions.

Once we have loaded the DBD driver, we can make use of any methods it provides. Most methods require a connection to the database server, but a few do not. In our case we want to get a list of databases on a particular server, which is a function that does not require a pre-existing database connection. We call the '_ListDBs' function as parameter to the 'func' method of the driver. This is different than standard DBI methods that are called directly as methods against a database handle.

However, as complete as DBI is, there are some features it does not provide, especially if they are specific to a certain database server. On such feature is the ability to list the available databases on a server. The database servers supported by DBI do not have a common concept of a 'database'. For many of them, being able to list the available databases would not be as useful as it is for MySQL. For this reason, the DBI does not provide a standard method for listing the available databases on a database server.

However, the author of DBI anticipated that DBI would not be able to provide every piece of functionality present in every supported database server. Therefore DBI was given the ability to run database server-specific functions. This should generally be avoided, as code that uses database server specific functionality can not be directly ported to a new database server if need be. But sometime it is necessary to resort to database server specific functionality to get the job done.

Database server specific functions are accessed via the 'func' method that is present in most DBI objects. In our case, we have a driver object created from the DBD MySQL module. Through the 'func' method on this object, we are able to call the MySQL-specific 'ListDBs' function and retrieve a list of the databases on a specific database and server.

Once we have that information, we can create an HTML response page that lists the databases available on a MySQL server.

Now that we know what databases are available to use, the next step is to see what tables we can use. In Example XX-2 tableshow.cgi accepts the name of a database server

(default is “localhost”) and the name of a database on that server. The program then shows all of the available tables on that server.

Example XX-2. The CGI program tableshow.cgi shows all of the tables within a database.

```
#!/usr/bin/perl -w

use strict;
use warnings;
use CGI qw(:standard);
use CGI::Carp;

# Use the DBI module
use DBI;

my $db = param('db') or die "Database not supplied!";
my $host = param('host') || 'localhost';

# Connect to the requested server.
my $dbh = DBI->connect("DBI:mysql:$db:$host", undef, undef);

# If $dbh does not exist, the attempt to connect to the database
# server failed. The server may not be running, or the given
# database may not exist.
if (not $dbh) {
    print header, start_html('title'=>'Information on $host => $db'
            "bgcolor" => 'white');

    print <<END_OF_HTML;
<h1>$host</h1>
<H2>$db</h2>
The connection attemp failed for the followign reason:<br>
$DBI::errstr
</body></html>
END_OF_HTML
        exit(0);
}

print header, start_html('title'=>'Information on $host => $db',
              'bgcolor' => 'write' );
print <<END_OF_HTML;
<h1>$host</h1>
<h2>$db</h2>
<p>
Tables:<br>
<ul>
END_OF_HTML
# $dbh->listtable returns an array of the tables that are available
# in the current database.
my @tables = $dbh->tables;
foreach (@tables) {
     print "<li>$_\n";
}

print <<END_OF_HTML;
</ul>
</body></html>
END_OF_HTML
     exit(0);
```

In this example, we created an actual connection to a MySQL server for the first time. This connection was made to the server and port number given as parameters from the client browser. If no specific hostname and port number are given, the Unix socket /tmp/mysql.sock on the localhost is used by default.

Once we have created an active connection to the desired database, we can interact with that database using the standard DBI methods. In our case, we want to obtain a list of tables that are available within the database. DBI provides the 'tables' method that returns a list of tables within a database.

Notice that at the end of the script we do not explicitly close the database handle or do any other cleanup. The DBI module will automatically close and cleanup any connections at the end of script.

Now that we know the names of all of the databases and tables available to us, we can take the last step and look at the structure and data within each table. Example XX-3 shows all of the information about a specific table, including its data.

Example XX-3. The CGI program tabledump.cgi Shows Information About a Specific Table

```
use struct;
use warnings;
use CGI qw(:standard);
use CGI::Carp;

# Use the DBI module
use DBI;

my ($db, $table, $host);
$host = param('host') || '';
$db = param('db') or die "Database not supplied!";
$table = param('table') or die 'Table not supplied!';

# Connect to the requested server.
my $dbh = DBI->connect("DBI:mysql:$db:$host", undef, undef);

# WE now prepare a query for the server asking for all of the
# data in the table.
my $table_data = $dbh->prepare("select * from $table");
# Now send the query to the server.
$table_data->execute;

# If the return value is undefined, the table must not exist.
# (Or it could be empty; we don't check for that.)
if (not $table_data) {
     print header, start_html('title'=>
       "Information on $host => $db => $table", 'bgcolor'=>'white');

     print <<END_OF_HTML;
<h1>$host</h1>
<h2>$db</h2>
The table '$table' does not exist in $db on $host.
</body></html>
END_OF_HTML
     exit(0);
}
```

```
# At this point, we know we have datat to display. First we show
# the layout of the table.
print header, start_html('title'=>
       "Information on $host => $db => $table", 'bgcolor'=>'white');
print <<END_OF_HTML;
<h1>$host</h1>
<h2>$db</h2>
<h3>$table</h3>
<p>
<table border>
<caption>Fields</caption>
<tr>
   <th>Field</th><th>Type</th><th>Size</th><th>NOT NULL</th>
</tr>
<ul>
END_OF_HTML

# $table_data->NAME returns a reference to an array of the fields
# of the database.
my @fields = @{$table_data->NAME};
# $table_data->TYPE returns an array reference of the types of
# fields. The types returned here are in SQL standard notation,
# not MySQL specific.
my @types = @{$table_data->TYPE};
# $table_data->is_not_null returns a boolean array reference
# indicating which fields have the 'NOT NULL' flag. Notice the
# term 'NULLABLE' has the opposite context as 'NOT NULL'
my @nullable = @{$table_data->NULLABLE};
# $table_data->PRECISION returns an array reference of the lengths
# of the fields. This is defined when the table is created.
# For CHAR-type fields, this is the maximum number of characters.
# For numeric fields this is the maximum number of significant digits.
my @length = @{$table_data->PRECISION};

# All of the above arrays were returned in the same order, so that
# fields[0], $types[0], $not_null[0] and $length[0] all refer to
# the same field.

foreach my $field (0..$#fields) {
   print <<END_OF_HTML;
<tr>
    <td>$fields[$field]</td><td>$types[$field]</td><td>
END_OF_HTML
    print $length[$field] if $types[$field] eq 'SQL_CHAR';
    print '<td>';
    print 'N' if not $nullable[$field];
    print "</tr>\n";
}

print <<END_OF_HTML;
</table>
<p>
<b>Data</b><br>
<ol>
END_OF_HTML

# Now we step through the data, row by row, using
# DBI::fetchrow_array(). We save the data in an array that has
# the same order as the informational arrays @fields, @types, etc.)
# we created earlier.
```

```
While ( my @data = $table_data->fetchrow_array ) {
   print "<li>\n<ul>";
   for (0..$#data) {
         print "<li>$fields[$_] => $data[$_]</li>\n";
   }
   print "</ul></li>";
}

print <<END_OF_HTML;
</ul>
</body></html>
END_OF_HTML
```

This is the most complex of the scripts by far. As in the tables script, we start of by connecting to the database using the parameters passed to us from the client browser. We then use that database connection to execute a SQL query that retrieves all of the data from a table.

The first step in executing a SQL query is to prepare it. DBI provides the 'prepare' method within database handle object. The prepare method takes a SQL query and stores it (either locally or on the database server) until execution. On database servers that store the query on the database server itself, it is possible to perform operations on the query before executing it. However, MySQL does not support that ability yet, and prepared queries are simply stored within the database driver until execution.

The result of the prepare method is an object known as a statement handle. A statement handle is a Perl program's interface to a SQL query, much like a database handle is the interface to the database server itself. While the statement handle is created when the SQL query is prepared, it is not possible to do anything useful with it until the query has been executed.

A query is executed by using the 'execute' method on a statement handle. That is, one a statement handle has been created using 'prepare', calling 'execute' on that handle will cause the query to be sent to the database server and executed. The result of executing a query depends on the type of query. If the query is a non-SELECT query that returns no data (such as INSERT, UPDATE and DELETE) the execute method will return the number of rows that were affected by the query. That is, for an insert query (that inserts one row of data), the execute query will return '1' if the query was successful.

For SELECT queries, the execute method simply returns a true value if the query was successful and a false value if there was an error. The data returned from the query is then available using various methods within the statemet handle itself.

In addition to the data returned from a SELECT query, the statement handle also contains various information about the data (called meta-data). The meta-data associated with the query can be accessed via various properties in the statement handle. In our example we use several of those properties to build a table containing information about the table in question:

$statement_handle->NAME

A reference to an array of the names of the columns in the result set. Since our query is selecting all of the columns from a table, this contains the names of all of the columns in the table.

$statement_handle->TYPE

A reference to an array of the SQL types of the columns in the result set. These types are returned as ANSI SQL standard types. While these are often the same as the MySQL data types, many of the more unusual MySQL data types (such as NUMERIC, and TEXT) are represented as simpler ANSI standard types.

$statement_handle->NULLABLE

A reference to an array of boolean values indicating whether the columns in the result set can contain NULL values. Note that this has the opposite meaning as the 'NOT NULL' which is used when defining MySQL columns. Thus, a NOT NULL column will have a value of false in the NULLABLE array, and vice versa.

$statement_handle->PRECISION

A reference to an array of the maximum lengths of the columns in the result set. These maximum values are defined when the table is created. For character-based columns, this is the maximum number of characters. For numeric columns, this is the number of significant digits.

After printing a table of this meta-data, the program then displays all of the data in the table, row by row. This is done by using the fetchrow_array method on the statement handle containing the data. The fetchrow_array method reads a single row of data from the result set and then advances an internal pointer so that the next call to fetchrow_array will return the next row of data. This continues until there are no rows left, at which time the method will return a false value.

Each row of data is returned as an array, in the order defined in the query. In our case, the query simply specifies 'SELECT *', so we don't know the order in which the fields were defined. However, it is guaranteed that the order of this array is the same as the order of the arrays of meta-data generated earlier. Therefore, we can loop through the data array and use the same indices on the meta-data arrays to describe the columns.

An Example DBI Application

DBI allows for the full range of SQL queries supported by MySQL. As an example, consider a database used by a school to keep track of student records, class schedules, test scores and so on. The database would contain several tables, one for class information, one for student information, one containing a list of tests, and a table for each test. MySQL's ability to access data across tables—such as the table-joining feature—enables all of these tables to be used together as a coherent whole to form a teacher's aide application.

To begin with, we are interested in created tests for the various subjects. To do this we need a table that contains names and ID numbers for the tests. We also need a seperate table for each test. This table will contain the scores for all of the students as wll as a perfect score for comparison. The test table has the following structure:

```
CREATE TABLE test (
```

```
      id INT NOT NULL AUTO_INCREMENT,
      name CHAR(100),
      subject INT,
      num INT
)
```

The individual tests have table structures like this:

```
CREATE TABLE t7 (
     id INT NOT NULL,
     q1 INT,
     q2 INT,
     q3 INT,
     q4 INT,
     total INT
)
```

The table name is t followed by the test ID number from the test table. The user determines the number of questions when he or she creates the table. The total field is the sum of all of the questions.

The program that access and manipulates the test information is the CGI program test.cgi. This program, which follows, allows only for adding new tests. Viewing tests and changing tests is not implemented but is left as an exercise. Using the other scripts in this chapter as a reference, completing this script should be only a moderate challenge. As it stands, this script effective demonstrates the capabilities of DBI.

```
#!/usr/bin/perl

use warnings;
use strict;

use CGI qw(:standard);

# Use the DBI module.
use DBI;
# DBI::connect uses the format 'DBI:driver:database', in our case
# we are using the MySQL driver and accessing the 'teach' database.
my $dbh = DBI->connect('DBI:mysql:teach');
# The add action itself is broken up into three seperate functions.
# The first function, add, prints out the template form for the
# user to create a new test.
sub add {
     $subject = param('subject') || '';
     $subject = '' if $subject eq 'all';

print header, start_html('title'=>'Create a New Test',
     'bgcolor'=>'white');
print <<END_OF_HTML;
<h1>Create a New Test</h1>
<form action="test.cgi" method="post">
<input type="hidden" name="action" value="add2">
Subject:
END_OF_HTML

my @ids = ();
my %subjects = ();
my $out2 =
      $dbh->prepare("SELECT id, name FROM subject ORDER BY name");
$out2->execute;
```

```
# DBI::fetchrow_array retrieves a single row of the results.
while ( my($id, $subject) = $out2->fetchrow_array ) {
     push(@ids, $id);
     $subjects{$id} = $subject;
}

print popup_menu('name'=>'subjects',
      'values'=>[@ids],
      'default'=>$subject,
      'labels'=>\%subjects);
print <<END_OF_HTML;
<br>
Number of Questions: <input name="num" size="5"><br>
An identifier for the test (such as a date):
  <input name="name" size="20">
<p>
<input type="submit" value=" Next Page ">
  <input type="reset">
</form></body></html>
END_OF_HTML
}
```

This function displays a form allowing the user to choose a subject for the test along with the number of questions and a name. In order to print out a list of available subjects, the table of subjects is queried. When using a SELECT query with DBI, the query must first be prepared and then executed. The DBI::prepare function is useful with certain database servers which allow you to perform operations on prepared queries before executing them. With MySQL however, it simply stores the query until the DBI::execute function is called.

The output of this function is sent to the add2 function as shown in the following:

```
sub add2 {
     my $subject = param('subjects');
     my $num = param('num');
     my $name = param('name') if param('name');

     my $out = $dbh->prepare("select name from subject where id=$subject");
     my ($subname) = $out->fetchrow_array;

     print header, start_html('title'=>"Creating test for $subname",
           'bgcolor'=>'white');
     print <<END_OF_HTML;
<h1>Creating test for $subname</h1>
<h2>$name</h2>
<p>
<form action="test.cgi" method="post">
<input type="hidden" name="action" value="add3">
<input type="hidden" name="subjects" value="$subject">
<input type="hidden" name="num" value="$num">
<input type="hidden" name="name" value="$name">
Enter the point value for each of the questions. The points need not add up to 100.
<p>
END_OF_HTML
     for (1..$num) {
          print qq%$_: <input name="q$_" size="3"> %;
          if (not $_ % 5) { print "<br>\n"; }
     }
     print <<END_OF_HTML;
<p>
```

```
Enter the test of the test:<br>
<textarea name="test" rows="20" cols="60">
</textarea>
<p>
<input type="submit" value="Enter Test">
<input type="reset"
</form></body></html>
END_OF_HTML
}
```

In this function, a form for the test is dynamically generated based on the parameters entered in the last form. The user can enter a point value for each question on the test and the full text of the test as well. The output of the function is then sent to the final function, add3, as shown in the following:

```
sub add3 {
    my $subject = param('subjects');
    my $num = param('num');

    $name = param('name') if param('name');

    my $qname;
    ($qname = $name) =~ s/'/\\'/g;
    my $q1 = "insert into test (id, name, subject, num) values (
        '', '$qname', $subject, $num)";

     my $in = $dbh->prepare($q1);
     $in->execute;

     # Retrieve the ID value MySQL created for us
     my $id = $in->insertid;

     my $query = "create table t$id (
           id INT NOT NULL,
           ";

     my $def = "insert into t$id values ( 0, ";

     my $total = 0;
     my @qs = grep(/^q\d+$/, param);
     foreach (@qs) {
        $query .= $_ . " INT,\n";
        my $value = 0;
        $value = param($_) if param($_);
        $def .= "$value, ";
        $total += $value;
    }
    $query .= "total INT\n)";
    $def .= "$total)";

    my $in2 = $dbh->prepare($query);
    $in2->execute;
    my $in3 = $dbh->preapre($def);
    $in3->execute;

    # Note that we store the tests in seperate files. Another
    # method of handling this would be to stick the entire test
    # into a TEXT column in the table.
    open (TEST, ">teach/tests/$id") or die "A: $id $!";
    print TEST param('test'), "\n";
    close TEST;
```

```
    print header, start_html('title'=>'Test Created',
           'bgcolor'=>'white');

    print <<END_OF_HTML;
<h1>Test Created</h1>
<p>
The tst has been created.
<p>
<a href=".">Go</a> to the Teacher's Aide home page.<br>
<a href="test.cgi">Go</a> to the Test main page.<br>
<a href="test.cgi?action=add">Add</a> another test.
</body></html>
END_OF_HTML
}
```

Here we enter the information about the test into the database. In doing so we take a step beyond the usual data insertion that we have seen so far. The information about the test is so complex that each test is best kept in a table of its own. Therefore, instead of adding data to an existing table, we have to create a while new table for each test. For we crate an ID for the new test using MySQL auto increment feature and enter the name and ID of the test info a table called test. This table is simple an index of tests so that the ID number of any test can be quickly obtained. Then we simultaneously create two new queries. The first is a CREATE TABLE query that defines our new test. The second is an INSERT query that populates our new table with the maximum score for each question. These queries are then sent to the database server, completing the process (after sending a success page to the user_. Later, after the students have taken the test, each student will get an entry in the test table. Then entries can be compared to the maximum values to determine the student's score.

Object Oriented (OO) Database Programming in Perl

Perl is rarely on anyone's list of theoretically complete object-oriented languages. However, this is mostly because of mis-education and Perl does in fact have very thorough and flexible object-oriented features. However, as with all things Perl, There Is More Than One Way To Do It. That is, while you can write object-oriented Perl, you can also write non-object-oriented Perl or a mixture of OO and non-OO. This flexibility leads to possibilities not available in most other program languages. On the other hand, it also introduces the necessity of discipline on programmers who want to use a strict Object Oriented structure.

One of the best ways to ensure discipline when creating a Object Oriented system, is to use a good design methodology. A design methodology is simply a framework that helps you visualize a system in an Object Oriented manner. There are several good methodologies in existence, but for simplicities sake we'll concentrate on one: Model/View/Controller.

Model/View/Controller

Model/View/Controller (MVC) is an Object Oriented methodology used to help design a software application. The base idea behind MVC is that any application can be split into three distinct parts, or layers: The Model, the View and the Controller. Each layer is an independent unit that performs a specific function.

View

> The View is the user-interface aspect of the application. The View is responsible for presenting information to the user, and also for collecting any user feedback. In a traditional desktop application, the View is the code that draws the screens and reads the keyboard and mouse inputs. In a Web-based application, the View is the code that generates the HTML viewed by the user's browser, as well as the code that interprets any form data submitted by the user. All I/O that involves the user of the system is done in the View. Any input by the user is passed to the Controller for processing.

Controller

> The Controller is the brains of the application. Any software logic performed by the application is done within the controller. In addition, the controller is also the communication center of the application. All user input from the View is processed here, as is all data from the Model (destined for the View). The controller should not be dependant on the View in any way. That is, it should be possible to replace the View (perhaps changing from the desktop application to a Web-based application) without altering the Controller.

Model

> The Model is the body of the application. Here, all objects that represent real-world 'things' within the application are modeled. For example, in the Teacher's Aide example used previously, concepts such as 'tests', 'classes', and 'students' may be represented as objects in the Model. The model is also responsible for any persistence of these objects. Therefore, all database-interaction is performed in that layer. The Model should not be dependant on either the View or Controller in any way. This allows entirely new applications (Controllers and Views) to be built around the same concepts and databases (Models).

As mentioned above, the Model is the only layer that interfaces with external data sources, such as databases. This provides a very convenient abstraction with designing a system. Consider the Teacher's Aide example used earlier. Each script used in that example had to deal with creating HTML, processing form input, accessing the database, and performing logic to manipulate the data. In a large project, these different tasks are logic places to divide labor and it is useful to have a way to separate them.

Using the MVC framework, creating the HTML and processing form input would be the responsibility of the View. Accessing the database is the responsibility of the Model and performing logic is the responsibility of the Controller.

Since our purpose here is to examine using Perl to interface with MySQL, it is clear that the Model is the only layer that direct affects us. This frees us from having to deal with code that is extraneous to our central purpose.

Designing the Model

Designing the Model layer is one of the most important tasks of creating an MVC application. The Model contains abstractions of all concrete 'things' that are used within the application. Therefore it is necessary to have a solid model as a foundation for the rest of the application.

Luckily for us, designing a Model for a database-driven application is very straightforward. That is because the work discovering the relavent abstractions in a system was already done when the database scheme was created. In most cases, each table in the database corresponds to one Model class. The fields of the tables correspond to the attributes of the class. Relationships between tables can usually be expressed in the following manner:

One-to-One

> If two tables have a one-to-one relationship, one of two things can happen. If the relationship is one of containment, the contained object should exist as an attribute of the container class. That is, a 'Person' table can be one-to-one with an 'Address' table if a Person has exactly one address. Therefore, the Person class should contain an Address object as an attribute. If the relationship is one of aggregation, the more specific class should be a subclass of the less specific class. That is, an 'Animal' table can be one-to-one with a 'Dog' table, if the Dog has all of the fields of an Animal, plus fields of it's own. Therefore, the Dog class should be a subclass of the Animal class.

One-to-Many

> If two tables have a one-to-many relationship, usually the 'One' class contains an array of 'Many' objects. That is, a 'Person' table can be one-to-many with a 'Phone' table as a Person usually has multiple phone numbers. Therefore, the Person class should contain an array of Phone objects.

Many-to-Many

> If two tables have a many-to-many relationship, each class can contain an array of objects from the other class. That is, a 'Person' table can be many-to-many with an 'Employer' table (with a many-to-many join table in the middle) since a Person usually has had more than one Employer and each Employer has more than one Person. Therefore, the Person class contains an array of Employer objects, and the Employer class contains an array of Person objects. This type of construct can be very challenging to implement (when you create a Person, you create all of their Employers, each of those Employers will then contain the Person, which contains the Employers, etc., etc.). Because of this, many designers avoid many-to-many relationships when possible. If they are necessary, however, it is possible to pull it off with careful implementation.

Like all classes, a Model class is comprised of attributes and methods. As mentioned above, the attributes of a Model class are simply the fields of the underlying table (as well as possibly objects from related tables). What about the methods?

In object-oriented programming, classes have two kinds of methods: instance and static. Instance methods are only called upon actual objects created from the class. Because of this, they have access to the attribute data within the object. Static methods, on the other hand, can be called on the class itself. They have no knowledge of individual objects of that class.

If you do not mind directly accessing attributes of an object, a Model class only needs three instance methods: update, delete and create. These methods parallel the SQL 'UPDATE', 'DELETE' and 'INSERT' statements that, along with 'SELECT', make up the vast majority of SQL statements.

The update method saves the current state of the object to the database. That is, when an attribute of a Model object is altered somewhere in the application, that change only happens within that object. If the application were to terminate, the object will leave memory and the change would disappear. To make a change permanent, it is necessary to save the attribute of the object to the database. The update method does that by constructing a UPDATE SQL query with all of the attributes of the object and sending it to the database. Like all methods this can be called anything, including 'update', which is simple and to the point.

The delete method removes the object data from the database. The objects running within the program are not directly tied to the database, so that if an object is destroyed (as when it is garbage collected, or when the program terminates), no change is made within the database. To delete an object's data from the database, it is necessary to send the database a 'DELETE' SQL statement to remove the row containing the data. This is the purpose of the delete method. Unfortunately, 'delete' is a keyword in Perl, so it is not possible to use that is the method name. Common alternatives include 'remove', 'destroy', 'Delete' and 'deleteObject'.

As mentioned above, the create method mirrors the SQL INSERT statement which creates a new row of data in the database. Because Model objects exist as data constructs within a running application with no direct tie to the database, it is possible to create a new object that has no corresponding data in the database. Thus, when you want to persist that data, you must create a new row in the database table for it. The create method does this by generating a SQL INSERT statement that contains the data within the object and sending it to the database. The name 'create' is used for the method here, because 'creating' an object makes more logical sense to the rest of the application than 'inserting'. Because the Model hides the details of persistence from the rest of the application, the rest of the application has no idea that there is a database behind the scenes where it is necessary to insert rows. However, this is merely semantics and 'insert', or anything else, would be just as good a name for this method.

While update and delete are the only necessary instance methods in a Model class, a common OO practice is to not directly access attributes, but rather access them through methods (called accessor methods or getter/setter methods). The advantage of this is that it allows the designer to change the attribute in some way in the future, while not changing it's appearance to the rest of the application.

If you follow this practice, then each attribute of the object should have two instance methods: a 'get' method that retrieves the value of the attribute and a 'set' method that sets the attribute to a new value. They can be named anything, but a common practice is to simply prepend 'get/set' to the name of the attribute. So an attribute called 'firstName' would have the methods 'getFirstName' and 'setFirstName'.

The instance methods described above cover three of the four basic SQL commands: 'INSERT', 'UPDATE', and 'DELETE'. This leaves 'SELECT' still untouched. For this, we turn to static methods (also known as class methods). Unlike all of the previous methods, the 'SELECT' method does not operate on already existing objects. The point of a SELECT query is to retrieve data from the database. In other words, you are creating new objects containing data from the database. Therefore, it is necessary to use static methods that do not rely on any instance data.

These methods send SELECT queries to the databases and create new Model objects from the data that is returned. Also differing from the other methods considered so far, there are often several select methods within a Model class. This is because there are usually different contexts in which to create new objects. In particular, there are two situations that are called for in almost every application: Primary Key select and Generic WHERE select

Generic WHERE select

The 'Generic WHERE' select method is the most versitile and common type of select method. In this method, a SQL WHERE query is passed in a parameter (or generated from some other parameters). A SQL SELET is then sent to the database using this WHERE. Out of the resulting data, an array of Model objects is created. Because of the flexbility of the SQL WHERE clause, this method can be leveraged by more specialized select methods, such as the Primary Key select.

Primary Key select

Almost uniformly, well designed relational tables have a primary key. This is a column, or columns, that define a unique row within the table. If you know a primary key value, you can retrieve a single row of data from the table. Having this ability within the Model class allows you to create a single object corresponding to a row of data. This is done by creating a SQL WHERE clause containing the value of the primary key and then calling the generic WHERE select to execute the query. Since we are sending in the value of the primary key, we know we will get an array containing a single object in return. This method then returns this single object.

Example

As an example of a Model class consider a table containing information about a book publisher. For simplicity we'll just use two fields in the table: 'id' and 'name'. The 'id' field is the primary key and uniquely identifies each row of the table. The 'name' field is the name of the publisher.

```
# A Model Class for the publisher table.
```

```
package CBDB::publisher;

our $VERSION = '0.1';

use strict;
use warnings;
use DBI qw(:sql_types);
use CBDB::DB;
use CBDB::Cache;

our @ISA = qw( CBDB::DB );
############################
###### CONSTRUCTOR ########
############################
sub new {
    my $proto = shift;

    my $class = ref($proto) || $proto;
    my $self = { };
    bless($self, $class);

    return $self;
}

##############################
########### METHODS ##########
##############################

# getId() - Return Id for this publisher
sub getId {
    my $self = shift;
    return $self->{Id};
}

# setId() - Set Id for this publisher
sub setId {
    my $self = shift;
    my $pId = shift or die "publisher.setId( Id ) requires a value.";
    $self->{Id} = $pId;
}

# getName() - Return Name for this publisher
sub getName {
    my $self = shift;
    return $self->{Name};
}

# setName() - Set Name for this publisher
sub setName {
    my $self = shift;
    my $pName = shift || undef;
    $self->{Name} = $pName;
}

# remove() - Removes an object from the database
sub remove {
    my $self = undef;
    my $where = undef;
    my $is_static = undef;
    if ( ref($_[0]) and $_[0]->isa("CBDB::publisher") ) {
            $self = shift;
```

```
            $where = "WHERE id = ?";
    } elsif (ref($_[0]) eq 'HASH') {
            $is_static = 1;
            $where = 'WHERE ' . make_where($_[0]);
    } else {
            die "CBDB::publisher::remove: Unknown parameters: " . join(' ', @_);
    }

    my $dbh = CBDB::DB::getDB();
    my $query = "DELETE FROM publisher $where";

    my $sth = $dbh->prepare($query);

    if ($is_static) {
            bind_where($sth, $_[0]);
    } else {
            $sth->bind_param(1, $self->getId(), {TYPE=>4});
    }
    $sth->execute;
    $sth->finish;
    $dbh->disconnect;
}

# update() - Updates this object in the database
sub update {
    my $self = shift;
    my $dbh = CBDB::DB::getDB();
    my $query = "UPDATE publisher SET name = ?, id = ? WHERE id = ?";
    my $sth = $dbh->prepare($query);

    $sth->bind_param(1, $self->getName(), {TYPE=>1});
    $sth->bind_param(2, $self->getId(), {TYPE=>4});
    $sth->bind_param(3, $self->getId(), {TYPE=>4});
    $sth->execute;
    $sth->finish;
    $dbh->disconnect;
    CBDB::Cache::set('publisher', $self->getId(), $self);

}

# getByPrimaryKey - Retrieves a single object
#                   from the database based on a primary key
sub getByPrimaryKey {
    my $pId = shift or die "publisher.get()";

    my $where = [ {'id' => $pId } ];
    return ( get( $where, 1 ) )[0];
}

# get - Retrieves objects from the database
sub get {
    my $wheres = undef;
    my $do_all = 1;
    if (ref($_[0]) eq 'ARRAY') { $wheres = shift; $do_all = shift if @_; }
    else { $do_all = shift; }

    my $dbh = CBDB::DB::getDB();
    #my $where = "WHERE ";
    my $where .= ' WHERE  ' . make_where( $wheres );
    my $query = "SELECT publisher.name as publisher_name, publisher.id as
publisher_id FROM publisher $where";
```

```
    my $sth = $dbh->prepare($query);
    bind_where( $sth, $wheres );
    $sth->execute;
    my @publishers;
    while (my $Ref = $sth->fetchrow_hashref) {
            my $publisher = undef;
            if (CBDB::Cache::has('publisher', $Ref->{publisher_id})) {
                    $publisher = CBDB::Cache::get('publisher', $Ref-
>{publisher_id});
            } else {
            $publisher = CBDB::publisher::populate_publisher( $Ref );

                    CBDB::Cache::set('publisher', $Ref->{publisher_id}, $publisher);
            }
            push(@publishers, $publisher);
    }
    $sth->finish;
    $dbh->disconnect;
    return @publishers;
}

# populate_publisher - Return a publisher object populated from a result set
sub populate_publisher {
    my $Ref = shift;
    my $publisher = new CBDB::publisher;
    $publisher->setName($Ref->{publisher_name});
    $publisher->setId($Ref->{publisher_id});

    return $publisher;
}

# create - Inserts the object into the database
sub create {
    my $self = shift;
    my $dbh = CBDB::DB::getDB();
    my $query = "INSERT INTO publisher ( name, id ) VALUES ( ?, ? )";
    my $sth = $dbh->prepare($query);
    my $pk_id = undef;

    $sth->bind_param(1, $self->getName(), {TYPE=>1});
    $sth->bind_param(2, undef, {TYPE=>4});
    $sth->execute;
    $sth->finish;

    $pk_id = CBDB::DB::get_pk_value($dbh, 'publisher_id');
    $self->setId( $pk_id);

    $dbh->disconnect;
    CBDB::Cache::set('publisher', $self->getId(), $self);
    return $self;
}

# make_where() - Construct a WHERE clause from a well-defined hash ref
sub make_where {
    my $where_ref = shift;
    if ( ref($where_ref) ne 'ARRAY' )
            { die "CBDB::publisher::make_where: Unknown parameters: " .
                join(' ', @_); }
    my @wheres = @$where_ref;
    my $element_counter = 0;
    my $where = "";
```

```
    foreach my $element_ref (@wheres) {
            if (ref($element_ref) eq 'ARRAY')
               { $where .= make_where($element_ref); }
            elsif (ref($element_ref) ne 'HASH')
               { die "CBDB::publisher::make_where: malformed WHERE parameter: "
                    . $element_ref; }
            my %element = %$element_ref;
            my $type = 'AND';
            if (not $element_counter and scalar keys %element == 1 and
                exists($element{'TYPE'})) {
                    $type = $element{'TYPE'};
            } else {
                    my $table = "publisher";
                    my $operator = "=";
                    if (exists($element{'table'})) { $table = $element{'table'}; }
                    if (exists($element{'operator'}))
                           { $operator = $element{'operator'}; }
                    if ($element_counter) { $where .= " $type "; } else
                            { $element_counter = 1; }
                    foreach my $term
                           ( grep !/^(table|operator)$/, keys %element ) {
                           $where .= "$table.$term $operator ?";
                    }
            }
    }
    return $where;
}

sub bind_where {
    my $sth = shift;
    my $where_ref = shift;
    my $counter_ref = shift || undef;
    my $counter = ref($counter_ref) eq 'Scalar' ?
            $$counter_ref : 1;
    if ( not $sth->isa('DBI::st') or ref($where_ref) ne 'ARRAY' )
            { die "CBDB::publisher::make_where: Unknown parameters: "
                    . join(' ', @_); }
    my @wheres = @$where_ref;
    foreach my $element_ref (@wheres) {
            if (ref($element_ref) eq 'ARRAY')
              { bind_where($sth, $element_ref, \$counter); }
            elsif (ref($element_ref) ne 'HASH')
              { die "CBDB::publisher::make_where: malformed WHERE parameter: "
                   . $_; }
            my %element = %$element_ref;
            unless (not $counter and scalar keys %element == 1 and
                  exists($element{'TYPE'})) {
                    my $table = "publisher";
                    if (exists($element{'table'})) { $table = $element{'table'}; }
                    foreach my $term
                        ( grep !/^(table|operator)$/, keys %element ) {
                            $sth->bind_param($counter, $element{$term},
                                    {TYPE=>CBDB::DB::getType($table,$term)});
                            $counter++;
                    }
            }
    }
}

1;
```

There are 13 methods in this class:

- new

  This is a generic constructor that simply creates an empty object.

- getId

  This is an accessor method that retrieves the current value of the 'id' attribute.

- setId

  This is an accessor method that sets the value of the 'id' attribute. Since the 'id' field of the table is the primary key, this method will rarely be called.

- getName

  This is an accessor method that retrieves the current value of the 'name' attribute.

- setName

  This is an accessor method that sets the value of the 'name' attribute.

- remove

  This method removes the row of data corresponding to this object in the database. After this method is called, this object should be destroyed since it's underlying data is gone.

- update

  This method updates the data in the database with the attribute data in the object. In effect, this method 'saves' the current state of the object into the database. For this method to work, the object must already have a row in the database, since it uses the SQL UPDATE statement.

- getByPrimaryKey

  This is a static method that creates a single object based on a primary key. This method calls the generic 'get' method to perform the actual query.

- get

  This is a static method that creates a SQL SELECT statement based on WHERE parameters passed to the function. For each row of the result set, a new object is created and an array of these objects is returned.

- populate_publisher

  This is a static method that creates a new object based on data from a result set. This is a utility method that is used by 'get'. The advantage of making this a seperate method is that it can be used externally by other Model classes that need to create new 'publisher' objects.

- create

This method inserts a new row into the table with the data from this object. In this case, the primary key for this table is a MySQL AUTO_INCREMENT field with automatically creates a new value. Therefore, this method only inserts the value of the 'name' field. After the new row has been created, it retrieves the value of the newly created primary key and sets the 'id' attribute accordingly.

- make_where

  SQL WHERE clauses can be very complex, especially when multiple tables are involved. The same information stored within a SQL WHERE clause can also be stored in a Perl hash tree. Using this construct, the relationships between the different WHERE elements can be made clear. This method flattens the hash tree into a regular SQL WHERE clause that can be used in a SQL query.

- bind_where

  As mentioned above, the parameters for a WHERE clause in this class are stored in a hash tree that clearly indicates the relationships between the parameters. While the 'make_where' method creates the actual SQL WHERE clause, the values of the parameters still need to be bound to the statement once the statement is prepared. This method traverses the hash tree and calls the bind_param method to insert the parameter values into the query. After this method is called, the SELECT statement can be executed.

You may have noticed in this class that there are two other classes referenced: DB and Cache. These classes help the Model class in different ways.

DB

This class is the super class of all model classes. It handles creating the connection with the database and other utility matters.

```
# this is a static database helper class
# that allows us to make connections to the database
package CBDB::DB;

use strict;
use warnings;
use BM::mysql;

our $VERSION = '0.1';
our $URL = "DBI:mysql:database=CBDB;host=localhost";
our $USER = "myuser";
our $PASSWORD = "mypass";


my %types = (
    'creator' => { 'name' => 1, 'id' => 4 },
    'book' => { 'title' => 1, 'publisher_id' => 4, 'date' => 11, 'id' => 4 },
    'book_creator' => { 'book_id' => 4, 'creator_id' => 4, 'role_id' => 4 },
    'publisher' => { 'name' => 1, 'id' => 4 },
    'role' => { 'name' => 1, 'id' => 4 },
);

sub getDB {
```

```
    my $dbh = DBI->connect($URL,$USER,$PASSWORD);
    return $dbh;
}

sub get_pk_value {
    my $dbh = shift or die "DB::get_pk_value needs a Database Handle...";

    my $dbd = new BM::mysql();
    return $dbd->get_pk_value( $dbh );
}

sub getType {
    my $table = shift;
    my $col = shift;
    return $types{$table}{$col};
}

1;
```

The class contains three methods:

- getDB

  This method creates a connection to the database and returns the database handle. This method is used by the subclass to get a database handle to execute a query.

- get_pk_value

  This method returns the most recent value of the automatically created primary key of the subclass table. Many tables have a primary key that is automatically generated (this is also known as a surrogate key). This method returns the value of that key when a new row is created. In order to provide complete database abstraction this class uses yet another class for the database specific operation (see BM::mysql, below).

- get_type

  This method returns the SQL type of a column within a table used by the application. All of the fields in all of the tables are stored within this superclass so that any subclass can access the type of any of the fields from any time.

mysql

As mentioned above, an attempt has been made to abstract out any database server specific functionality so that the model base classes can use different database servers. The database-specific functionality for MySQL is placed in a utility class called simply 'mysql'.

```
package BM::mysql;

use strict;
use warnings;

sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self = { };
```

```
    bless($self, $class);
    return $self;
}

sub is_pk ($$$) {
    my $self = shift;
    my $sth = shift;
    my $i = shift;

    return 1 if $$sth->{mysql_is_pri_key}->[$i];
    return 0;
}

sub is_auto_increment($$$) {
    my $self = shift;
    my $sth = shift;
    my $i = shift;

    return 1 if $$sth->{mysql_is_auto_increment}->[$i];
    return 0;
}

sub get_pk_value {
    my $self = shift;
    my $dbh = shift or die "mysql::get_pk needs a Database Handle...";
    my $mysqlPk = "Select last_insert_id() as pk";
     my $mysqlSth = $dbh->prepare($mysqlPk);
     $mysqlSth->execute();
     my $mysqlHR = $mysqlSth->fetchrow_hashref;
     my $pk = $mysqlHR->{"pk"};
     $mysqlSth->finish;
     return $pk;
}
1;
```

This class contains four methods:

- new

  This is the generic object constructor

- is_pk

  This method determines whether a field is part of the primary key of the table.

- is_auto_increment

  This method determines whether the primary key of the table is an AUTO_INCREMENT-style field.

- get_pk_value

  Ths method returns the value of the most recently inserted AUTO_INCREMENT field.

Cache

The final class used by our Model class is 'Cache'. This class contains a static hash of all of the Model objects used by the application. Besides increasing performance, it also

allows multiple objects to exist that represent the same row of data in the underlying table. By using references to the cached objects, the objects will automatically reflect changes.

```
package CBDB::Cache;
# This file keeps a copy of all active objects, with records of their primary key.
use strict;
use warnings;

my %cache = ();

sub set { $cache{$_[0]}{$_[1]} = $_[2]; }
sub get { return $cache{$_[0]}{$_[1]}; }
sub has { return exists $cache{$_[0]}{$_[1]}; }
1;
```

As you can see, this class is very simple. It contains a static hash of all the model objects used by the application. Three simple methods are present to add an object to the cache, get an object from the cache and check to see if an object already exists in the cache.

Example

Now that we have our Model class and all of it's supporting classes, let's look at how it is used in practice. The following code would be part of the Controller layer of the application. The is the layer that performs all of the logic. Note how all of the actual database calls are hidden from this layer. All that is used is calls to the Model class.

```
use CBDB::publisher;

# First let's create a new publisher object
my $pub = new CBDB::publisher();

# Now let's set some data... We're in a hurry, so we can't be
# bothered with good spelling...
$pub->setName("Joe's Boks");

# Note that we don't set the 'id'. This is an auto-increment field that is
# taken care of automatically by the database. A more abstract way of thinking
# about it is that the 'id' is not a real-world property of this object, it only
# exists because of the necessities of object-relational design. Therefore, at
# the controller level, we shouldn't have to worry about it.

print 'Our new publisher is ' . $pub->getName();

# If the program were to terminate at this point, this object's data would
# be lost. We need to save it to that database for it to be persistent.
$pub->create();
# Now the object has been created in the database and is persistent…
my $new_id = $pub->getId();
# Now that the object has been persisted, it has been assigned a primary key
# We can store that primary key in a variable for later reference.

# ... (some time later)

# Let's retrieve the object we created earlier...
# We use the getByPrimaryKey value with the PK we stored earlier.
# Notice that this is a static method; called on the class itself.
my $pub2 = CBDB::publisher::getByPrimaryKey($new_id);

# Because of the caching mechanism, $pub2 is literally the same object as
```

```
# $pub.
If ($pub2 != $pub) { print 'Whoops! Something isn't working right!' }

# Let's change some data and fix that typo from earlier.
$pub2->setName("Joe's Books");

# At this point, the data has been changed in the object only, not in the
# underlying data store. However, since all active instances of this object
# are references to the same object, this change takes place everywhere in
# the application immediately.
print 'The publisher's name is now ' . $pub->getName();
# This will print Joe's Books, not Joe's Boks, even though we didn't explicitly
# touch $pub.

# Now let's save these changes to the database...
$pub2->update(); # <-- This could just as well have been $pub

# The data has been changed now...

# ... (even later)

# Okay we're done with this data now, time to delete the row...
$pub->remove();

# The underlying data row in the database has now been removed. However, this
# object (as well as $pub2) still contains the data until the program is
# terminated or the objects are destroyed.
print "The publisher " . $pub->getName() . " was just erased...\n";
```