LAMP, MySQL/PHP Database Driven Websites - Part II

Navigate: PHP Tutorials > PHP

Author: Gizmola
Date: 04/26/2005
Version 1.0
Experience Level: Beginner

This is Part 2 of a 3 Part Series. Part 1 of the Series is here.

A Re-Introduction

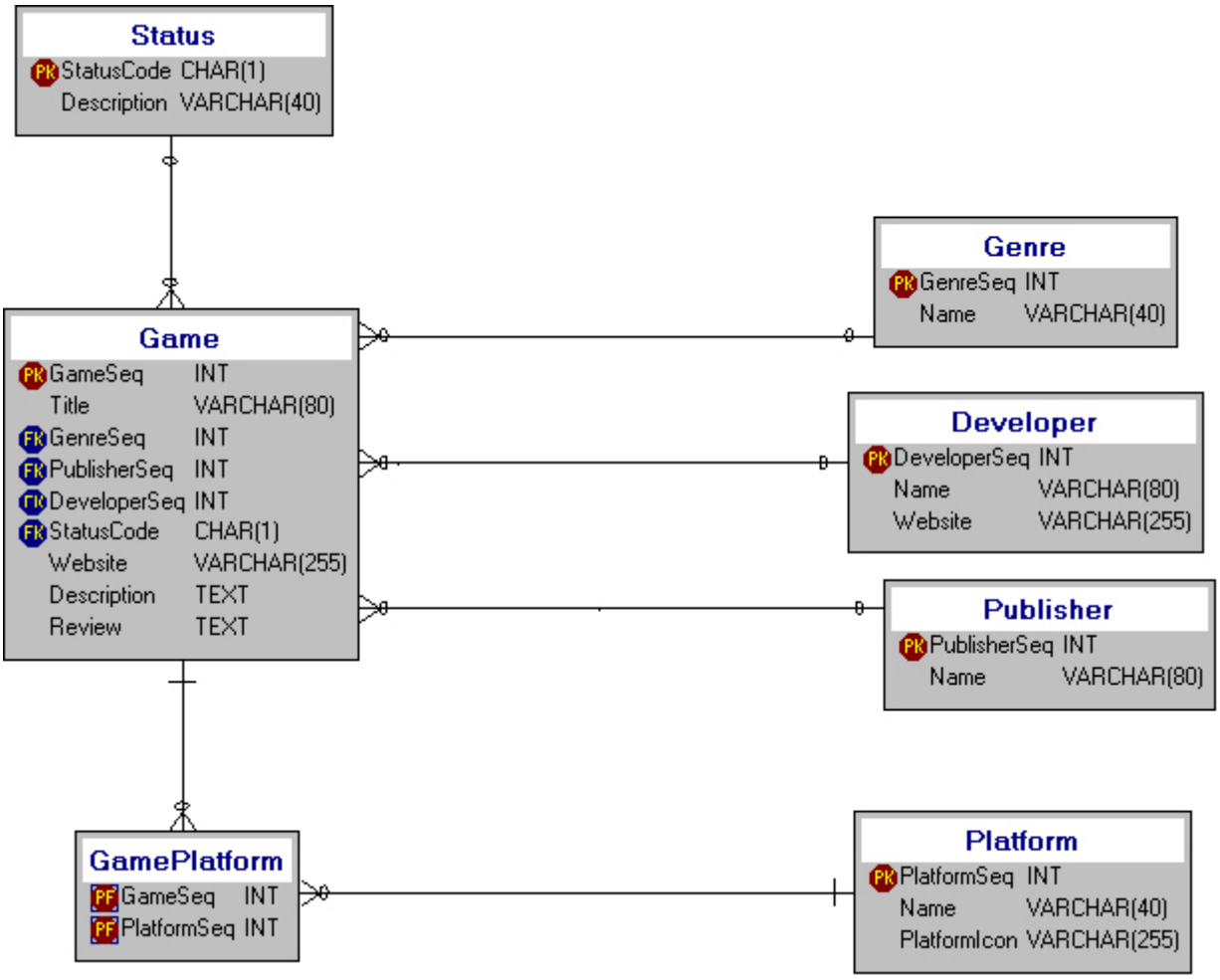
At the end of Part I, the gamestatus team had begun to develop an admin system for their interactive website. To review a little bit about that system, it was constructed using a paradigm I called 'list-detail-post'. The general idea of this paradigm, is that each logical set of information begins in a list view. In Part I, I showed how to give these lists some nice sortation capabilities, allowing you to click on the column headers to re-sort a list in various ways.

Clicking on any individual row drills down into a detail view for that list item. Usually this is a row in the database, however, in any relational database design, there are often related tables which contain additional information about that row. The advantage of implementing a list-detail paradigm is that there is always a simple and relatively intuitive way to present detail information to the user via drill down. For example, a detail page that had further items related to it, in a 1-to-Many fashion can always have a list attached to its form, at the bottom of the page.

The advantage for the user is that the pages all work in basically the same fashion, and it's easy for them to understand how to navigate using the idea of drilling down by clicking. In Part II, we'll address this exact concept in extending the Developer detail script.

Considering the Initial Design

You will probably want to revisit the database design diagram I presented in part 1, to refresh your memory of the gamestatus system design. That diagram looked like this:



Clearly our Developer admin script from part I did not go far enough. Why? If you consider the problem, and think about Game developers it is clear that many game Developers develop multiple games over the lifespan of their company. Take the game company Blizzard Entertainment for example: Blizzard's well known games include Starcraft, Diablo and now World of Warcraft.

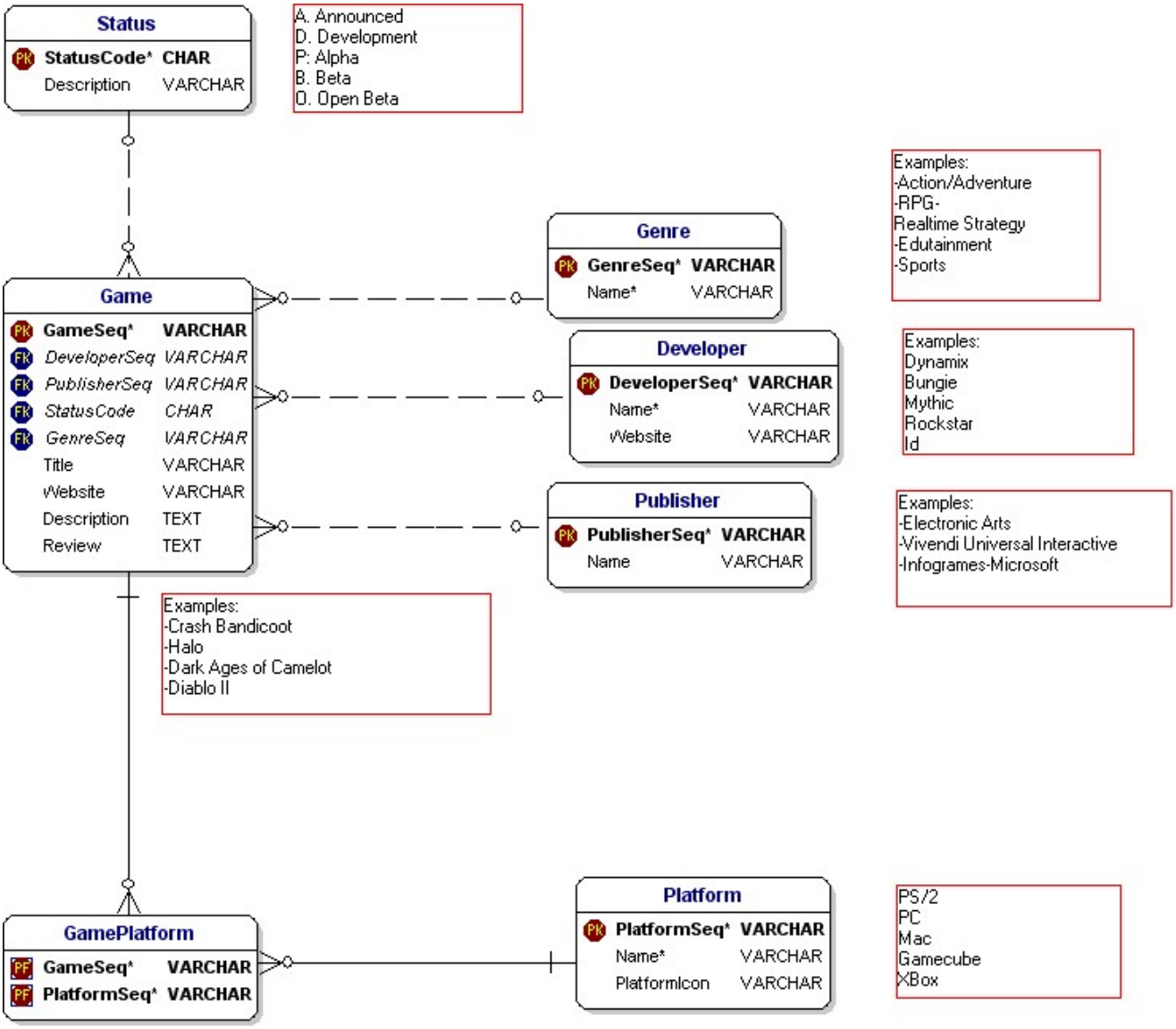
The relationship between Developer and Game is One to Many (1-M), but is it also logically more than that? The question to be asked is this: *Could a game exist without a developer?* For the purposes of the gamestatus system, the answer to that question is -- no. Games don't just appear out of thin air, they have to be developed by someone. It's also good to look at any relationship from the opposite direction. Can a developer exist without a game? Without getting existential about it, Developers can and do exist without having any games. Many game companies start up without even knowing what game they first plan to develop.

These are the sorts of questions you should be asking during the database design process, and before you have started coding PHP, because the answers will guide the design of your system. Your database design should come first.

More Relational Database design

In the case of the relationship between Developer and Game, that relationship is not only 1-M but could easily be considered to be a **defining** relationship. To discuss this further we need to digress into database mechanics for a moment.

In the first tutorial the diagrams I presented weren't quite right. Part of the reason for this, is that the tool I used to make those diagrams, DeZign for Databases didn't provide a visual indication of the difference between a defining relationship and a non-defining one. Since then and now, DeZign has been upgraded several times (it's now at version 3.x). It now follows the convention that a non defining relationship is illustrated by a dashed line, while a defining relationship is shown with an unbroken line. Here is the same data model used in Part I, rendered using the 3.x version of DeZign.



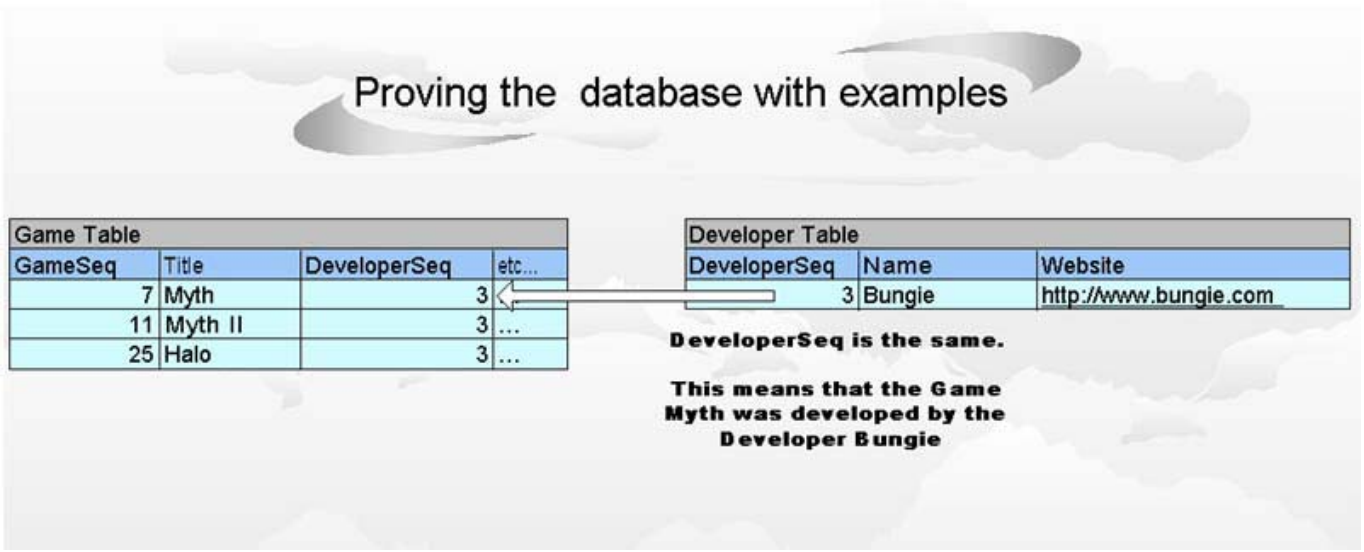
Foreign Keys

What happens when you connect two tables in a 1-M relationship? If you look at Developer and Game, it is easy to see that the table on the *many* side of the 1-M **receives the key of the table on the 1 side**. In other words, DeveloperSeq becomes *part of the Game table*. This is what people mean when they use the term a **'Foreign key'**. DeveloperSeq in the Game table is a column that is actually the *primary key of the Developer table*. It is a key that is **foreign** to Game, hence the name 'Foreign key'. A foreign key is literally 'someone else's primary key'. What you need to understand about this type of relationship between two tables, is that for the table on the "one" side of the 1-M relationship, any single row can only *provide a link to ONE row in the "Many" table*. This should make sense when you look at the Game table, and note that it has a

column called DeveloperSeq. Obviously for any one Game row, you can only store one value in the DeveloperSeq column, and that is going to be the Primary Key for the Developer who developed the Game.

Proving your Design

One way to "prove" or test out your model with example data, is to use a whiteboard, or a spreadsheet, or even a piece of paper to create conceptual tables, and plug in example rows. Doing this can help you visualize what will happen when the physical data is stored inside your database, without having to actually create it and use sql insert statements and selects.



Using this technique, the following diagram illustrates a sample game company (Bungie) who has published a number of different games. We assume that Bungie's "primary key" will be 3. Now we place some sample rows in the Game table for Bungie. This diagram should make the following very clear:

- The way you can determine if a Game was developed by Bungie, is by looking at rows in the Game table which have a DeveloperSeq of 3.
- Any single game (or if you prefer, any single ROW) can ONLY have one developer, because you can only store one value in the DeveloperSeq column in the Game table.

If we had decided to go ahead and make the relationship between the 2 tables 'defining' not only would DeveloperSeq be a part of the Game table, but it also would become a part of the Primary key of the Game table. In the Gamestatus system it would be proper to make the 1-M between Developer and Game a defining relationship, since we already answered the question of whether or not a Game could exist without a Developer **(NO,**

remember?). For the purposes of this tutorial I didn't make it defining for a couple of reasons.

The first reason is that we are using MySQL, and MySQL does not provide for 'Declarative Referential Integrity (DRI) unless you are using the InnoDB engine. I'm going to assume for the purposes of this tutorial that we're using the MyISAM record manager, which is the default with MySQL, and for many people in a hosted environment, the only option available to them.

What is Declarative Referential Integrity (DRI), and should we be using it? DRI allows you to declare how tables are related to each other at the database level, and having done so, the database will stop you from doing things that violate those relationships, or will insure that the relationships are enforced. With MySQL you get this functionality by using the InnoDB engine.

For example, if we had DRI, the database engine would not let us put in a row in the Game table that had a DeveloperSeq that did not match a Developer in the Developer table. It would also not allow us to delete a Developer that had existing games in the Game table, or as an alternative, it would do a "cascading delete" and delete all the game from the Game table that are related to the deleted Developer.

DRI is one of those things that is nice to have (most relational database engines do), but isn't essential. If you have it available, by all means use it, but with MySQL at least, in many cases it's just not an option. Since your application handles keys, as you will see, lack of DRI is not really that important. You do however, have to be aware of issues like the ones I described above when dealing with system functions like "Delete". I don't want people to misinterpret me, so let me reiterate it: if you have the option of using a feature like DRI you should, and I firmly believe in putting as much of the application into the database as I possibly can. If you can use InnoDB, you should, because it will make your application simpler to code, and more robust. Not having it however, as the history of MySQL has proven, is not the end of the world, although it is something you must be aware of.

Considering relationships

The database would probably be more robust if it was implemented using a defining relationship between Developer and Game, but there is a cost to doing so, and that is in making the size of the Game table primary key larger. As mentioned previously, the Game table would then **have DeveloperSeq as part of it's primary key**, which would mean that any tables related to Game in a 1-M relationship would now be multi-segmented (since they would inherit the full key of Game, whatever that might be). A defining key also makes key allocation more complicated, since you can't simply use AUTO_INCREMENT to give you a fresh sequential primary key number whenever you

need to insert a new row. For these reasons, I've kept the relationship non-identifying, even if I know in the back of my mind that the *real* relationship is 'identifying'. We'll return to this idea when we finally deal with the thorny issue of how you handle a many to many resolver table.

Why, you may ask, is any of this important to consider? Going back to the earlier discussion about Detail forms, I mentioned that some Details require related Lists to be a part of their page, and Developer is one of those pages. The way we'll handle this is to add a list to the bottom of the Developer Detail, that lists any Games developed by that Developer. We know that this is called for, because Game depends on first having a developer.

Adding Games

With the design of the system clear in their minds, the Gamestatus development team begins to code the Administration function for "Games". I'm going to assume that a set of Administration pages for Developer and admin functions supporting maintenance for all the other tables with a similar relationship to Game (Publisher, Genre, Status) are also already complete. I like to refer to tables like these as **"Lookup" tables**, because they typically provide a descriptive element for your main entit(ies). The appropriate value can be "looked up" in the related table, which often has a simple key/name/description structure.

When a table primarily exists on the Many side of a relationship, it is usually a good indication that it is a "Lookup" table. When approaching the development of your system, it's best to build your administration scripts for the lookup tables first, since your primary entities will be dependent on having a set of values already loaded. It is usually pretty simple to do the administration for a lookup table, and as a group, these tables lend themselves to the development of a generic function or class library to handle them.

As an element on a form, the lookup table can be handled from the user interface perspective as a drop-down list. In the case of the Game detail form, loading all the lookup tables into drop-down list boxes is a substantial part of the work. So we begin by considering how we might write a generic function that can be used to return us the desired output.

Coding Lookups

In order to come up with a generic function to handle Lookup tables on our input Forms, a good way to start is to consider the inputs we need.

1. First, we need to know what the lookup table name is.
2. Second, we need the name of the primary key column.
3. We'll also need the name of the column which will have our name or description. The values from this column will be displayed in the drop down list.
4. Because we will render the html for the dropdown, we also need to know which item in the drop down list was selected, in the case that we're displaying an existing row. We'll use an optional parameter, which by default is passed as an empty string or NULL.
5. The function will return a string which is all the HTML we need to display the drow down listbox.

common.function.php

PHP Example: (!)

```
/**
*************************************************************************
*
*
common.function.php                                                    *
*
==========================                                             *
* Copyright (c) 2004 by David
Rolston  (gizmo@gizmola.com)              *
*
http://www.gizmola.com                                                 *
*
*
* This program is free software. You can redistribute it and/or modify
*
* it under the terms of the GNU General Public License as published by
*
* the Free Software Foundation; either version 2 of the
License.       *
*************************************************************************
*
Questions or comments can be left at
http://forum.gizmola.com           *
*************************************************************************
*
*/

/**
* common.function.php is the gamestatus system function library
* @package GameStatus common.function.php
* @author David Rolston <gizmo@gizmola.com>
* @copyright 2004, David Rolston
*/

/**
* function makelookup
```

```
*
* makelookup is a generic html select (drop down list) generator.  It
* assumes that that the global database connection variables are in
* global scope prior to calling the function. example:
* <code>
* <?php
* makelookup('Status', 'StatusCode', 'Description');
* ?>
* </code>
*
* @author David Rolston <gizmo@gizmola.com>
*
* @param  string $table name of lookup table. Also used for the select
name=
* @param  string $key name of primary key column of lookup
table.  Becomes the value= for each option.
* @param  string $desc name of Description column from lookup
table.  Displayed in the drop-down List of options.
* @param  string $keyval optional param, that will be the currently
'selected' item in the drop down list.
*
* @return string An html <select> is returned, populated with
<option>'s set to the values of the $desc column
*
* @global dbhost mysql host string
* @global dbname mysql database
* @global dbuser mysql user
* @global dbpasswd mysql user password
*
*/
function makelookup($table, $key, $desc, $keyval = '') {
  global
    $dbhost,
    $dbname,
    $dbuser,
    $dbpasswd;

  $sql = "SELECT $key, $desc FROM $table ORDER BY $desc";
    $dbh = mysql_connect($dbhost, $dbuser, $dbpasswd) or die("Could not
connect");
    mysql_select_db($dbname,$dbh) or die("Could not select database");
    $rslt = mysql_query($sql, $dbh) or die("Query failed");
    $s = '<select name="'.$table.'">'."n";

    while ($row = mysql_fetch_assoc($rslt)) {
      //
        // Output the options
        //
        $s .= '<option';
        if ($row[$key] == $keyval)
          $s .= ' selected';
        $s .= ' value="'.$row[$key].'">'.$row[$desc].'</option>'."n";
    }
    $s .= '</select>'."n";
  return $s;
}
?>
```

This is the first time in the gamestatus system where we have strong need for a function. The reason for this, is that the process of generating an html option list for a lookup table needs to be repeated for several foreign keys in the Game table (Status, Genre, Developer, Publisher) so it's a good candidate for writing a function that can do the job, every time we need a new drop down list populated with the values from our lookup table.

In order to organize our project, it's best to put functions or classes in seperate files that you can include when needed. We'll call our Gamestatus function library *common.function.php.*

Documenting your code

I'm going to digress for a moment and talk about documentation. Every script can benefit from comments, but in the case of classes and functions, this is especially true. You may need to modify these components later, or reuse them. Having some form of documentation in the comments of the script will be very useful when you're looking at your code later. Luckily, there is a popular open source php documentation generator called phpDocumentor that can help us, so long as we get in the habit of putting comment blocks into our code. You simply need to follow a few conventions for how you add comments just prior to your function definitions, and add tags that phpDocumentor can use to build your documentation from.

phpDocumentor is a php package that will parse your php files and generate documentation in a variety of different formats. Visit the phpDocumentor site for more information: http://www.phpdoc.org.

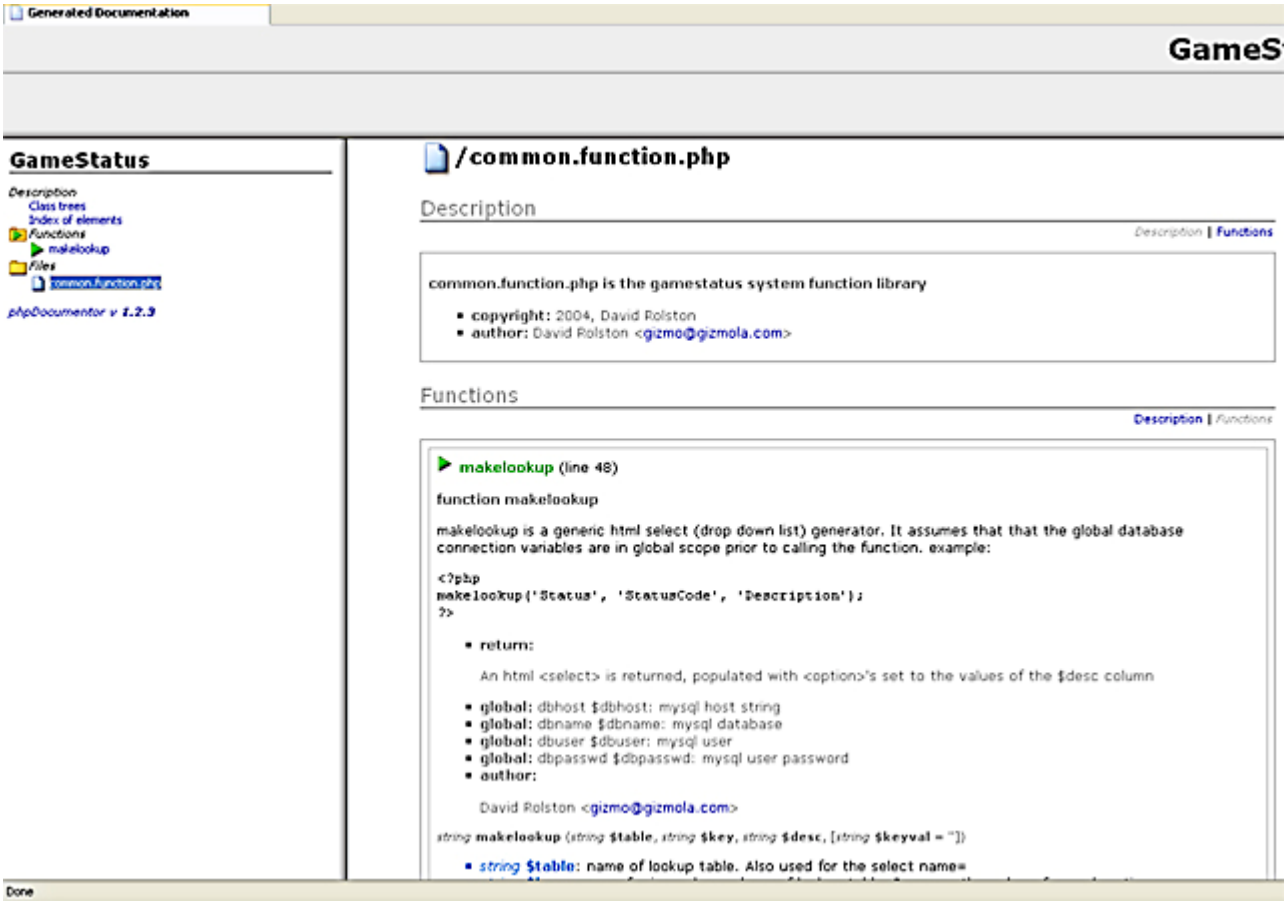
Example doc-block for phpDocumentor

PHP Example: (!)

```
/**
* function makelookup
*
* makelookup is a generic html select (drop down list) generator.  It
* assumes that that the global database connection variables are in
* global scope prior to calling the function. example:
* <code>
* <?php
* makelookup('Status', 'StatusCode', 'Description');
* ?>
* </code>
*
```

```
* @author David Rolston <gizmo@gizmola.com>
*
* @param  string $table name of lookup table. Also used for the select
name=
* @param  string $key name of primary key column of lookup
table.  Becomes the value= for each option.
* @param  string $desc name of Description column from lookup
table.  Displayed in the drop-down List of options.
* @param  string $keyval optional param, that will be the currently
'selected' item in the drop down list.
*
* @return string An html <select> is returned, populated with
<option>'s set to the values of the $desc column
*
* @global dbhost mysql host string
* @global dbname mysql database
* @global dbuser mysql user
* @global dbpasswd mysql user password
*
*/
```

You start out with two sections that let you provide an introduction or title followed by a description of what the function does. This can include an example use block, between the **<code>** and **</code>** html block. Documentation tags, all begin with the **@** character. The important tags here, are the @param, @return, and @global, which indicate clearly what the parameters are, what the function returns, and whether or not there are any global variables one should be aware of. I like to avoid global variables, but for gamestatus, we already assume that the MySQL database credentials will be available via inclusion of the config.php file (See Part I). Documenting this makes this dependency very clear, so if we decide later to try and use the function in another project, there will be no surprises. If you get in the habit of documenting your code in this fashion, you can use phpDocumentor to whip up cool looking html docs like this:



function makelookup

PHP Example: (!)

```
function makelookup($table, $key, $desc, $keyval = '') {
  global
    $dbhost,
    $dbname,
    $dbuser,
    $dbpasswd;

  $sql = "SELECT $key, $desc FROM $table ORDER BY $desc";
    $dbh = mysql_connect($dbhost, $dbuser, $dbpasswd) or die("Could not
connect");
    mysql_select_db($dbname,$dbh) or die("Could not select database");
    $rslt = mysql_query($sql, $dbh) or die("Query failed");
    $s = '<select name="'.$table.'">'."n";

    while ($row = mysql_fetch_assoc($rslt)) {
      //

        // Output the options
        //
```

```
        $s .= '<option';
        if ($row[$key] == $keyval)
          $s .= ' selected';
        $s .= ' value="'.$row[$key].'">'.$row[$desc].'</option>'."n";
    }
    $s .= '</select>'."n";
  return $s;
}
?>
```

If it's not already clear from the comments, **function makelookup** performs a database query of the table you pass as the param. It selects the primary key, and the description (ordering by description) using the column names passed as parameters, by building the SELECT statement and assigning it to the local $sql variable. It then issues the MySQL query and fetches the results using mysql_fetch_assoc(). Mysql_fetch_assoc() is a function that returns the fetched row in an associative array, keyed by column name.

If you remember Part I, we used mysql_fetch_array($rslt, MYSQL_ASSOC) to constrain it so that it only returned an associative array to us, rather than the default behavior of providing an array that has both standard numerically indexed elements AND the associative keys. In fact, mysql_fetch_assoc() is simply a wrapper around the mysql_fetch_array() which passes the optional MYSQL_ASSOC constant for us. The Gamestatus developers agree that from here out, they will use mysql_fetch_assoc() for fetching rows from a result set, in almost all cases, I suggest you do as well.

When in doubt, use mysql_fetch_assoc() to fetch rows from a MySQL query. Your code will be cleaner and clearer, and less likely to break if you modify your database when you use column names as the keys to your row[] array.

The primary purpose of the function is to build the html select options as it fetches each row. If the optional $keyval param is passed, the primary key column ($key) is checked to see if the $keyval matches, and if so, we set that option to be 'selected' so that the html generated will look like this:

PHP Example: (!)

```
<option selected value="3">Action/Adventure</option>
```

Notice that the function is not using echo or print, but rather concatenating the output onto the local (private) function variables **$s**. When the function is complete, it returns $s. This way the calling script can call *makelookup()* whenever it wants to, and store the result in a string if the developer chooses. If you were using a template class, this approach would be very attractive.

GameDetail

Now it's time to start on our Game Detail script *gamedetail.php*. We start by copying the code from the developerdetail.php, and modifying the things that should be different. The philosophy of what a detail script does was covered in Part 1, so this should make sense to you by now.

I'm going to refactor the gamedetail.php script to illustrate how you go about the process of writing a detail in incremental fashion, testing along the way. For this reason, you should pay attention to the versions of the script. The first version is gamedetail1.php

gamedetail1.php

PHP Example: (!)

```
<?php

$m = $_GET["mode"];
$sort = $_GET["sort"];
$seq = $_GET["seq"];
$s = $_GET["status"];

include("adminheader.php");
require("config.php");
require("common.function.php");

if ($m == 'edit') {
    $dbh1 = mysql_connect($dbhost, $dbuser, $dbpasswd) or die("Could
not connect");
    mysql_select_db($dbname,$dbh1) or die("Could not select database");
    $q1 = "SELECT FROM Game where GameSeq=$seq";
  $rslt1 = mysql_query($q1, $dbh1) or die("Query failed");
    $row1 = mysql_fetch_assoc($rslt1);
  /* Free resultset */
  mysql_free_result($rslt1);
} else {
    $seq = 0;
}
/* Closing connection */

$backurl = 'gamelist.php';
$posturl = 'gamepost.php';
$addurl = "gamedetail.php?mode=add";

if (isset($sort)) {
    $backurl .= "?sort=$sort";
    $posturl .= "?sort=$sort";
    $addurl .= "&sort=$sort";
}

echo '<a href=$backurl>Back</a>  <a href=$addurl>Add New
Game</a><br><br>';
```

```
echo "<form name="form1" method="POST" action="$posturl">";
echo "<input type="hidden" name="seq" value="$seq">";
echo '<table>';
echo '<tr><th colspan="2">Game Detail</th></tr>';
echo "<tr><td>Seq:</td><td>$seq</td></tr>";
echo '<tr><td>Title:</td><td><input type="text" name="title" size="80"
maxlength="80"';
if ($m == 'edit') {
    echo " value='".htmlentities($row1['Title'],ENT_QUOTES)."'";
}
echo '></td></tr>';
echo '<tr><td>Status:</td><td>';
if ($m == 'edit') {
    echo makelookup('Status', 'StatusCode', 'Description',
$row1['StatusCode']);
} else {
    echo makelookup('Status', 'StatusCode', 'Description');
}
echo '</td></tr>';
echo '<tr><td>Website:</td><td><input type="text" name="website"
size="80" maxlength="255"';
if ($m == 'edit') {
    echo " value='".htmlentities($row1['Website'],ENT_QUOTES)."'";
}
echo "></td></tr>";

echo "<tr><td colspan=2 align=center><input type='submit'
name='SubmitFrm' value='Save'></td></tr>";
echo "</table>";
echo "</form>";
if (isset($s)) {
    echo "Edit Status: $s <br>";
}
if ($m == 'edit') {
    $addUrl = "gamedetail.php?mode=add&seq=$seq";
    if (isset($s))
        $addUrl .= "&sort=$s";
}
?>
```

Note this small section of includes:

PHP Example: (!)

```
include("adminheader.php");
require("config.php");
require("common.function.php");
```

We will be using our common.function.php function library and *function makelookup* to generate drop down lists for our lookup tables, so the first thing we need to do is include it using require. Note that we require it AFTER we have required config.php, so that the global variables we need (as documented) will be available prior to defning our function.

Sometimes people are confused about what include and require actually do (see phpFreak's recent Tutorial!). The way to think of it, is that, it's as if the included script was sitting in your paste buffer, and at the point of the include, you hit your paste key, and all the code was copied in the script at that exact point. A variable from the parent script that is used in the include script, must be defined before the include, or when the include script code is run, that variable will have no value. You can minimize this issue by avoiding the use of global variables, and sticking to functions and classes as much as possible.

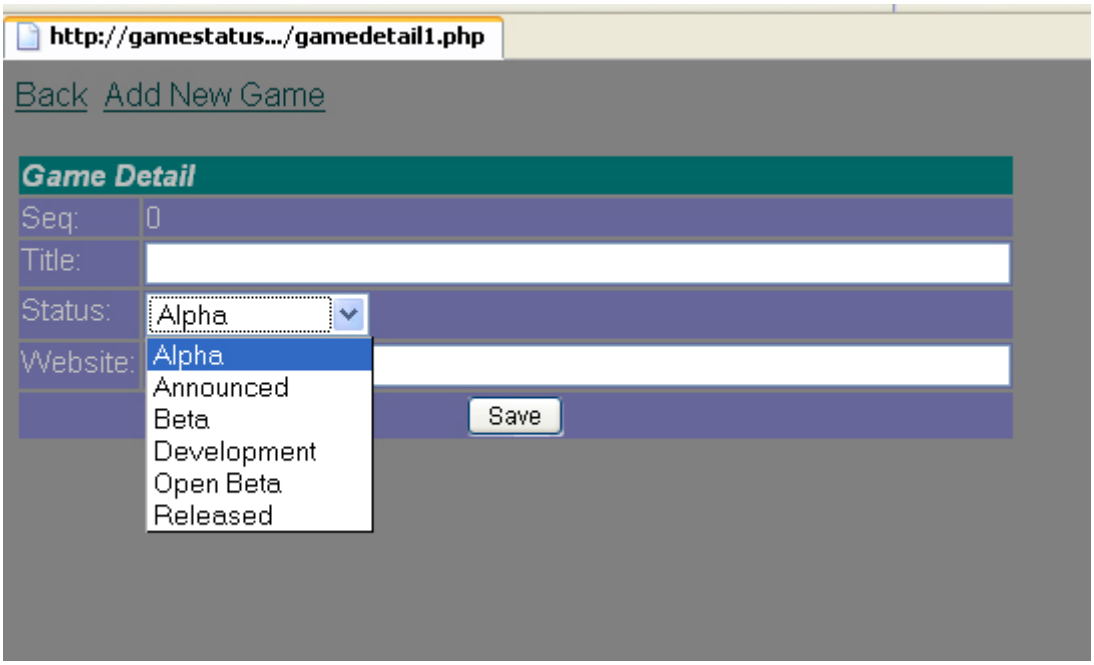
Integrating the makelookup function

Now we're ready to use *makelookup* in the form. We drop in a new table row, and call the function with the appropriate set of functions. If our mode=edit, then we will need to pass the optional param that tells the function to select the value that was already stored in the database. Otherwise we can omit that param.

PHP Example: (!)

```
if ($m == 'edit') {
    echo makelookup('Status', 'StatusCode', 'Description',
$row1['StatusCode']);
} else {
    echo makelookup('Status', 'StatusCode', 'Description');
}
```

We alter our first input column so that it reflects that we will want to record the Game title, and save the script, so we can do a quick test and make sure that our *makelookup* function is working.



Things are going well so far, as we can see that our status values are available in the the dropdown list. With that working we'll go ahead and add the rest of the elements we needed to match the structure of our Game table. Here is where the power of functions becomes evident... going from version 1, to version2 took no more than a minute or 2, to check the names of the tables, and their columns, and call the functions in the proper places so that all of our html selects work.

gamedetail2.php

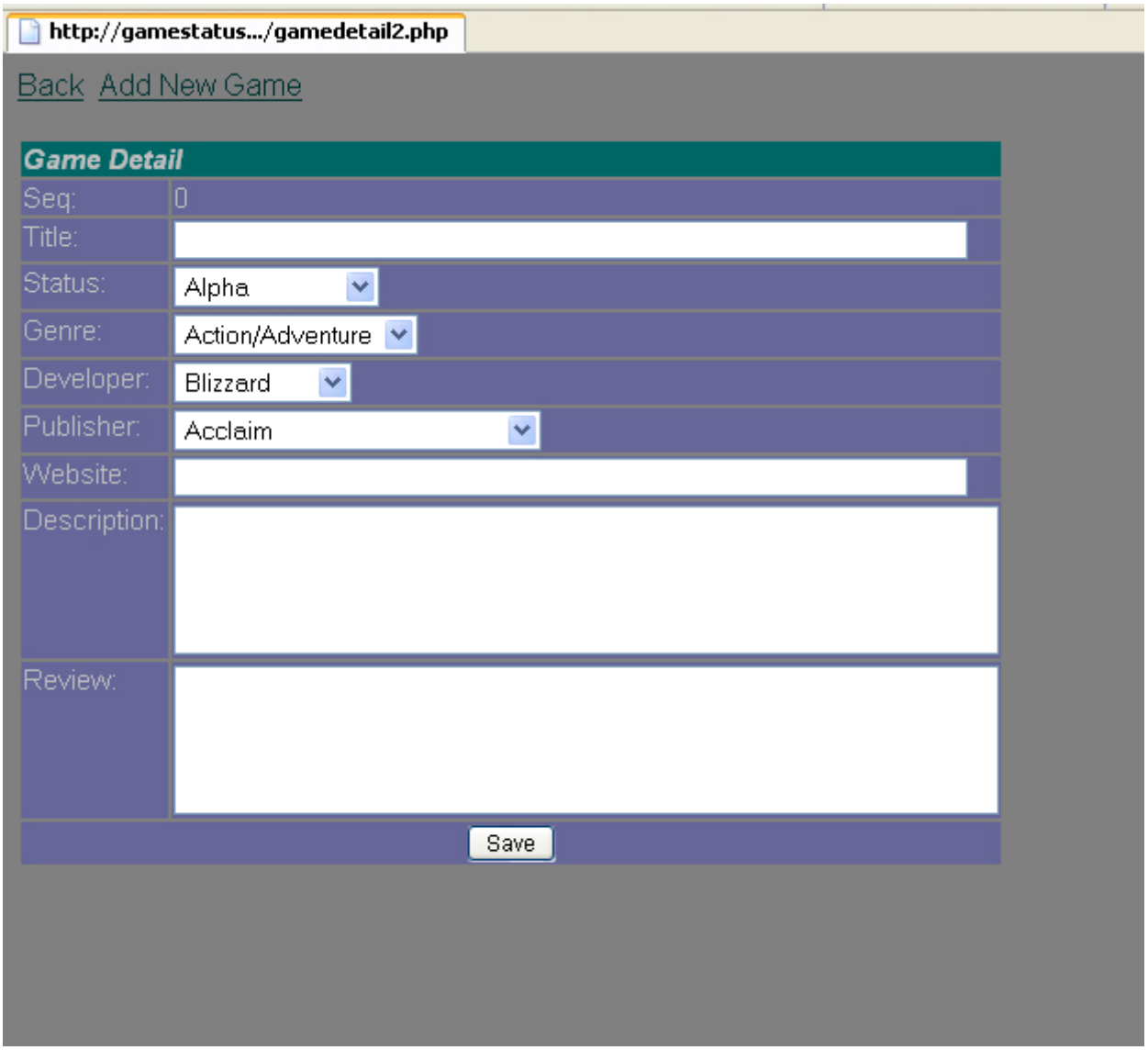
PHP Example: (!)

```
<?php

$m = $_GET["mode"];
$sort = $_GET["sort"];
$seq = $_GET["seq"];
$s = $_GET["status"];

include("adminheader.php");
require("config.php");
require("common.function.php");

if ($m == 'edit') {
    $dbh1 = mysql_connect($dbhost, $dbuser, $dbpasswd) or die("Could
not connect");
    mysql_select_db($dbname,$dbh1) or die("Could not select database");
    $q1 = "SELECT * FROM Game where GameSeq=$seq";
  $rslt1 = mysql_query($q1, $dbh1) or die("Query failed");
    $row1 = mysql_fetch_assoc($rslt1);
  /* Free resultset */
  mysql_free_result($rslt1);
```

```
} else {
    $seq = 0;
}
/* Closing connection */

$backurl = 'gamelist.php';
$posturl = 'gamepost.php';
$addurl = 'gamedetail.php?mode=add';

if (isset($sort)) {
    $backurl .= "?sort=$sort";
    $posturl .= "?sort=$sort";
    $addurl .= "&sort=$sort";
}

echo "<a href=$backurl>Back</a>  <a href=$addurl>Add New
Game</a><br><br>";
echo "<form name="form1" method="POST" action="$posturl">";
echo "<input type="hidden" name="seq" value="$seq">";
echo '<table>';
echo '<tr><th colspan="2">Game Detail</th></tr>';
echo "<tr><td>Seq:</td><td>$seq</td></tr>";
echo '<tr><td>Title:</td><td><input type="text" name="title" size="80"
maxlength="80"';
if ($m == 'edit') {
    echo " value='".htmlentities($row1['Title'],ENT_QUOTES)."'";
}
echo '></td></tr>';
echo '<tr><td>Status:</td><td>';
if ($m == 'edit') {
    echo makelookup('Status', 'StatusCode', 'Description',
$row1['StatusCode']);
} else {
    echo makelookup('Status', 'StatusCode', 'Description');
}
echo '</td></tr>';

echo '<tr><td>Genre:</td><td>';
if ($m == 'edit') {
    echo makelookup('Genre', 'GenreSeq', 'Name', $row1['GenreSeq']);
} else {
    echo makelookup('Genre', 'GenreSeq', 'Name');
}
echo '</td></tr>';

echo '<tr><td>Developer:</td><td>';
if ($m == 'edit') {
    echo makelookup('Developer', 'DeveloperSeq', 'Name',
$row1['DeveloperSeq']);
} else {
    echo makelookup('Developer', 'DeveloperSeq', 'Name');
}
echo '</td></tr>';

echo '<tr><td>Publisher:</td><td>';
if ($m == 'edit') {
    echo makelookup('Publisher', 'PublisherSeq', 'Name',
```

```
$row1['PublisherSeq']);
} else {
    echo makelookup('Publisher', 'PublisherSeq', 'Name');
}
echo '</td></tr>';

echo '<tr><td>Website:</td><td><input type="text" name="website"
size="80" maxlength="255"';
if ($m == 'edit') {
    echo " value='".htmlentities($row1['Website'],ENT_QUOTES)."'";
}
echo "></td></tr>";

echo '<tr><td>Description:</td><td><textarea name="description"
rows="4" cols="80">';
if ($m == 'edit') {
    echo htmlentities($row1['Description'],ENT_QUOTES);
}
echo "</textarea></td></tr>";

echo '<tr><td>Review:</td><td><textarea name="review" rows="4"
cols="80">';
if ($m == 'edit') {
    echo htmlentities($row1['Review'],ENT_QUOTES);
}
echo "</textarea></td></tr>";

echo "<tr><td colspan=2 align=center><input type='submit'
name='SubmitFrm' value='Save'></td></tr>";
echo "</table>";
echo "</form>";
if (isset($s)) {
    echo "Edit Status: $s <br>";
}
if ($m == 'edit') {
    $addUrl = "gamedetail.php?mode=add&seq=$seq";
    if (isset($s))
        $addUrl .= "&sort=$s";
}
?>
```

We add more calls to *makelookup* for our other drop downs, and then we add 2 html form textareas: one for Description, and the other for our Review. Remember that the Game table has Text columns for these two values, since we are assuming that GameStatus will eventually have these, and they could very well be hundreds or thousands of words long, perhaps even with embedded html tags (if we choose to support that). The TextArea is the right html form element to use when we want to input data that will be stored in a TEXT column. Having made these changes, we test again, and now the form matches the columns in the Game table.



What about the Many to Many?

It has been a long and winding road, but we arrive at last at the question of Many to Many relationships. In the case of the GameStatus website, we have such a relationship, between the entities **Game** and **Platform**. If you recall, I mentioned in Part 1, that many to many relationships have to be resolved by an intermediary table. In the GameStatus data model, that table is *GamePlatform*. We want to indicate for a game, the number of Platforms it will be released on. To do so, in the database, we'll need to insert a GamePlatform row with the corresponding GameSeq for our Game, and the corresponding PlatformSeq of the Platform upon which that Game was released (PC or Xbox, or PS/2, etc).

From a database standpoint, what we need to to accomplish is not complicated, however, integrating this into our form in a simple user friendly manner isn't so simple. Unfortunately, HTML's form elements are functionally limited, and there's no way to bundle up a set of form widgets or enhance them without using javascript.

We could attack the problem using entirely serverside logic, and a form that is continuously submitted back to itself and achieve similar functionality, but I don't find that to be a very user friendly interface. Our goal with this system is to provide administrative functionality. It's therefore acceptable for us to use Javascript, if it will enhance our user interface and make the system more usable, since we can expect our users to access the admin system using a javascript enabled browser.

For quite a while the debate between using javascript and not using it has been raging amongst web developers. On one hand, javascript has the potential to be misused in ways that can be highly annoying, and some people feel that the only way to handle this is to turn javascript off. It seems that developers and users in general have come to agree that they want the most intuitive interface possible, and basic html, while safe, is putting a lot of users to sleep. Many developers have chosen to use javascript for this reason.

Javascript is entirely clientside technology. Any javascript in your page must be loaded up by the user's browser and executes there inside the javascript interpreter, that has been implemented by the developer of the user's browser.

I don't think javascript should be overused, or used for things that can be done just as well in html and css, and as a developer you need to understand what javascript can't do reliably. Making sure a form has been filled out correctly and disallowing a submit if its not, is one example where javascript can be helpful but can not be depended upon. The reason for this is that someone who wants to subvert your security could use an application that simply posts data to the target, bypassing your javascript validation functions entirely. With that said, there has recently been a lot more interest in using javascript to do cool interface tricks that are intuitive to users, and are otherwise impossible with just css and html. Google's gmail service is an example of just such an application, but there are many more. It seems that javascript is now once again acceptable.

Introducing The Chooser

In our case, using it provides a way to accomplish an intuitive interface to our many to many problem. We create a class that emits what we'll call a "selectChooser". The "selectChooser" will display all the available platforms in one box, and the Platforms chosen in a second box, and will use buttons to modify the selection. This is tricky stuff, but I'll walk you through it as we go.

Because I used phpdocumentor, this source code has a lot of comments, including a snippet of source code that illustrates how the class can be used, and what it does. Although I haven't talked previously about the power and benefit of using PHP's object oriented programming, there are a number of other tutorials on that subject in the tutorial library. I encourage you to read those (including one by yours truly) so that you have a good understanding of the subject. It's also worth looking into the advances made to PHP's OOP capabilities that were brought in with PHP 5, especially if you are building new applications, which in my opinion should now be built with PHP 5 in mind. With that said, the OOP you'll see here is based on PHP 4, but should run fine under 4 or 5.

chooser.class.php

PHP Example: (!)

```
<?php
//************************************************************************
**/
//
chooser.class.php                                                       */
//
==========================                                              */
// Copyright (c) 2004 by David
Rolston  (gizmo@gizmola.com)            */
//
http://www.gizmola.com                                                  */
//
 */
// This program is free software. You can redistribute it and/or modify
*/
// it under the terms of the GNU General Public License as published by
*/
// the Free Software Foundation; either version 2 of the
License.       */
//************************************************************************
**/
//
                                      //
// Questions or comments can be left at
http://forum.gizmola.com        //
//************************************************************************
**/

/**
* common.function.php is the gamestatus system function library
* @package GameStatus chooser.class.php
* @author David Rolston <gizmo@gizmola.com>
* @copyright 2004, David Rolston
*/

/**
* selectChooser creates linked selects as interface to a many-to-many
relationship.
*
* This class should be loaded with 2 arrays of name/value pairs.  In
```

```
union, these
* items should represent the full set of items available in resolving
the many to
* many relationship.  For example, for a shirt that is available in
multiple colors
* these arrays should represent collectively the set of all possible
colors.
* These are associative arrays keyed on the description or tag which
will be displayed
* in the Select List.  The value is the key for the item which should
match the key for
* the item in the lookup list.  example:
* <code>
* <?php
*  require('chooser.class.php');
* $color = array('Red' => 1, 'Blue' => 2, 'Green' => 3, 'Black' =>
'4');
* // According to database, shirt available only in Blue;
* $r_shirt_selected = array('Blue' => 2);
* $r_shirt_unselected = array('Red' => 1, 'Green' => 3, 'Black' => 4);
* $shirt_selectChooser = new selectChooser($r_shirt_unselected,
$r_shirt_selected);
* //emit javascript functions
* $shirt_selectChooser->outputjavascript();
* //emit selectChooser
* echo '<form>';
* echo $shirt_selectChooser->outputChooser();
* echo '</form>';
* ?>
* </code>
* Typically these arrays would be populated by a query that outerjoined
from the parent table
* through the many to many resolution table to the lookup table.  It is
then simple to go through
* the result and set up each array in a manner reflecting the current
state of the relationship
* as stored in the database. This provides methods to generate all HTML
and javascript code required.
*/
class selectChooser {
// Properties
/**
  * @access private
  * @var The form elements for the chooser will be created based on
this name. The selected list will be
    * named $listname_s[], the unselected list will be $listname_u[].
  */
var $listname;

/**
  * @access private
  * @var name/value associative array with unselected items from total
set
  */
var $rUnSelectL;

/**
```

```
  * @access private
  * @var name/value associative array with selected items from total
set
  */
var $rSelectL;

/**
  * @access private
  * @var total items in set (select + unselected)
  */
var $scSize;


/**
* php 4.x style constructor: selectChooser
*
* selectChooser requires 2 associative arrays:
* $rUnSelect (items in list that have not been selected), and $rSelect
(items in list
* that have been selected).
*
* @param reference to an associated array keyed by description,
reflecting Unchosen items
* @param reference to an associated array keyed by description,
reflecting Chosen items
* @param optional base name for the html selects generated.  Defaults
to 'list'.
*/
function selectChooser(&$rUnSelect, &$rSelect, $listname='list') {
    $this->listname = $listname;
    $this->rSelectL = $rSelect;
    $this->rUnSelectL = $rUnSelect;
    $this->scSize = (count($this->rSelectL) + count($this-
>rUnSelectL));
}

/**
* outputChooser
*
* This is the method that outputs the select lists.
*
* @param string default = 0. For the current page DOM, this number
reflects the embedded form which contains the selectChooser
* @return string The generated selectChooser HTML is returned as a
string.
*/
function outputChooser($formnbr=0) {
    $line =
"<table><tr><td>Unselected</td><td></td><td>Selected</td></tr>n";
    $line .= "<tr><td><select name="".$this->listname."_u[]"
size="$this->scSize" multiple>";
    if (count($this->rUnSelectL) > 0) {
        foreach($this->rUnSelectL as $key => $value) {
            $line .= "<option value='$value'>$key</option>n";
        }
    }
    $line .= "</select></td>";
```

```
    $line .= "<td><input type="button" value="  >>  "
onclick="move(document.forms[$formnbr].elements['".$this-
>listname."_u[]'],document.forms[$formnbr].elements['".$this-
>listname."_s[]'])"><br>";
    $line .= "<input type="button" value="  <<  "
onclick="move(document.forms[$formnbr].elements['".$this-
>listname."_s[]'],document.forms[$formnbr].elements['".$this-
>listname."_u[]'])"></td>";
    $line .= "<td><select name="".$this->listname."_s[]" size="$this-
>scSize" multiple>";
    if (count($this->rSelectL) > 0) {
        foreach($this->rSelectL as $key => $value) {
            $line .= "<option value='$value'>$key</option>n";
        }
    }
    $line .= "</select></td></tr></table>";
    return $line;
}

/**
* outputjavascript
*
* emits the javascript functions needed by a selectChooser.  Needs to
be called prior to
* calling selectChooser().
* If there are multiple selectChoosers only one needs to call
outputjavascript.  This would
* be a good candidate for a static class variable.
*/
function outputjavascript() {
$hd = <<<HEREDOC
<SCRIPT LANGUAGE="JavaScript">
<!-- Begin
sortitems = 1;  // Sort list items? (1=yes)

function move(fbox,tbox) {
    for(var i=0; i<fbox.options.length; i++) {
        if(fbox.options[i].selected && fbox.options[i].value != "") {
            var no = new Option();
            no.value = fbox.options[i].value;
            no.text = fbox.options[i].text;
            tbox.options[tbox.options.length] = no;
            fbox.options[i].value = "";
            fbox.options[i].text = "";
       }
    }
    BumpUp(fbox);
    if (sortitems)
        SortD(tbox);
}

function BumpUp(box)  {
    for(var i=0; i<box.options.length; i++) {
        if(box.options[i].value == "")  {
            for(var j=i; j<box.options.length-1; j++)  {
                box.options[j].value = box.options[j+1].value;
                box.options[j].text = box.options[j+1].text;
```

```
            }
            var ln = i;
            break;
        }
    }

    if(ln < box.options.length)  {
        box.options.length -= 1;
        BumpUp(box);
   }
}

function SortD(box)  {

    var temp_opts = new Array();
    var temp = new Object();

    for(var i=0; i<box.options.length; i++)  {
        temp_opts[i] = box.options[i];
    }
    for(var x=0; x<temp_opts.length-1; x++)  {
        for(var y=(x+1); y<temp_opts.length; y++)  {
            if(temp_opts[x].text > temp_opts[y].text)  {
                temp = temp_opts[x].text;
                temp_opts[x].text = temp_opts[y].text;
                temp_opts[y].text = temp;
                temp = temp_opts[x].value;
                temp_opts[x].value = temp_opts[y].value;
                temp_opts[y].value = temp;
            }
        }
    }
    for(var i=0; i<box.options.length; i++)  {
        box.options[i].value = temp_opts[i].value;
        box.options[i].text = temp_opts[i].text;
    }
}

function SelectListAll(box) {
    for(var i=0; i<box.options.length; i++) {
        if(box.options[i].value != "") {
            box.options[i].selected = true;
        }
    }
    return true;
}
// End -->
</script>
HEREDOC;
$hd .= "n"; //clean up source a bit
echo $hd;
}
} // end class selectChooser
?>
```

The example assumes that the chooser is meant to allow someone to choose from a set of available colors. One might imagine this being used in an apparel store, where the store admin is able to indicate what colors a particular shirt is available in, with the assumption that this information will then show up in the catalog, when a person is shopping in their shirt factory online store.

The $color array has no purpose other than to illustrate the idea, that the total universe of colors would be represented in an array. The chooser actually works with 2 arrays: the *selected* array and the *unselected* array. To understand how you would use the selectChooser, you need to think in terms of sets. The set of all colors, is the $color arrray above, and in your application would be the lookup table in your application. In the gamestatus.org site, it will be the *platform* table.

Items that would be stored in the database in the many to many table, populate the *selected* array. If you think in terms of sets, the *unselected* array is the portion of the entire set that **does not intersect** with the selected portion. Another way of thinking about this is that selected will always be a *subset* of the entire set of values in the lookup table (unless the selected set is empty). Conversely, *unselected* is the set of the lookup table **minus** the set of selected items.

Having this clear in your mind will allow you to visualize and comprehend the SQL statements you'll need when reading and writing to the database. You as the developer have to grapple with these ideas, but for the end user, things should be simple and intuitive, and the selectChooser provides a way of making it easy for people to move things from one bucket to another, and back again, which is another way you could visualize the set theory involved.

When we run the example script we get a functional chooser that lets you play with it. Under the hood, the selectChooser is really just a set of html form objects that are being glued together with javascript functions that take advantage of the Document Object Model (DOM) and javascript events. By putting it in a class we can forget about how it actually works, and think of it as a form widget that we can drop into our application whenever we need it. From the php point of view, we have to follow a few simple steps:

1. Create the object using the php **new** keyword.

   PHP Example: (!)

```
$shirt_selectChooser = new selectChooser($r_shirt_unselected,
$r_shirt_selected);
```

   In looking at the selectChooser class code above, if we want to understand how it works, we need to look for the constructor function, as this is the one that is called for us automatically by php when an object is created with new. The constructor has the same name as the class, so in this case, look at the code for **function selectChooser**

As you can see, all the code really does is set up the internal class variables of the object, most importantly, by creating internal references to the selected and unselected arrays we pass to it when we create our object.

2. Output the javascript functions.

   PHP Example: (!)
   ```
   $shirt_selectChooser->outputjavascript();
   ```

   This couldn't really be much easier, as it's a simple call the class function **outputjavascript**.

   As expected, when you look at the source code you find that the purpose of this function is to output a script block which contains javascript functions that will be used by the selectChooser. Javascript of course, is not php code, and trying to use the echo or print functions to emit the static html and javascript can potentially be painful, and unnatural if you use echo or print. Even if you used a gigantic string and tried to maintain all the natural javascript indention, you still can end up with something that ruins your efforts to maintain code that has clear indention blocks so that it's easy to read. A nice alternative is to use the PHP heredoc facility. Basically a heredoc is similar to a php interpolated string. You start with a heredoc tag (I used the word HEREDOC, but it could be any string constant you like) and the special heredoc assignment symbol "<<<".

   PHP Example: (!)

   ```
   $hd = <<<HEREDOC
   ```

   This line starts the assignment of our heredoc block to the variable $hd. What follows is all the html and javascript code we need, exactly as if that code had originally been in an html file. PHP keeps assuming that it's html until it sees the heredoc tag (the word HEREDOC in this code) on a single line with the php statement terminator.

   PHP Example: (!)

   ```
   HEREDOC;
   ```

   A HEREDOC allows you to place HTML or Javascript into your PHP source without having to exit your PHP block. The nice thing about a HEREDOC is that you can use php variables inside the heredoc block, and PHP will interpolate those variables and merge their values into the code.

We don't need the interpolation feature for the selectChooser javascript, but when you do have that requirement, heredocs are one of the best solutions to the problem of intermixing large amounts of html code and php. The other advantage is that you get a string variable you can then perform additional processing on, or return from a function, rather than just having the HTML output directly to the browser.

About the Javascript

I feel that I have to say a little bit about the javascript functions, while at the same time I don't want to try and turn this into a javascript tutorial. If you don't know javascript and have no idea what the language is like, the javascript functions will likely be hard to understand. Javascript is a language that was modelled after the C language, with a built in set of classes you can use to manipulate objects inside the client browser. This main built-in class is the "Document object model" or DOM. Through an understanding of the DOM it's possible to manipulate an html page in a variety of ways.

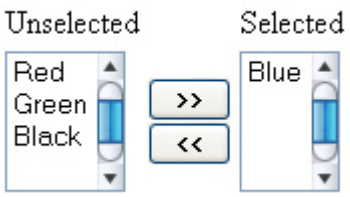
Again the important thing to keep in mind, is that the javascript interpreter runs inside the browser, on the client's machine. So in essence what we're accomplishing is giving the user the tools (the selectChooser) which they will download from the server, and then (as far as the server is concerned) wait until they've done manipulating the data inside their form, which in this case may include moving items around in the chooser. Once they're done, and finally submit the form, we'll need to clean up and make sure that the selectChooser is in a state that will cause the form data to indicate the choices they made while they were running the form "offline" through the clientside javascript code.

In order to better understand this, let's talk about what the selectChooser actually is. As I mentioned previously, it's an amalgam of html form objects that have been tied together with a combination of javascript code and a php class. The html form object that makes this all work is the html **<select multiple>** which displays a group of options in a list box, and lets you "select" as many of them as you want, by clicking on them, using click, shift-click or ctrl-click. If you think back to the discussion of sets I talked about before, this is a good metaphor as we're picking from items in one set, in order to add them to another.

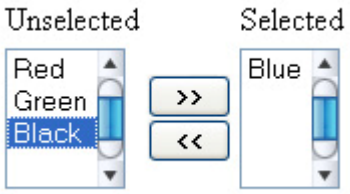
While the <select multiple> gives us a lot of the functionality we need purely by the nature of how it works thanks to good old html form functionality, the ability to move items from one <select multiple> to another can only be accomplished through client-side javascript. If we didn't use javascript, every time someone wanted to either "select" or "unselect" a choice from the list of total available choices, we'd have to submit the form to the server for processing, and pass back the new version that had the selected and unselected choices in their respective <select multiple> boxes. Using Javascript we can avoid having to do that, and let

the user select or unselect items as they please, as well as make any other changes to the other form fields, and only submit the form when they're done.

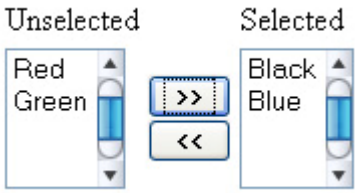
Aside from the two interconnected <select multiple> objects, we also need 2 buttons to execute the javascript that will actually do the moving of selections from one to the other. These four objects, wired together are what I call a **selectChooser**. Let's take a look at the demo code in action:



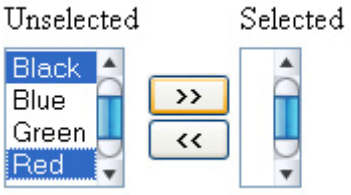
Here we have the simple select chooser as initialized at the start, with only the color blue in the selected list.



Now we're preparing to add "black" to the selected list.



Having clicked on the ">>" button, we've moved "black" to the selected list, and the selectChooser automatically resorts the items in each set by name.



This last picture just illustrates that the user can pick multiple items at the same time, and moving those items will work as expected.

Javascript move

PHP Example: (!)

```
function move(fbox,tbox) {
    for(var i=0; i<fbox.options.length; i++) {
        if(fbox.options[i].selected && fbox.options[i].value !=
"") {
            var no = new Option();
            no.value = fbox.options[i].value;
            no.text = fbox.options[i].text;
            tbox.options[tbox.options.length] = no;
            fbox.options[i].value = "";
            fbox.options[i].text = "";
       }
    }
    BumpUp(fbox);
    if (sortitems)
        SortD(tbox);
}
```

function move is the most important of the javascript functions in the small series of javascript functions we need to make the selectChooser work, because it is the function that actually moves items from one <select multiple> to another. Stepping back for a moment, to consider how a regular old html form would work, the way you populate a <select multiple> is to have **<option> </option>** items inside it. If you hardwired one of these it might look like this:

PHP Example: (!)

```
<option value="3">Three Years</option>
```

Outputting the correct option lists for the two <select> boxes is not a hard task with php, but those lists are then static. While you can "select" any items you wish in the select list, nothing happens until you actually submit the form. You should all know that when you submit a form to a php script using the html post method, the target php script makes the posted values available in the $_POST[] superglobal array. A <select multiple> can have multiple items selected however. If we have a form with this object:

PHP Example: (!)

```
<select name="colors" multiple>
```

When we look at the value of $_POST['colors'] in our target script, we'll find something disconcerting -- no matter how many items were selected, we'll only get one value!

The way that PHP solves this problem in concert with html, is that, believe it or not, you can specify the name of the <select multiple> as an *array* by using the **[]** array brackets at the end of the variable name. So to make a regular select work in PHP, you need to have the name be:

PHP Example: (!)

```
<select name="colors[]" multiple>
```

In doing so you will find that $_POST['colors'] is now an array that contains an element set to the "value=" for every <option> that was selected.

Looking again at the javascript move() function, you should notice that it takes two parameters:*fbox* and *tbox*. These will be the two select lists that comprise the chooser (FromBox and ToBox). The DOM provides an object model that allows you to inspect and set properties of these objects. You should notice that the following properties are used: fbox.options.length and fbox.options[]. fbox.options.length is the select list that items are being moved *from*. So the first thing move determines is how many items in the select list there are in total. Javascript provides this property as fbox.options.length. It then provides an array (fbox.options[]) containing each individual *<option> in the select.* An individual option has the properties "value" and "text" which correspond to whatever their html defintion would be. For example:

PHP Example: (!)

```
<option value="7">Maine</option>
```

In the DOM you might find that fbox.options[0].text would be "Maine". The other important option property is *selected* which will only be set when an option is "selected". Since we only want the chooser to move items that we've chosen from the list, it's important that this property be investigated. The other thing to keep in mind about a select multiple form object is that only the items that are selected when the form is submitted will be available in the $_POST. I'll elaborate on why this is important later on. At this juncture you should be able to follow the general flow of the move() function which is that it goes through each item in the *from list* checks its *selected* property and if set, makes a new option element in the *to box* and copies it's properties. Once copied, move() clears that item's properties in the *from list* and calls some clean up functions that reorder and sort the from and to boxes.

Hopefully, you now have a general idea of how javascript is providing us a means of providing a simple intuitive interface to a complicated database construct. Let's proceed with the gamestatus system and wire this new functionality into our form.

CONTINUED IN PART 3 OF THE SERIES