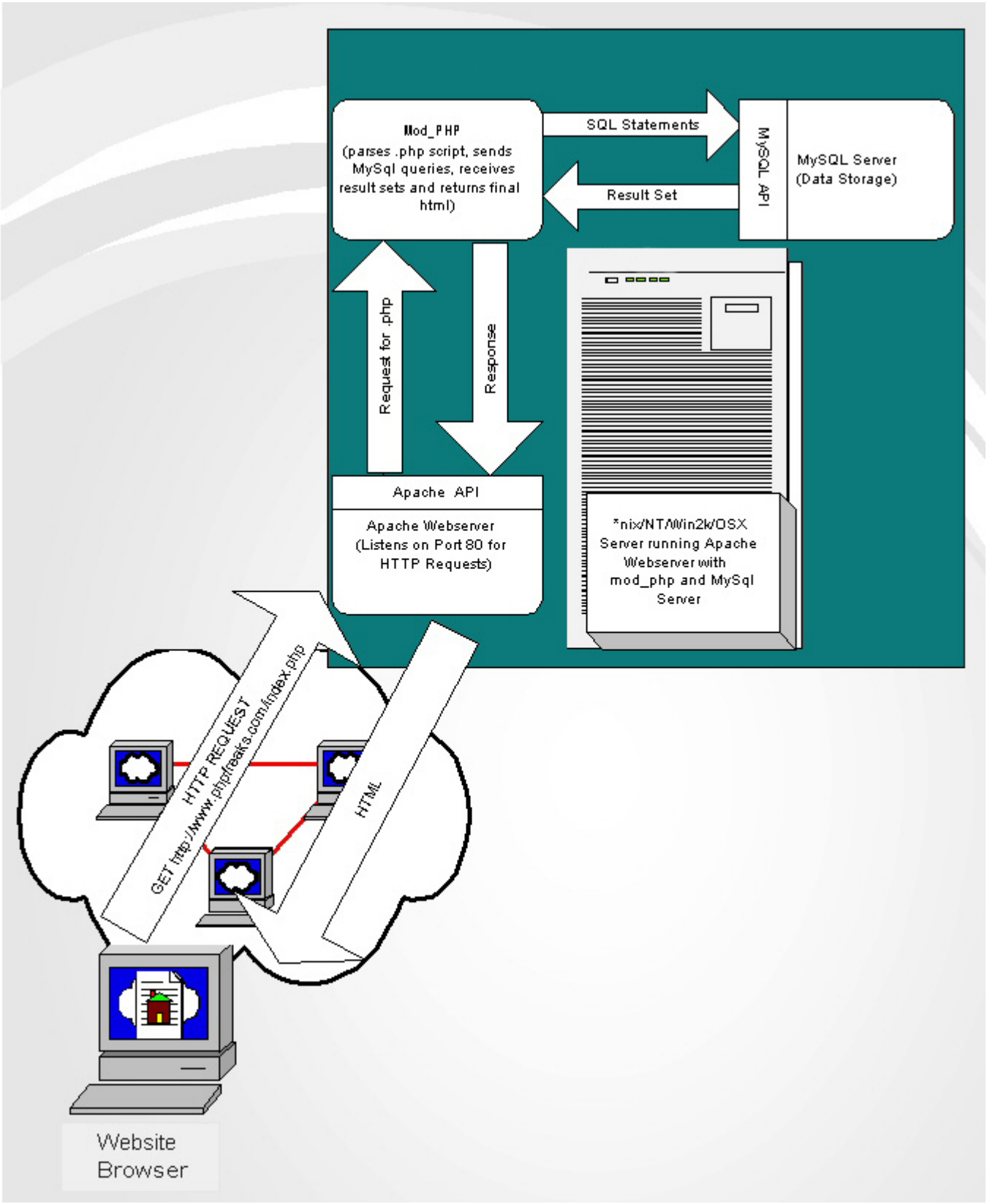
LAMP, MySQL/PHP Database Driven Websites - Part I

Navigate: PHP Tutorials > PHP

Author: Gizmola
Date: 07/24/2003
Version 1.4
Experience Level: Beginner

Introduction

One of the great "secrets" of almost all websites (aside from those that publish static .html pages) is that behind the scenes, the webserver is actually just one part of a two or three tiered application server system. In the open source world, this explains the tremendous popularity of the Linux-Apache-MySQL-PHP (LAMP) environment. LAMP provides developers with a traditional two tiered application development platform. There is a database, and a "smart" webserver able to communicate with the database. Clients only talk to the webserver, while the webserver transparently talks to the database when required. The following diagram illustrates how a typical LAMP server works.


Mod_PHP
(parses .php script, sends MySql queries, receives result sets and returns final html)
SQL Statements
Result Set
MySQL API
MySQL Server
(Data Storage)
Request for .php
Response
Apache API
Apache Webserver
(Listens on Port 80 for HTTP Requests)
*nix/NT/Win2k/OSX Server running Apache Webserver with mod_php and MySql Server
HTTP REQUEST
GET http://www.phpfreaks.com/index.php
HTML
Website
Browser

LAMP AND HTML

Like a traditional client-server application, LAMP allows developers to write client applications (in this case via server side PHP scripts) that communicate with a relational database server. If you're thinking this sounds complicated, you will have to accept that complexity is the price for creating the type of dynamic interactive websites that people want.

Once you face the inevitable (that your application will require a user interface, AND a database) you will eventually confront additional dilemmas. For example, people expect the same sort of GUI design and functionality that they'd find in a Windows, Mac or XWindows application.

Soon thereafter, you must confront the truth about HTML form objects. Interactive web applications require the use of HTML form objects, and the simple fact of the matter is that HTML form objects are not too smart. If HTML had richer user interface capabilites, people wouldn't have been compelled to attempt to supplement HTML with plugin technologies like ActiveX, Flash and Java.

Compared to the capabilities available in a Fourth Generation Windows language (4GL) like Visual Basic, Delphi, Centura or 4D or c++ class library, HTML form objects are simplistic and difficult to add state, modality and interactivity to. Aside from the basic out of the box behavior of an HTML form object, your only option for adding or modifying its behavior, is to use javascript. Even with the availability of javascript, a Web browser is hampered by the nature of the underlying request/answer protocol HTTP. HTTP was designed to be fast and lightweight, and connectionless. The original designers of the protocol just didn't imagine that the webrowser would become what it is today: the universal internet application client! For this reason developers have been fighting an ongoing battle to re-engineer the behavior of client server applications, both in terms of retaining connections (simulated via cookies and Sessions) and complicated interactive GUI's (via HTML with Javascript and server side functionality). This Tutorial will attempt to address a substantial number of these issues, addressing common relational database design concepts, and an enhanced GUI to complement the database design. Hopefully I'll pull this all together before the series is done.

Where do you start?

The most common mistake I see new php developers make, is that they jump into the php code for their application without giving much if any thought to how their database should be designed. This approach provides the the type of foundation, that one might expect if a builder were to begin construction of a skyscraper without an architect's blueprint. Pieces don't always fit properly, there are mysterious bugs, and worst of all, the application is often kludgy and inflexible for the end user, and being "codebound",

requires re-programming when even the smallest tweak or change is requested. These applications often have the stability of a tin shack, and the bigger they get the more unstable they become. Even applications with a good database design can suffer from some or all of these symptoms, just as a building based on an architect's blueprint can still have faults, but the chances are much greater that your application will be structurally sound, and adaptable if it's built upon a solid database design.

Many neophytes soon realize that they should store data in a relational database. They may even become somewhat familiar with a few basic SQL statements allowing them to insert data into the database and retrieve it. What they often miss, is that there is an art and science to good database design, and any good developer needs to spend some time learning the basics.

The database design is something that needs to be done prior to the start of coding.

Typically you have a requirements gathering phase of the project, which is a fancy way of saying, you brainstorm and figure out what the application must do, what you'd like it to do, and what you imagine it having the potential to do in the future. You may create some mockups or screen prototypes in this phase. The end result is that you should have a clear vision of what the application will do, and how it will work.

The next step is to design your database. MySQL and Postgresql are popular open source alternatives to commercial RDBMS's like Oracle, Sybase, Informix, Microsoft SQLServer or DB2/2. In this tutorial I use MySQL, but the concepts are equally applicable to any relational database engine. This is not a tutorial about relational database design, but I would urge you to read some books on the subject or seek out tutorials online. The buzzword for good database design is **Normalization**. When a database is designed properly it attains compliance with one or all of the rules of normalization. For example, a database that follows rules one through three of Normalization is known as being in *Third Normal Form.*

One webpage that gives a quick overview of normalization is http://www.datamodel.org/ There are many others which can be found with a quick Google search on data modelling, normalization, or entity relationship diagram. An excellent free tutorial on database design concepts is available here

Database design

During the database design phase of an application, experienced data modellers create what is called an Entity-Relationship or ER Diagram. Some will begin with a logical model prior to proceeding to the physical model. The goal of this stage is to properly model every entity the application will require, and establish the relationships between

the entities. Almost all entitities will become tables in the database, and the relationships between them will be the keys that allow you to join the tables together when the application's SQL select statements are executed.

At this point in the tutorial, I can only ask you to accept the statement that *the database design is the foundation of any good database application (and almost all dynamic websites are database applications).* Once you dig deeper into the role of database design, you will find the the key to a flexible data-driven application is not just the columns in your tables. Of equal importance is the relationships between the tables in the database. This is the reason that these systems are based on "Relational" databases. Tables have "relationships" to other tables, and these relationships are very important. With that statement in mind, one very common logical relationship between two entities is the "Many to Many" relationship. Here's some examples:

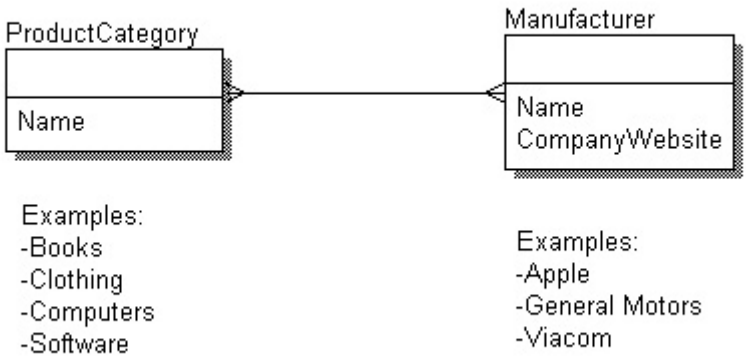
You're developing a PC mall/superstore website with Manufacturers listed by Product Category:

- A Manufacturer may make products in multiple Categories.
  For example, Motorolla makes Microprocessors AND Cell Phones.
- A Category may include many Manufacturers.
  "Book Sellers" include Amazon, Barnes and Noble, Crown and many more.

Thus the relationship between ProductCategory and Manufacturer is a "Many to Many" relationship: Each Category can include many Manufacturers, and each Manufacturer can make products in Many ProductCategories.

There are different diagram formats for creating an ER Diagram. In this tutorial I use the Information Engineering (IE) methodology/notation for those keeping score. One thing to note aout IE notation, is the *crows feet connector*, which indicates that the relationship to that table from the connecting table is a *many* relationship.

Here's a logical model of the Many to Many between a Category Entity and a Manufacturer entity in our hypothetical superstore application.



Proceeding with the design process, and moving from the logical to the physical database design, you quickly come to find that there is no way to form a "Many to Many" relationship in the physical model of a relational database. This is because all columns are atomic, or in other words, there is no column datatype that allows you to store multiple values in a single column of a database.

MySQL does offer a SET type, but for a number of reasons, in my opinion you should avoid using a SET column, not the least of which is that it makes the application inflexible and violates the atomic value principle.

Modeling a M-M

If a logical design can include a many to many relationship, then how do you create a physical database model that is functionally equivalent to it? The solution is to use a a table between the two entities to bridge the many to many (M-M) gap. I call this table a "many to many resolver". It is quite common for a complicated ERP/Accounting system like SAP/Peoplesoft or Oracle Financials to have hundreds to thousands of tables and the M-M relationships occur frequently. As you build complicated systems, the places where you will need a M-M resolver become fairly obvious, and the physical database design becomes second nature. However, even a small system with a few tables, often requires a M-M relationship between entities. I will return to the subject of M-M's later in the series.

Avoid this common mistake

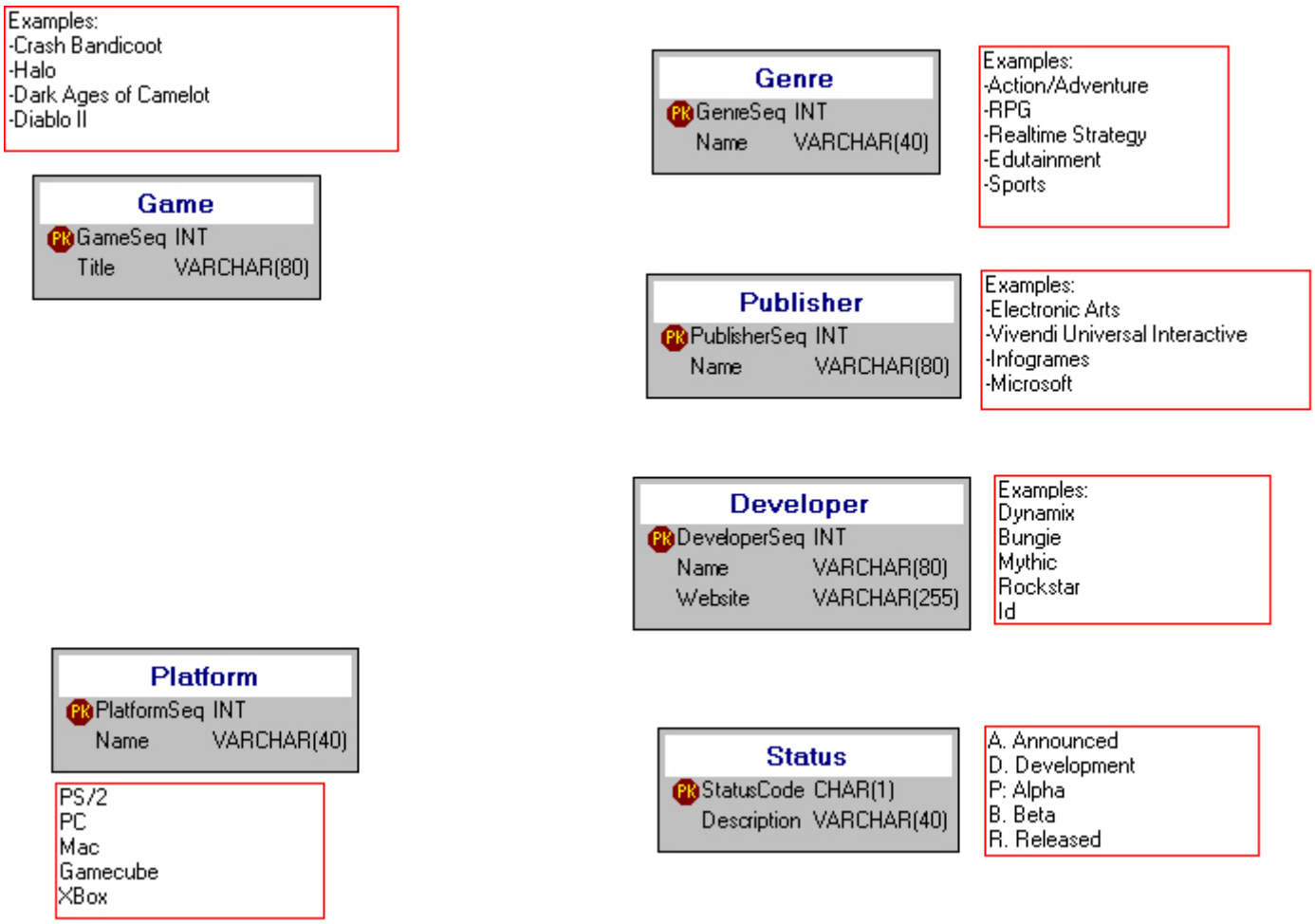
I frequently come across people who are trying to use a varchar column to store a comma seperated list of items, to get around having to normalize their database design to accomodate a Many to Many relationship. They then wonder how they can do queries on

that column without using LIKE '%somevalue%'. Without belaboring the point, if you have to use a like '%somevalue%' clause in your select statement, your database design is probably wrong. You've also guaranteed that the database will not be able to use an index, and if your database gets very large, performance will suffer. Proper implementation of a Many to Many as a set of normalized related tables will continue to perform well up to the capacity of the database server, no matter how large your database grows.

Our sample website: Gamestatus.org

The rest of this tutorial revolves around a website called Gamestatus.org. Gamestatus.org is our tutorial website created by some gamers to provide news and information about Computer games. The programmers making the site sit down and talk about features they want to have and one of the most important ones they can think of, is a page that lists the various Games and provides information about their development status, developer, publisher and genre. They decide that the game database will be the core of the site, and they want to be sure they design it right.

They create a list of Entities they think the database will need:
Game of course, The Publisher, the Developer, the Platform it will come out on, and the Genre of Game for category. One developer also figures that status is important to track, so they create a list of status codes they will use to keep track of where a game is in the development cycle. They put together a diagram listing the entities and figure out a few of the columns that each Entity will use to store information about it, using examples to help them understand how things should fit together. When they're done the diagram looks like this:



Adding the Relationships

Having these entities is a good start. They've been careful to keep entities seperate, and have given each entity a primary key so that any one instance (or row) in an entity can be seperately identified from any other row in the entity. They make sure that the attributes (columns) of each entity are directly related to it. One technique for figuring out which entities one requires, is to make short lists of examples. Typically an example should fit nicely as one row of a single entity. Consider whether the attributes of your entity actually belong to it. This is all part of the process of database modelling and design known as normalization. In the diagram each primary key is noted with a hexagon around the letters "PK".

Now they decide they're ready to move on to relating (ie. connecting) the different entities, by creating relationships between them. They also figure out what the cardinality of each relationship should be, and whether or not it is a defining relationship.

I realize you may not understand these terms, so make sure to go back and look them up later, while referencing the diagrams.

They start with Game and its relationship to Genre. They decide that a game can only be in one Genre. (This is debatable, but the Game industry itself through groups like the SPA categorize games as having a simple genre, and leave the consumers to figure it out.) Thus the relationship between a Game and Genre is that a Game can be classified with one Genre. Looking at the relationship between Genre and Game, the reverse is true. There are many RealTime Strategy Games like Warcraft, C&C, and Age of Empires. Clearly the relationship between Genre and Game is "one to many" or **1-M**.

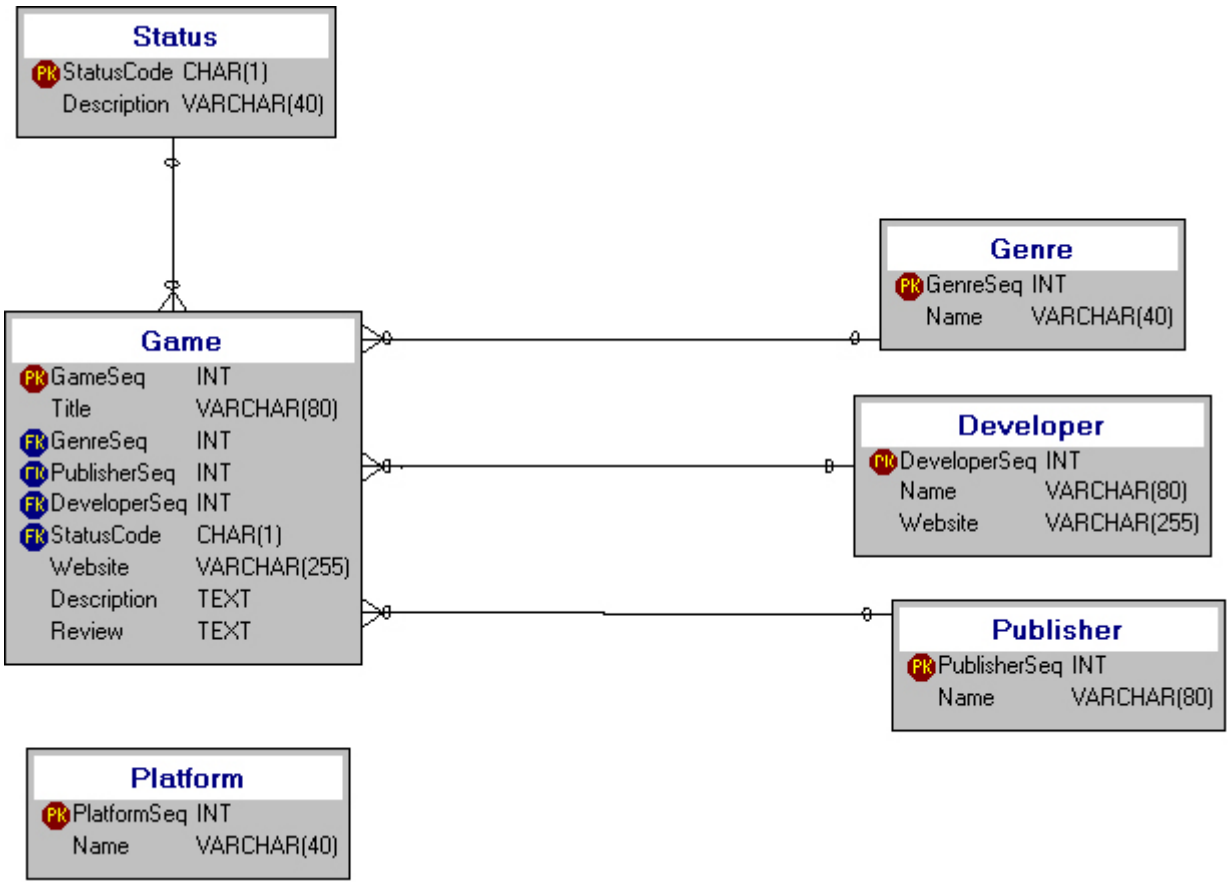
What happens in the "physical model" is that when Table A has a 1-M relationship with Table B, The Primary key of Table A becomes a "foreign" key in Table B. The primary key of Genre becomes part of the Games table, hence it is a key that is part of Games, but really belongs to another (foreign) entity.

It's important that you understand this basic relationship, and how it translates to the physical database model, because your queries will join the tables back together at select time using primary and foreign keys to connect the tables (entities) in the WHERE clause. This idea is discussed more fully later in the series.

Updating the model with relationships

The Gamesttus.org developers continue to relate the other tables in their diagram, and find that publisher and developer also have 1-M relationships to game. They also realize the same is true of Status: there will be many games in the database at any one time, that will be in Beta, for example. They add a few more attributes to Game: a column to store the official website for the game, and text columns to store a description, and a review.

When they're finished the model now looks like this:



Note the crows feet indicating the many side of the 1-M relationship, and how Game now includes the foreign keys of the related tables.

The only entity left is to relate is Platform, and they know that platform has a relationship to Game. Only the logical relationship between the two entities isn't a simple one to many. From the game side, they realize that while most games are released to one platform, some actually get released to multiple platforms: for example, Grand Theft Auto: Vice City is available for the PS/2 AND the PC. From the *Platform to Game* side of the relationship, **one Platform has many games** which have been released for it. They realize that they now have need of a many to many relationship.

What's a many to many resolver?

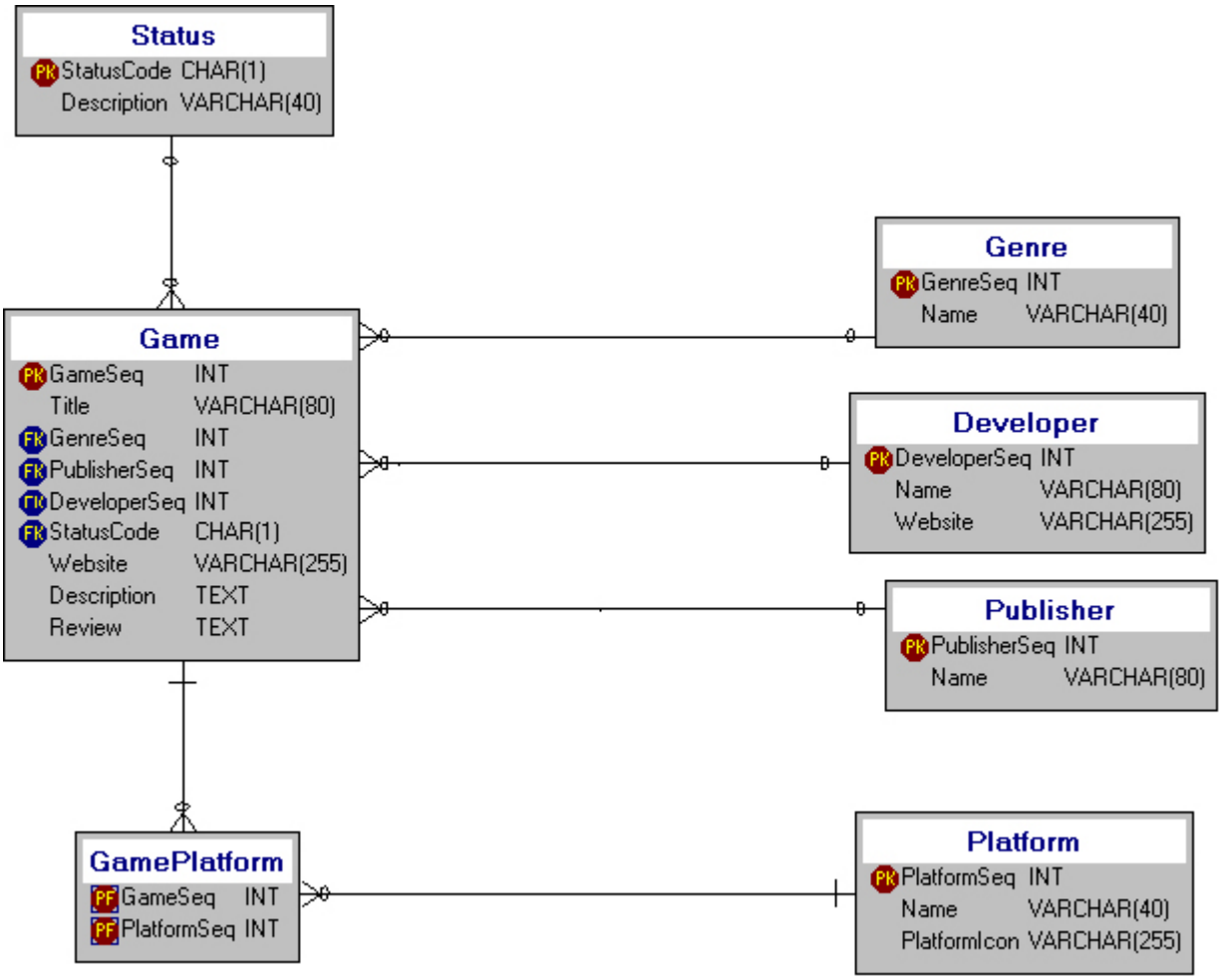
The way to handle a many to many in the physical model is to create a new table that will sit between Game and Platform. They decide to call this table GamePlatform to indicate its role as a M-M resolver. Since GamePlatform is a surrogate for the M-M relationship, you resolve it physically by relating each of the entities to the resolver table with a 1-M relationship. First they connect Game to GamePlatform as 1-M. GamePlatform then

receives GameSeq as a Foreign key. They make the same connection for Platform to GamePlatform (1-M) and GamePlatform also receives Platform as a foreign key.

While this would theoretically "work" in most databases, something isn't quite right, because GamePlatform has two (FK's) and no (PK's). The way to fix this in the data model is to make the relationships "dependent". This clarifies that GamePlatform is dependent on its relationship to Game. Another implication is that unlike other entities which might have a row that exists independently of the presence or absence of rows in another table, there can never be a row in GamePlatform without there first being a row in Game. The idea of dependence isn't exclusive to M-M's, it can be applied to any relationship between two tables. People often use the term Parent-Child to describe a dependent 1-M relationship, for example in a shopping cart application, there is often an order table and a lineitem table, where lineitem is dependent on order. If you have a lineitem row without a parent order, something has gone wrong.

Creating the Database and Tables

The Gamespot.org team adds GamePlatform, creates the relationships between it and the Game and Platform tables, and decides that this database design will support their "Game Status" page nicely.



The database will allow them to show a list of games sorted by Name, Status or Genre. They decide to display a set of icons with the game listing (one for each platform the game will be available on, and they add a menu to the homepage design that will let people list games for a single platform. They can also display the Publisher and Developer for a game, and can allow people to click to a list of other games the developer created, or via a link to the publisher, a page that shows all the games the Publisher has published.

With a relatively simple design they are able to support a number of different views of the information, at least conceptually. Now all they have to do is create the GameStatus database, and create the tables in their model.

PHP Example: (!)

```
[david@scrappy david]$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or g.
Your MySQL connection id is 15693 to server version: 3.23.56

Type 'help;' or 'h' for help. Type 'c' to clear the buffer.
```

```
mysql> create database gamestatus;
Query OK, 1 row affected (0.01 sec)
```

They login to mysql as root and create a database called gamestatus, and a user with full rights to the gamestatus database.

PHP Example: (!)

```
mysql> GRANT ALL ON gamestatus.* TO gsuser@localhost IDENTIFIED BY
"gsuserpw";
Query OK, 0 rows affected (0.00 sec)
```

Next they decide they will need a mysql user for their application. They decide to call this user *gsuser*. They craft a sql script that will create the database inferred from the design.

I used the low cost data modelling tool ($149) Dezign for databases to construct my ER diagrams, and to generate mostly trouble free DDL (Data Definition Language) SQL using it's SQL generation facility. Since the article was written, Dezign has been upgraded and the price raised to $229.

With a MySQL database user, and the proper access for that user to access a database, they are now ready to build their database. This can be done by running the script from a mysql command line, or by using a tool like SQLyog or phpMyAdmin. Use whatever database and tool is available to you, to build the database, if you're following along.

PHP Example: (!)

```
CREATE TABLE Publisher(
PublisherSeq INT UNSIGNED NOT NULL AUTO_INCREMENT,
Name VARCHAR(80),
PRIMARY KEY (PublisherSeq),
UNIQUE UC_PublisherSeq (PublisherSeq),
UNIQUE UC_Name (Name));


CREATE TABLE Platform(
PlatformSeq INT UNSIGNED NOT NULL AUTO_INCREMENT,
Name VARCHAR(40) NOT NULL,
PlatformIcon VARCHAR(255),
PRIMARY KEY (PlatformSeq),
UNIQUE UC_PlatformSeq (PlatformSeq),
UNIQUE UC_Name (Name));


CREATE TABLE Status(
StatusCode CHAR(1) NOT NULL,
```

```
Description VARCHAR(40),
PRIMARY KEY (StatusCode),
UNIQUE UC_StatusCode (StatusCode));


CREATE TABLE Developer(
DeveloperSeq INT UNSIGNED NOT NULL AUTO_INCREMENT,
Name VARCHAR(80) NOT NULL,
Website VARCHAR(255),
PRIMARY KEY (DeveloperSeq),
UNIQUE UC_DeveloperSeq (DeveloperSeq),
UNIQUE UC_Name (Name));


CREATE TABLE Genre(
GenreSeq INT UNSIGNED NOT NULL AUTO_INCREMENT,
Name VARCHAR(40) NOT NULL,
PRIMARY KEY (GenreSeq),
UNIQUE UC_GenreSeq (GenreSeq),
UNIQUE UC_Name (Name));


CREATE TABLE Game(
GameSeq INT UNSIGNED NOT NULL AUTO_INCREMENT,
Title VARCHAR(80),
GenreSeq INT UNSIGNED,
PublisherSeq INT UNSIGNED,
DeveloperSeq INT UNSIGNED,
StatusCode CHAR(1),
Website VARCHAR(255),
Description TEXT,
Review TEXT,
FOREIGN KEY (DeveloperSeq) REFERENCES Developer (DeveloperSeq),
FOREIGN KEY (GenreSeq) REFERENCES Genre (GenreSeq),
FOREIGN KEY (PublisherSeq) REFERENCES Publisher (PublisherSeq),
FOREIGN KEY (StatusCode) REFERENCES Status (StatusCode),
PRIMARY KEY (GameSeq),
UNIQUE UC_GameSeq (GameSeq));


CREATE TABLE GamePlatform(
GameSeq INT UNSIGNED NOT NULL,
PlatformSeq INT UNSIGNED NOT NULL,
FOREIGN KEY (GameSeq) REFERENCES Game (GameSeq),
FOREIGN KEY (PlatformSeq) REFERENCES Platform (PlatformSeq),
PRIMARY KEY (GameSeq,PlatformSeq));
```

Building the Admin System using the List-Detail-Post paradigm.

There are many different ways to handle the organization of scripts. Gamestatus.org will use a simple but effective paradigm, which I call "List-Detail-Post". We will develope a List-Detail-Post for administration of the Developer table.

There are many ways to provide security for an administrative application. That will not be covered here, but I'd recommend the session and membership tutorials here on phpfreaks.com. The simplest way would be to use an .htaccess file that specified an htpasswd file, and employs the browser's built in ability to provide "realms" security. Look at the Apache documentation for more information on **AuthType Basic** and **Require**

List-Detail-Post requires 3 different scripts for any one table or logical view. "List" is usually the first to be built, and provides a way of Listing all the rows in the table in a columnar format. From List, you can click on an individual row and call the detail script, which provide the full information for a single row, or begin the process of creating a new one. The third script is the "post" script, called from Detail when a new row is added, or an existing one is changed or deleted. In order to keep things secure, they use an htaccess file, create a directory off the root of the site called gsadmin, and puts the admin scripts inside it. They start with a few support files. First a .css file to provide a little bit of color for the admin screens.

admin.css

PHP Example: (!)

```
BODY {
    font-family : Arial, verdana,Helvetica, sans-serif;
    background-color : Gray;
    color : Silver;
}
A:ACTIVE {
    color : #003333;
}
A:LINK {
    color : #003333;
}
A:VISITED {
    color : #003333;
}
P {
    font-family :  Arial, verdana,Helvetica, sans-serif;
    font-size: 10pt;
    vertical-align :  top;
}
TR {
    text-align : left;
    background-color : #666699;
}
TD {
    font-family : Arial, verdana,Helvetica, sans-serif;
    font-size: 11pt;
    vertical-align : top;
    background-color : #666699;
}
```

```
TH {
    font-size: 11pt;
    background-color: #006666;
    font-style : italic;
}
TH.TITLE {
    font-size: 12pt;
    background-color: #006666;
    font-style : normal;
    text-align : center;
}
H1 {
    font-family : Arial, Helvetica, sans-serif;
    color: Black;
    font-size: large;
}
.Name {
  font-size: 12pt; }
.small{
  font-size: 8pt; }
input {
  font-family: arial,helvetica,sans-serif;
  font-size: 9pt; }
textarea {
  font-family: arial,helvetica,sans-serif;
  font-size: 9pt; }
```

This sets up some simple built-in styles to give the admin system a consistent look. We stick the link tag into a small include file called adminheader.php, which we'll include in the admin scripts.

adminheader.php

PHP Example: (!)

```
<HEAD>
<LINK REL=stylesheet HREF="admin.css" TYPE="text/css">
```

Adminheader.php illustrates one of the many interesting properties ofphp: there's no php whatsoever, but as usual the php interpreter handles the intermixing of html and php for you.

config.php

PHP Example: (!)

```
<?php
$dbhost = 'localhost';
```

```
$dbname = 'gamestatus';
$dbuser = 'gsuser';
$dbpasswd = 'password';
?>
```

In a real application, I would recommend the use of a database management class like ezSQL. For gamestatus.org we're rolling our own code, so this simple config file will help provide a small amount of indirection. If we have to change database details, the config.php would save us having to do search and replace in every script where database connections are required.

adminmenu.php

PHP Example: (!)

```
<?php include("adminheader.php"); ?>
<TABLE>
<TR></TR><TH colspan=2>GameStatus.org Admin Menu</TH></TR>
<TR></TR><TH>Item</TH><TH>Description</TH></TR>
<TR><TD><a href=developerlist.php>Developer List</a></TD><TD>List of
Developers</TD></TR>
</TABLE>
```

We start with a simple menu script to tie the various administration scripts together. Including the Adminheader.php brings in the use of our admin.css style sheet, since it has defined some style attributes for table elements like TH and TD. Starting with the Developer table, we start to develope a List Script. As we develope other scripts, we will need to edit adminmenu.php, but for now we will concentrate on Developer.

developerlist.php

PHP Example: (!)

```
<?php
    $s = $HTTP_GET_VARS["sort"];
    include("adminheader.php");
?>
<script language="javascript">
function confirmdelete(delurl) {
    var msg = "Are you sure you want to Delete this Row?";
    if (confirm(msg))
        location.replace(delurl);
}
</script>
</head>
<?php
    require("config.php");
```

```
    function loadlist($s) {
        Global $dbhost, $dbname, $dbuser, $dbpasswd;

        switch ($s) {
            case "name" :
                $sortby = "Name";
                break;
            case "website" :
                $sortby = "Website";
                break;
            default:
                $sortby = "DeveloperSeq";
        }

        $dbh1 = mysql_connect($dbhost, $dbuser, $dbpasswd) or
die("Could not connect");
           mysql_select_db($dbname,$dbh1) or die("Could not select
database");
        $q1 = "SELECT * FROM Developer ORDER BY $sortby";
           $rslt1 = mysql_query($q1, $dbh1) or die("Query failed");

        echo "<table><tr><th colspan=4>Developer List</th></tr>";
        echo "<tr><th><a href=".$_SERVER['SCRIPT_NAME'].">Seq</a></th>
        <th><a href=".$_SERVER['SCRIPT_NAME']."?sort=name>Name</a></th>
        <th><a
href=".$_SERVER['SCRIPT_NAME']."?sort=website>Website</a></th><th>
        </th></tr>n";
      while ($row1 = mysql_fetch_array($rslt1, MYSQL_ASSOC)) {
          echo "t<tr>n";
            $dtlUrl =
"developerdetail.php?mode=edit&seq=".$row1['DeveloperSeq'];
            $delUrl =
"developerpost.php?mode=delete&delseq=".$row1['DeveloperSeq'];
            if (isset($s)) {
                $dtlUrl .= "&sort=$s";
                $delUrl .= "&sort=$s";
            }
            echo "tt<td><a
href=$dtlUrl>".$row1['DeveloperSeq']."</a></td>";
            echo "tt<td><a href=$dtlUrl>".$row1['Name']."</a></td>";
            echo "tt<td><a href=$dtlUrl>".$row1['Website']."</a></td>";
        echo "<td><input type="button" value="Delete"
onClick="confirmdelete('".$delUrl."');"></td>t</tr>n";
        }
        echo "</table>n";
        /* Free resultset */
        mysql_free_result($rslt1);
        /* Closing connection */
        mysql_close($dbh1);
    }

  // Main Script
    $addUrl = "developerdetail.php?mode=add";
    if (isset($s))
        $addUrl .= "&sort=$s";
    echo "<a href=adminmenu.php>Back</a>  <a href=$addUrl>Add New
Developer</a><br><br>";
```

```
    loadlist($s);
?>
```

This is a very simple and not untypical admin script for a support or lookup table. Chances are, endusers will not be given access to the Developers table, so we don't need to be overly concerned with making the script bulletproof. We can now see some of the basic functions of a "List". First we get a tabular display of the rows in the table.

Most column headers allow you to click on them and change the sortation. This is handled by reloading the list with the sort parameter set. By default the table is sorted (in SQL order by) the table key (seq).

The list also provides a delete button to allow removal of any row. One important aspect of deletion in regards to MySQL is that MySQL does not have Declarative referential integrity or cascading deletes. What this means, is that Gamestatus.org *could* conceivably have a developer with many associated game entries. Although we've declared **DeveloperSeq** to be a foreign key for the Game table, unless you're using the InnoDB engine, MySQL doesn't do anything with this information to protect us from for example, orphaning a whole set of Game rows, by deleting their related Developer. In order to protect against this, or to offer the option of deleting the related Games, you will need to handle that with code. This tutorial doesn't include any code to check for rows in related table(s) prior to deletion. You might want to add them yourself as an exercise.

A **List** allows you to Add a new row or view/modify an existing member (row) of the list. The script should be fairly self explanatory... a query on developer is executed, and the result set is fetched:

PHP Example: (!)

```
while ($row1 = mysql_fetch_array($rslt1, MYSQL_ASSOC)) {
  //List table row built inside loop.  There will be a row in the table
for every row in the database.
}
```

For most columns, we make them links to the developerdetail.php script, with seq= set to be the **developerseq** for that row. I prefer to fetch values into an associative array, as it makes it very easy to refer to the value for a particular column. For example, $row1['Website'] is simple, self documenting (compared to $row1[2]), and avoids any potential side effects that might occur if the table structure is changed or added to in the future.

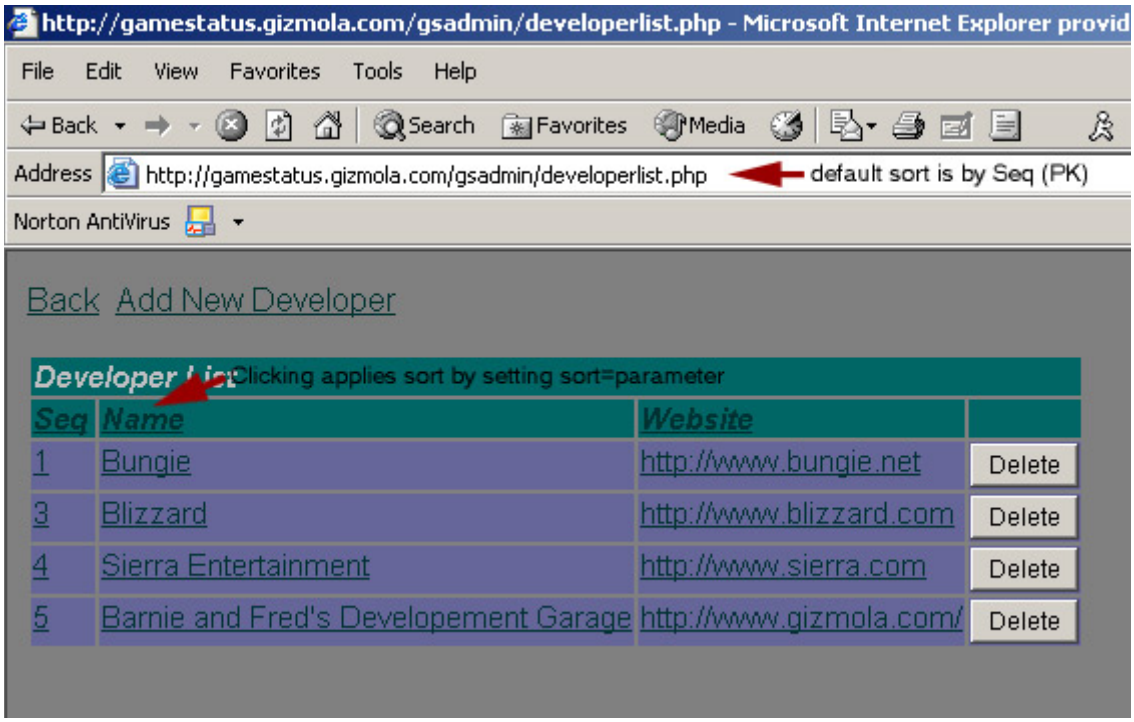
Handling CRUD

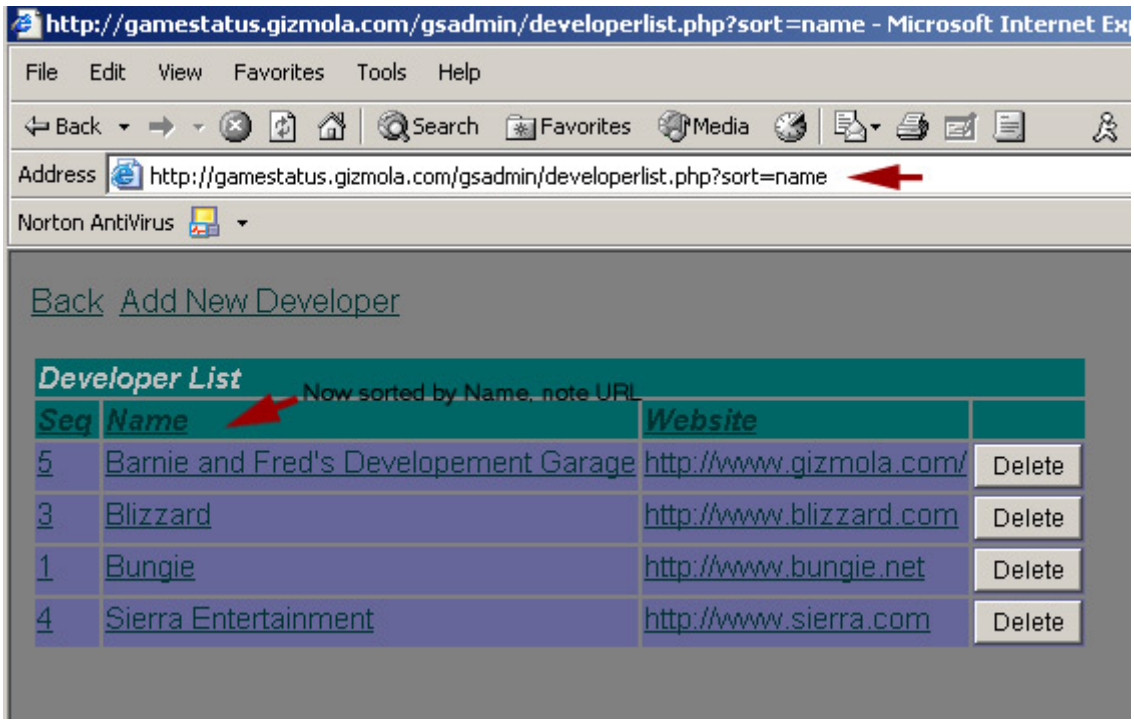
Probably the most complicated aspect of the script is the scheme for controlling Adds, updates and deletes, sometimes referred to as CReate,Update & Delete (CRUD) screens. By managing a set of url variables, we're able to insure that parameters are assembled correctly, and a level of context is preserved (for example, if a sort has been set the admin scripts make an attempt to retain that sort if and when the user returns to list.

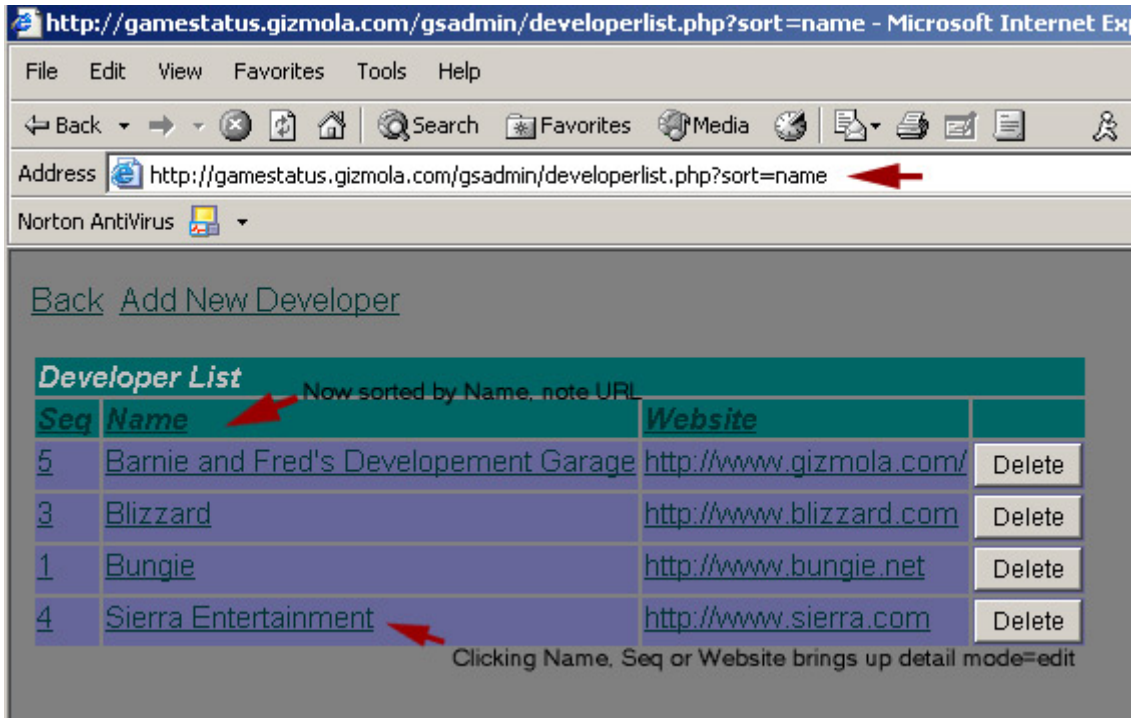
In the list script there is a $dtlUrl (Detail), $delUrl (delete) and $addUrl (Add/Insert). But how do these work? The first mechanism is use of a mode url parameter. Setting mode to "add", "edit", or "delete" helps the called scripts (Detail or Post) know what the intention of the user was. The seq parameter is set to the primary key relative to the corresponding row fetched from the database. This allows the detail and post scripts to know which row should be processed further.

Gamestatus.org takes one additional precaution of setting a **delseq** parameter, when a delete is specified, since this is a critical and potentially detrimental operation. Keeping a seperate **seq** and **delseq** is not really necessary, however. The script could easily be rewritten to eliminate the **delseq**, and instead use the **seq**. I used the delseq parameter as an extra precaution and to help illustrate the possible uses of url parameters, although the mode parameter is certainly sufficient control.

Developer LIST in action.







developer list row will drill down to developerdetail.php">

developerdetail.php

PHP Example: (!)

```php
<?php
$m = $HTTP_GET_VARS["mode"];
$sort = $HTTP_GET_VARS["sort"];
$seq = $HTTP_GET_VARS["seq"];
$s = $HTTP_GET_VARS["status"];

include("adminheader.php");
require("config.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");
    $q1 = "SELECT * FROM Developer where DeveloperSeq=$seq";
    $rslt1 = mysql_query($q1, $dbh1) or die("Query failed");
    $row1 = mysql_fetch_array($rslt1, MYSQL_ASSOC);
    /* Free resultset */
    mysql_free_result($rslt1);
} else {
    $seq = 0;
}
/* Closing connection */

$backurl = "developerlist.php";
$posturl = "developerpost.php";
$addurl = "developerdetail.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
Developer</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>Developer Detail</th></tr>";
echo "<tr><td>Seq:</td><td>$seq</td></tr>";
echo '<tr><td>Name:</td><td><input type="text" name="name" size="80"
maxlength="80"';
if ($m == 'edit') {
    echo " value='".htmlentities($row1['Name'],ENT_QUOTES)."'";
}
echo "></td></tr>";
echo '<tr><td>Website:</td><td><input type="text" name="website"
size="80" maxlength="255"';
if ($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>";
```

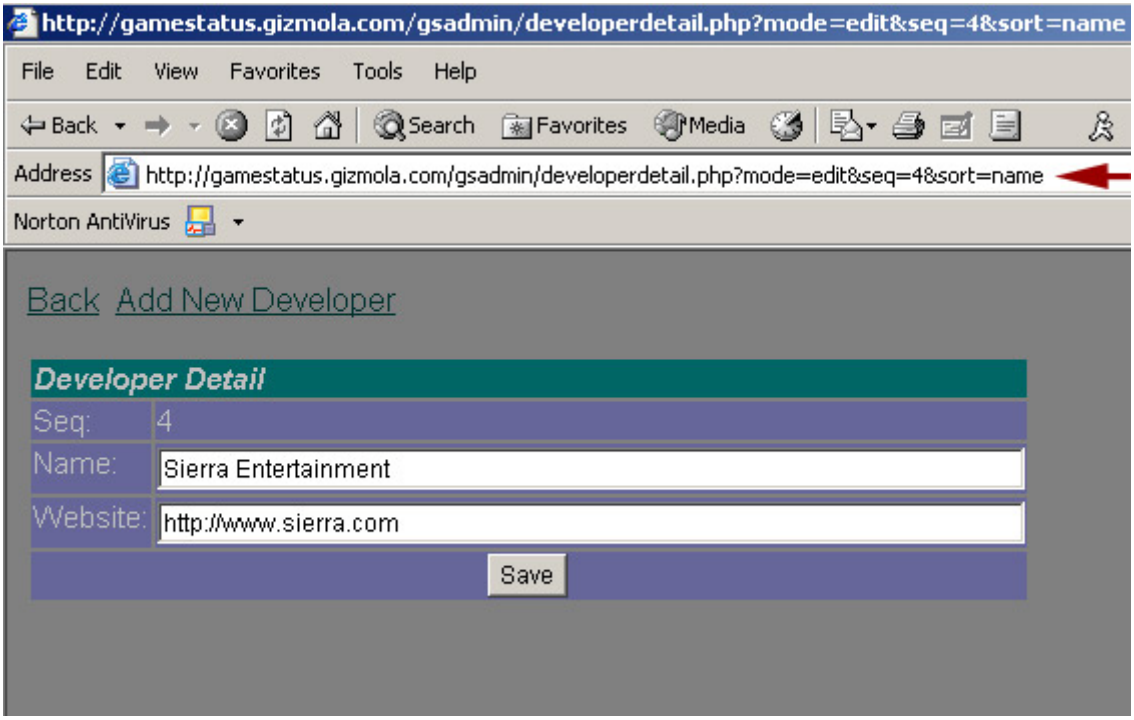
```
echo "</form>";
if (isset($s)) {
    echo "Edit Status: $s <br>";
}
if ($m == 'edit') {
    $addUrl = "developerdetail.php?mode=add&seq=$seq";
    if (isset($s))
        $addUrl .= "&sort=$s";
}
?>
```

The Detail form is organized in a simple manner, with modality driven by the **mode** parameter. The detail screen will be arrived at from one of two modes: add or edit. To keep things simple and uniform, a 2 column HTML table is built inside the form, with the typical prompt: value design. Depending on whether or not this is an add or an edit, we set the value= parameter for the corresponding object.

Notice, we have taken a shortcut. There is no "READONLY" or display only mode. Viewing the details for a row is seen as having the potential to become an edit. Remember that this is an admin system! It would be easy to add code that would bring all the values up without the form input objects, however omitting that code makes things a bit simpler and doesn't have any particular drawbacks in my experience.

We make one small MySql consideration and apply the htmlentities() function with the **ENT_QUOTES** constant flag. The main reason for this is to keep the script from breaking if the name of a Developer includes the ' character. Without it encoding that character, the SQL statement which inserts or updates developer information would break, since we delimit varchar values with the '' characters. This will become evident when looking at the post script.

One other point of interest is the inclusion of an "Add" link on the detail form. This facilitates data entry of a series of rows, without having to return to the list screen each time. With only a few rows, it wouldn't make much difference if the posting of a new Developer did return you to the list, but in the example of the games table, once a few hundred games were entered into the database, waiting for a requery of of the games table would make the process very slow. Thus we make it possible to add a new row from either the list table or the detail form. Aside from the mode, one of the indicators that a new row is in process, will be the setting of **seq** = 0.



Handling the POST

developerpost.php

PHP Example: (!)

```
<?php
    require("config.php");

    $sort = $HTTP_GET_VARS["sort"];
    $mode = $HTTP_GET_VARS["mode"];
    $delseq = $HTTP_GET_VARS["delseq"];

    $seq = $_POST['seq'];
    $name = $_POST['name'];
    $website = $_POST['website'];

/*
    // For debugging
    foreach($_POST as $key => $value) {
        echo "$key: $value <br>n";
    }
*/

    $dbh1 = mysql_connect($dbhost, $dbuser, $dbpasswd) or die("Could
```

```
not connect");
    mysql_select_db($dbname,$dbh1) or die("Could not select database");

    if (isset($mode) && ($mode == 'delete') && ($delseq > 0)) {
        $status = "Deleted";
        $q1 = "DELETE FROM Developer WHERE DeveloperSeq = $delseq";
        $rslt1 = mysql_query($q1, $dbh1) or die("Query failed");
    } elseif ($seq > 0) {
        $status = "Saved";
        $q1 = "UPDATE Developer SET Name='$name', Website='$website'
WHERE DeveloperSeq=$seq";
        $rslt1 = mysql_query($q1, $dbh1) or die("Query failed");

    } else {
        if ($seq == 0) {
            $status = "Added";
            $q1 = "INSERT INTO Developer (Name, Website) VALUES
('$name', '$website')";
            $rslt1 = mysql_query($q1, $dbh1) or die("Query
failed");
              $seq = mysql_insert_id($dbh1);
        }
    }
    if ($mode == 'delete') {
        $returnurl = "developerlist.php";
        if (isset($sort))
            $returnurl .= "?sort=$sort";
    } else {
           $returnurl =
"developerdetail.php?mode=edit&seq=".$seq."&status=".$status;
        if (isset($sort))
            $returnurl .= "&sort=$sort";
    }

    mysql_close($dbh1);
    header("Location: $returnurl");
?>
```

The Last of the List-Detail-Post scripts is the Post script. Post scripts don't display anything, but instead handle the insertion, updating, or deletion of a row. Since the post can be called either from a form or via a link, it needs to grab both $_GET and $_POST variables. In this script, I used $HTTP_GET_VARS, but $_GET would also work fine, and is the current best practice.

$_GET is one of the PHP Superglobal arrays. It works the same as $HTTP_GET_VARS, in that it contains the values of any URL parameters in an associative array, keyed by the parameter name. The only difference between $_GET and $HTTP_GET_VARS is that $_GET is "global" ie. visible inside of functions. For that reason alone, it's better to stick with $_GET.

Notice that there is no problem with making use of url parameters, even when the post script is being called by POST from a form. This allows us to keep track of the list *sort=*

parameter, even though our main goal is to update the database with edits from the detail form.

Once again the most complicated code is the code that handles the url's. One way of improving on this handling would be to employ a set of routines that would allow you to push and pop url parameters from a stack. Even without a stack, having URL variables, and some simple logic blocks does an acceptable job of preserving applicable parameters in most cases.

Once the post script has determined what to do, it sets a status parameter, which can be picked up by the script it's returned to, in order to display some information about what the post accomplished. When the post script has done it's job, it returns to the calling script the header("Location: $returnurl") function. If we deleted a row, we return to the list. If it's an addition or an edit, we return to the detail, using the seq= parameter.

PHP Example: (!)

```
if ($seq == 0) {
    $status = "Added";
    $q1 = "INSERT INTO Developer (Name, Website) VALUES ('$name',
'$website')";
    $rslt1 = mysql_query($q1, $dbh1) or die("Query
failed");
    $seq = mysql_insert_id($dbh1);
}
```

The main indicator that a new row is required, is that **$seq** was set to 0. Note that like most of the tables in the Gamestatus system, MySQL takes care of a lot of important details. If you refer back to the schema you'll note that the Developer table is keyed by DeveloperSeq which is declared to be an **AUTO_INCREMENT** column. Thanks to the **AUTO_INCREMENT**, we do not need to set the value of DeveloperSeq. We simply insert the row, and MySQL provides a unique sequential value for DeveloperSeq. Our only problem is finding out, what the key generated by the MySQL **AUTO_INCREMENT** was. Once the insert is complete we use the mysql_insert_id() function, which handles this problem for us. Since it's run immediately following the insert, on the same database handle, MySQL is able to return the value we just inserted. We set **$seq** to this value, and return to the detail form, which promptly looks up our new row. Nice eh?

Summary

To summarize, we've now covered the basic architecture of a LAMP system, talked about normalized database design, and implemented a **List-Detail-Post** application to administer one of the tables for the gamestatus.org website. In PART II, we'll continue the developement of the gamestatus.org administration system, and provide a user

interface solution for handling a many to many relationship. In the meantime, I encourage you to use Developer as an example, and develope list-detail-post scripts for Genre, Publisher, Platform and Status. I think you'll find that cloning a set of scripts to handle any of these tables, will not take you very long at all, once you understand how the Developer list-detail-post scripts work.

If you feel you're ready for more, move on to Part II.