

## APPENDIX D: USING AMFPHP WITH GAMES

One of the most common features of Flash games on the Web these days is a persistent leaderboard. Competition and the desire to publicly be declared the best at something can drive people to play games, even long after they've lost interest in the game itself. Unfortunately, because maintaining even a simple leaderboard requires setting up back-end services (or paying for a third-party service), the barrier to entry for the technology turns many developers away. As such, in this appendix, we'll look at how relatively little code has to be written to create an effective leaderboard system from scratch. I should warn that this isn't for the faint of heart; nothing beats knowing someone who does this kind of work for a living to set this up for you.

For this example, I'll be building the back-end in PHP using the AMFPHP framework ([amfphp.org](http://amfphp.org)). There are, probably, a dozen different platforms you could use for the server-side portion of a leaderboard, but I've chosen PHP for a few reasons.

- PHP is open-source, as is the AMFPHP framework; pretty much anyone who has the most basic Web-hosting plan is bound to have them installed. If you don't have even a Web host, you can download and install it on your local machine for development.
- The AMF (Action Message Format) protocol is very fast and efficient for communicating with Flash, and the AMFPHP library is extremely easy to install and get going for free. It's not considered an enterprise-level solution, but if you work at an enterprise-level company, you're not likely to need the contents of this appendix because someone other than you does that task.
- The code is pretty succinct for what we need to accomplish, and you should be able to take the source provided along with this chapter and get it running on your server with very few issues.
- MySQL, the free database with which PHP is most commonly installed, is very easily accessed and scripted through AMF.

### Setting up AMFPHP

For the purposes of not getting bogged down in details because everyone's server configurations can be different, I'm going to assume that, at this point, you have a working installation of PHP and can view Web pages either on your local machine or a Web server. You will also need an installation of the free MySQL database for storing the scores. If you're running on a commercial Web

server with PHP, chances are *very* good that it is already installed, along with the admin panel known as phpMyAdmin. If you’ve only got your local machine to use as a server, check into WAMP on Windows or MAMP on Mac. These are consolidated installs that contain the Apache Web server, PHP, and MySQL. I’m also going to assume that you have a basic working knowledge of database terminology, such as fields and tables. If you’re starting completely from scratch, Wikipedia is a good place to brush up on the basics. In the coming pages, I’ll illustrate how to set up the basic tables your database will need to save and retrieve scores.

Download the latest version of AMFPHP from [amfphp.org](http://amfphp.org). The latest version as of this writing is 1.9. When you extract the archive, you should have a few different folders and files. This is the core AMFPHP installation. Simply take the containing folder and copy it to your Web server—I like to name the folder “services” as shown in Fig. D.1.

The two features inside this package that are of interest to us are `gateway.php` and the `services` subfolder. The gateway file is the “conduit” through which we will connect to and access all the services from Flash. It is also often referred to as the AMF “endpoint.” When you establish a connection in Flash, which I discussed in Chapter 15, it is to this file. From then on, all requests are directed through the gateway. In the `services` subfolder is where we will place the PHP code we need to do our custom behaviors. Referring back to Fig. D.1, you’ll see that I have created a folder called “games,” inside of which is a file named `HighScores.php`. Going forward, I will simply refer to this as the `HighScores` service. Before we dig into this file, however, we need to set up our database. Don’t worry though; I’ve included an exported SQL database in the files for this appendix that you can import to get started faster.

Filename	Size
discussions	4.0 KB
files	4.0 KB
index.php	397 B
license.txt	15.0 KB
readme.html	8.9 KB
services	4.0 KB
browser	4.0 KB
core	4.0 KB
gateway.php	5.9 KB
globals.php	815 B
json.php	218 B
services	4.0 KB
amfphp	4.0 KB
games	4.0 KB
HighScores.php	1.8 KB
xmlrpc.php	217 B

**Figure D.1** A view of the AMFPHP folder as installed on my Web server.

The following section is just to walk through the structure, so that it makes contextual sense.

## Setting up the Database Tables

Using the Control panel that comes with your Web-hosting account (or the tools that are part of your local installation), create a database called HighScores. Once you have the database and can access it, launch phpMyAdmin or the database administration tool of your choice. We're going to create two tables inside the database to start—one will store the names and passwords of every game that will have its own leaderboard, and the other will be the table for the high scores of our first game. The first table we'll call GameIDs and all subsequent tables we'll name with the convention "Scores\_<game name>". Figure D.2 shows the list of these tables in phpMyAdmin, showing the table created for Marble Runner.

When you create the GameIDs table, it should include a field for the ids (names) and secretkeys (passwords), as shown in Fig. D.3. Although there are no hard and fast rules for creating fields, I set both of these as varchar (variable number of characters); the game name can be up to 20 characters, and the password can be 10.

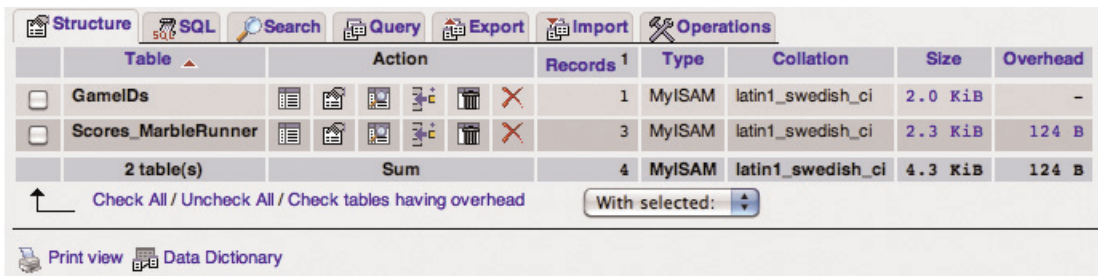
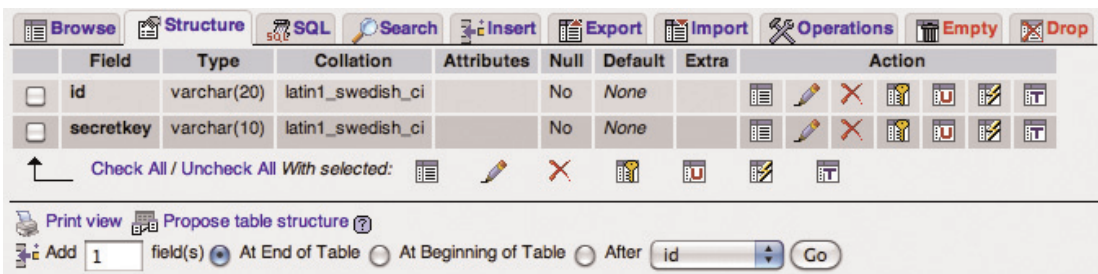


Table	Action	Records	Type	Collation	Size	Overhead
GameIDs	[Icons]	1	MyISAM	latin1_swedish_ci	2.0 KiB	-
Scores_MarbleRunner	[Icons]	3	MyISAM	latin1_swedish_ci	2.3 KiB	124 B
2 table(s)	Sum	4	MyISAM	latin1_swedish_ci	4.3 KiB	124 B

**Figure D.2** The two initial tables that we'll need for our service: GameIDs and Scores\_MarbleRunner.



Field	Type	Collation	Attributes	Null	Default	Extra	Action
id	varchar(20)	latin1_swedish_ci		No	None		[Icons]
secretkey	varchar(10)	latin1_swedish_ci		No	None		[Icons]

**Figure D.3** The fields required for the GameIDs table.

For any of the score tables, we'll need three fields: the score (I make it an integer, which will give plenty of room for large numbers), the initials of the player (a three-character string), and a timestamp (up to 100 characters). The score and initials will be displayed on a leaderboard, and the timestamp will be logged to ensure that a player is not able to register the same score multiple times and fill up the leaderboard. A score submission with the same initials, timestamp, and score as one already in the table will be ignored. These fields can be seen in Figure D.4.

The final step in setting up our databases will be to add the first game entry to the GameIDs table. Figure D.5 shows that I've created one for MarbleRunner with the secret key of "fandango." This is the password that will be used inside of Flash to encrypt and protect the data as it is sent to the server. Using this structure, all you'll have to do in the future to add leaderboards for other games will be to add an entry in the GameIDs table and create a new Scores\_<game name> table with the same fields as MarbleRunner. The whole process should not require more than a minute or two.

Field	Type	Collation	Attributes	Null	Default	Extra	Action
<input type="checkbox"/> score	bigint(20)			No	None		[Icons]
<input type="checkbox"/> initials	varchar(3)	latin1_swedish_ci		No	None		[Icons]
<input type="checkbox"/> timestamp	varchar(100)	latin1_swedish_ci		No	None		[Icons]

Check All / Uncheck All With selected: [Icons]

Print view Propose table structure

Add 1 field(s) At End of Table At Beginning of Table After score Go

Figure D.4 The fields required for the score tables.

Show: 30 row(s) starting from record # 0

in horizontal mode and repeat headers after 100 cells

+ Options

id	secretkey
<input type="checkbox"/> MarbleRunner	fandango

Check All / Uncheck All With selected: [Icons]

Show: 30 row(s) starting from record # 0

in horizontal mode and repeat headers after 100 cells

Query results operations

Print view Print view (with full texts) Export CREATE VIEW

Figure D.5 The MarbleRunner game ID created in the table.

## The HighScores Service

Now that we have the tables we need in the database, we can create the service that will take data from the game and store it in the tables. For this, we will look at the HighScores.php file. Even if you're not familiar with PHP syntax, you should be able to follow along—it's not *too* different from ActionScript.

```
<?php

class HighScores
{

    public function __construct() {
        $username = "<username here>";
        $password = "<password here>";
        $db = "<your server name>_HighScores";
        mysql_connect("localhost", $username, $password);
        mysql_select_db($db);
    }
}
```

In the constructor function (called “\_\_construct” in PHP), we connect to the MySQL database using the username and password you associated with it when you created it (either through your Web host or your local machine). This will be automatically called when this service is accessed, so the database will be available for any method calls that Flash makes. When you put this file on your server, you will need to input your custom values; obviously, my credentials won't work for you. Now, we'll look at the method we call from Flash to save scores.

```
public function saveScore($gameID, $score, $initials,
    $dateStamp, $hash)
{
    $sql = "SELECT `secretkey` FROM `GameIDs` WHERE `id` = \"".
        $gameID."\"";
    $res = mysql_query($sql);
    if (mysql_numrows($res) == false)
    {
        return "ERROR - GAME ID DOES NOT EXIST";
    }
    $secretKey = mysql_fetch_object($res)->secretkey;
    $compareHash = $secretKey.$score.$initials.$dateStamp;
    $compareHash = md5(md5($compareHash));
    if ($compareHash != $hash)
    {
        return "ERROR - HASH DOES NOT MATCH";
    }
}
```

```
$sql = "SELECT * FROM `Scores_`.$gameID.`" WHERE `score`=".
    $score." AND `datestamp`=". $dateStamp." AND `initials`=
    \"\".$initials.\"\"";
$res = mysql_query($sql);
if (mysql_numrows($res) > 0) //RECORD ALREADY EXISTS
{
    return "ERROR - SCORE ALREADY EXISTS";
}
$sql = "INSERT INTO `Scores_`.$gameID.`" (`score`,`initials`,
    `datestamp`) VALUES (\".$score.\", \"\".$initials.\"\", \"\".
    $dateStamp.\"\"");
$res = mysql_query($sql);
return $res;
}
```

Obviously, there's a lot going on in this method, so we'll break it down into individual pieces. We'll start with the parameters that are passed to the method.

```
saveScore($gameID, $score, $initials, $dateStamp, $hash)
```

The required arguments are a game ID (that matches a value in our GameIDs table), the player's score, the player's initials, a datestamp created for this submission, and a hash of these values for security. For more information on creating hashes and securing communication, refer to the bonus chapter "On Your Guard," which can be found at [www.flashgamebook.com](http://www.flashgamebook.com).

```
$sql = "SELECT `secretkey` FROM `GameIDs` WHERE `id` = \"\".
    $gameID.\"\"";
$res = mysql_query($sql);
if (mysql_numrows($res) == false)
{
    return "ERROR - GAME ID DOES NOT EXIST";
}
```

The first few lines do a look-up in the GameIDs table using the id parameter. If it can't retrieve a secret key, it is assumed that game has not been created in the database, so we return an error message to Flash explaining what has happened.

```
$secretKey = mysql_fetch_object($res)->secretkey;
$compareHash = $secretKey.$score.$initials.$dateStamp;
$compareHash = md5(md5($compareHash));
if ($compareHash != $hash)
{
    return "ERROR - HASH DOES NOT MATCH";
}
```

The next step is to verify that the hash passed as a parameter matches a hash made with the secret key for this game. In this case, we concatenate the secret key, the score, the initials, and the datestamp together as one string. This string is then hashed twice using MD5. If the resultant value does not match what was passed by Flash, a different error message is returned.

```
$sql = "SELECT * FROM `Scores_`.$gameID."` WHERE `score`=" .
    $score." AND `datestamp`=" . $dateStamp." AND `initials`="
    "\".$initials."\"";
$res = mysql_query($sql);
if (mysql_numrows($res) > 0)
{
    return "ERROR - SCORE ALREADY EXISTS";
}
```

The third verification step is to make sure that this score submission is not a duplicate. Basically, we scan the database for any entry where all three parameters match. If one is found, an error message is returned to Flash.

```
$sql = "INSERT INTO `Scores_`.$gameID."` (`score`,`initials`,`
    `datestamp`) VALUES (".$score.", "\".$initials."\", "\".
    $dateStamp."\"");
$res = mysql_query($sql);
return $res;
```

At this point, if the method has not ended because of an error, the score, initials, and datestamp are all saved as an entry in the database. Now, let's look at the `getScores` method.

```
public function getScores($gameID, $number)
{
    $sql = "SELECT * FROM `GameIDs` WHERE `id` = "\".
        $gameID."\"";
    $res = mysql_query($sql);
    if (mysql_numrows($res) == false) //GAME ID DOES NOT EXIST
    {
        return false;
    }
    $sql = "SELECT * FROM `Scores_`.$gameID."` ORDER BY `score`
        DESC LIMIT ".$number;
    $res = mysql_query($sql);
    $scores = array();
    for ($i = 0; $i < mysql_num_rows($res); $i++)
    {
        array_push($scores, mysql_fetch_assoc($res));
    }
    return $scores;
}
```

This method is much simpler because it has to do almost no validation. Flash simply has to pass it a game ID and the number of results it wants to display. I'll still break it down into its two main parts.

```
$sql = "SELECT * FROM `GameIDs` WHERE `id` = \"\".$gameID.\"\"";
$res = mysql_query($sql);
if (mysql_numrows($res) == false) //GAME ID DOES NOT EXIST
{
    return false;
}
```

Much like the `saveScore` method, we first have to test to make sure the game ID exists for this game. If it does not, we simply return false rather than an error message.

```
$sql = "SELECT * FROM `Scores_\".$gameID.\"` ORDER BY `score` DESC
LIMIT \".$number;
$res = mysql_query($sql);
$scores = array();
for ($i = 0; $i < mysql_num_rows($res); $i++)
{
    array_push($scores, mysql_fetch_assoc($res));
}
return $scores;
```

The next few lines extract the number of scores requested from the table. If there are fewer scores than requested, the smaller number will be returned. Note that they are selected from the database by score in the descending order. This will make the largest number at the top and subsequent numbers will get progressively smaller. If you had a game where a lower score was better (like golf), you'd want to create an alternate version of the `getScores` method to order them ascending. Each of these results is then added to an array and returned to Flash. To see how this data is then extracted and parsed, refer to Chapter 15 in the book.

That's pretty much all there is to it. This should hopefully provide a solid model for creating other services. The same principles would apply if you wanted to store more robust profile data for a user. Once you start persisting user data, the possibilities are truly endless. For other tutorials using AMFPHP to store data, I highly recommend Lee Brimelow's series on <http://gotoAndLearn.com>.