

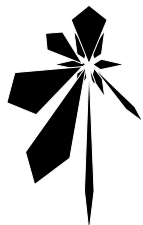
Introduction to PHP Programming for Online Game Development

Revision 1.1

June 18, 2010

by

Aloysius Indrayanto



(C) 2010 AnemoneSoft.com

This document is multi-licensed under the Creative Commons Attribution Share-Alike (CC-BY-SA) license version 3.0 and the GNU Free Documentation License (GNU FDL) version 1.3 or later.

1. Introduction

An online game is a game that is played by multiple players across a computer network. For a public online game, the network is usually the internet. In this configuration, there will be a public server (or a cluster of servers) to where players can connect to and play the game.

An online game may require the user (player) to install a client application to be able to play the game. However, there are online games that can be played without installing any additional client application. This second type of games are usually played using web browser. Some games may still require the player to install a browser plugin such as Flash player, while some other games may require none.

This tutorial presents some basic concepts and techniques that would be needed to develop an online game application using PHP. The topics that will be discussed in this tutorials include: structure of the application/game, database stack, session management and security, and captcha image generator. The presented basic concepts and techniques would be mostly suitable to create browser-based online games that requires only JavaScript (no additional plugin). However, it may be possible to adapt the concepts and techniques for other types of online games. Please note that this tutorial will not create a fully functional online game.

2. Requirements

A basic knowledge in programming using PHP and MySQL will be needed to understand the topic discussed in this tutorial. A system with an installation of Apache Web Server (HTTPD) version 2.2.x, PHP: Hypertext Preprocessor version 5.x, MySQL Community Server version 5.x, and a recent enough browser will be needed to run the code snippets.

All the test database and tables used in the code snippets are unreal. The code snippets assume that those database and tables just exist in the system. However, if necessary, it would be easy to actually create those database and tables after understanding how they are accessed/used.

3. File and Directory Structure of an Online Game

There are two groups of PHP script files (web pages): the client-accessible pages and non-client-accessible pages. It is necessary to store those files/pages in a proper directory tree for clarity. It would be also necessary to inform the web browser, proxy, etc. that these pages are dynamic and should not be cached by setting the appropriate HTTP header using:

```
header('Cache-Control: no-store, no-cache, must-revalidate');  
header('Pragma: no-cache');
```

3.1. Client-Accessible (Visible) Files/Pages

This group contains files/pages that are visible and accessible by clients' browser. Some files/pages that are part of this group are:

- **The login page.** This is the starting page for users when they want to log in to the game and start playing. Basically, if there is no special needs, the index page ('index.php') should be redirected to this login page. Of course, besides the login form, this page can contain some other information/contents such as, the background story of the game, advertisements, links, etc.
- **The registration page.** This is the page where a new user can register for the game. Basically, the login page must link to this registration page.
- **The 'forgot password' page.** Sometime, users may forget their passwords. Hence, it is necessary to have a page from where users can recover their passwords. Basically, the login page must also link to this 'forgot password' page.
- **The help page.** It is necessary to write a simple page about how the game can be played. Due to new users may want to know about the gameplay before actually registering, having a link from the login page to this help page would be a very good idea.
- **The main game page.** This is the landing page after a user submit the login form with the correct user name and password. While it is possible to have multiple pages for the actual gameplay, it would be a good idea to have a kind of controller page from where the other gameplay-related (content) pages will be included. This approach is a kind of Model View Controller (MVC) software architecture.
- **The main administrator page.** This is the main page for game administration. This page will act as a kind of controller page for game administrator to manage or modify the state/content of the game. It is necessary to ensure that only users with administrative privileges can access this page.
- **AJAX pages.** Basically, these pages are responsible for processing AJAX requests when the player perform actions. When requested, this pages may not return a completely valid HTML code. Instead, they may return JSON (JavaScript Object Notation) data, partial HTML tags/data, image stream, etc.

- **Informational pages.** Sometimes, it is necessary to inform a user if the game is currently under maintenance, or, maybe if the user's account is being blocked for some reason. While it is possible to include or display such information from the main game page, it can be better to have separated and simpler pages for these kind of information. Also it would be better to separate pages for optional game status, buying payed items/services, etc. from the main game page.

3.2. Non-Client-Accessible (Invisible) Files/Pages

This group contains files/pages that are invisible and not directly accessible by clients' browser. These files/pages are meant to be included from other files/pages. It is important to ensure that these files/pages cannot be viewed by clients. This can be achieved by using Apache Web Server access control or, easier, by using a one line PHP statement at the beginning of the script file, such as:

```
defined('ACCESS_OK') or die;
```

Hence, all the client-accessible pages should define the 'ACCESS_OK' constant before attempting to include any of the non-client-accessible pages. Some files/pages that are part of this group are:

- **Configuration files.** It is a good idea to store the game, database, PayPal (if you use its service), etc. configuration data in their own separated file.
- **Text definition files.** It is a good idea to separate the actual in-game texts from the game itself. With this approach, it will be much easier to translate the game to other language. Basically, a text definition file will just contains constants or global variables such as:

```
define('TEXT_GC_LEVEL',      'Level'      );  
define('TEXT_GC_SKILL_POINT', 'Skill Point');  
define('TEXT_GC_CREDIT_POINT', 'Credit Point');
```

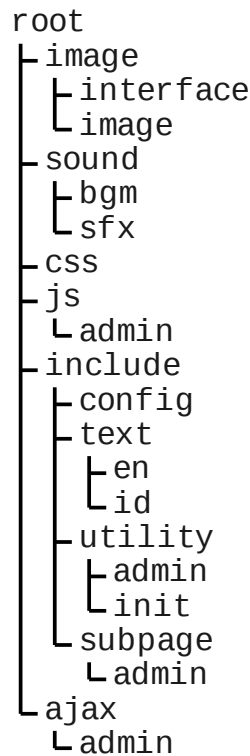
or:

```
$GLOBALS['text_gcLevel']      = 'Level';  
$GLOBALS['text_gcSkillPoint'] = 'Skill Point';  
$GLOBALS['text_gcCreditPoint'] = 'Credit Point';
```

- **Utility files.** Basically these files only contains utility classes and functions. These files should not generate any direct HTML output, except maybe, error messages.
- **Game sub pages.** These pages are content pages. These pages are meant to be included dynamically from within the main game page.
- **Administrator sub pages.** These pages are content pages. These pages are meant to be included dynamically from within the main administrator page.

3.3. Directory Structure

An example of a possible directory structure is shown in the picture below. The root directory is the root directory of your web server (in Linux, it is usually /var/www/html).



Explanation:

- The image directory is basically where all the images related to the application will be stored.
 - The interface sub-directory is for storing images related to the user interface such as buttons, icons, status indicators, etc. It is possible to have more sub-directories under this sub-directory.
 - The game sub-directory is for storing images related to the actual game such as animated character images, weapons, items, etc. It is possible to have more sub-directories under this sub-directory.
- The sound directory is for storing all audio files related to the game. It is a good idea to separate audio files for background music and sound effect (such as walking/running sound, laser beam, explosion, etc.) to their own sub-directories. It is possible to have more sub-directories under those sub-directories for more clarity.

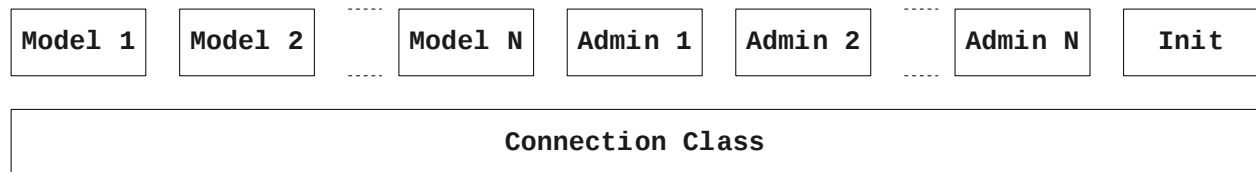
- The `css` directory is where all CSS (Cascading Style Sheet) files are stored. It is possible to have multiple CSS for one game, for example: if the game supports multiple themes for its user interface, each theme can have its own CSS file.
- The `js` directory is for storing all JavaScript files. It is possible to have more sub-directories under this sub-directory to group the script files based on the classes/functions defined in the script files. However, at least, separating the administration-related scripts from the ordinary game script would be a good idea.
- The `include` directory is the place where all non-client-accessible PHP script files should be put.
 - The `config` directory is for storing all configuration files.
 - The `text` directory is where all text definition files are stored. If the game/application support multiple language that can be chosen when playing, then put each language's text definition files in its own sub-directory.
 - The `utility` directory is for storing all utility files. Basically, it would be a good idea to separate ordinary game-related, administration-related, and initialization-related utility files in their own sub-directories. Note that, initialization-related utility files is most likely only need to be run once (when the game is first initialized). Therefore, do not forget to delete this directory in a production server after initialization so that they will not cause security problem.
 - The `subpage` directory is where all sub pages are stored. It is possible to have more sub-directories under this sub-directory to group the pages based on its functions/area. However, at least, separating the administrator sub pages from the ordinary game sub pages would be a good idea.
- The `ajax` directory is where all AJAX responder pages are stored. It is possible to have more sub-directories under this sub-directory to group the pages based on its functions. However, at least, separating the administrator script from the ordinary game script would be a good idea.

4. Database Stack

In this tutorial, we will use MySQL for the database server. However, in a real application, any supported database server can be used. It is even possible to switch the database server to other vendor at some point in the application lifetime. Therefore, it is important to

isolate the database access commands in their own files so that only those files that need to be changed in case the developer decided to switch database vendor.

Isolation, in this regard, is obtained by using database stack. Basically, it is just a collection of classes that encapsulate low-level data base connection and SQL commands. The picture below shows a possible database stack.



The `Connection` class is the lowest-level class of the stack. Basically it encapsulates all the commands to connect to a database server, execute SQL, locking/unlocking tables, performing transaction, etc. The classes on top of this class (`Model`, `Admin`, and `Init` classes) will use the public API exported by this `Connection` class to read and write the database.

The `Model` classes encapsulate models. A model in a game can refer to a playable character, a non playable character, party's inventory, etc. Basically the `Model` classes encapsulate all the SQL to get the status, update the status, modify the content, etc. of those models. The idea is that these classes will export a consistent public API and hide any possible SQL incompatibilities at the back.

The `Admin` classes also has similar purposes, to export a consistent public API and hide any possible SQL incompatibilities at the back. However, unlike the `Model` classes, the `Admin` classes do not necessarily models something. Basically, a game administrator may need to access multiple models directly to accomplish his/her task. In this case, the developer may just use the individual `Model` classes directly, or create a combined `Admin` class to optimize the particular process. Also, sometime, an administrator needs to modify global game status (for example the list of winners for the last month competition), in this case the `Admin` class created for the purpose does not even encapsulate a model.

The `Init` classes is similar to the `Admin` classes. The idea is also the same, to export a consistent public API and hide any possible SQL incompatibilities at the back. The difference is that these classes will only need to be used once for the game initialization. Hence, do not forget to delete them from a production server so that they will not cause security problem (imagine an attacker use these classes to reset all the game data!).

4.1. An Example Connection Class

The code snippet below shows an example of Connection class (named DBConn) that is developed using the PHP's MySQL extension.

```
<?php
/*
   Copyright (C) 2010 AnemoneSoft.com
   Aloysius Indrayanto

   This program is free software and comes with ABSOLUTELY NO WARRANTY.

   This program is licensed under the GNU Lesser General Public License (GNU LGPL)
   either version 3 of the license, or (at your option) any later version as
   published by the Free Software Foundation.
*/

// Access information (in a real application, these should be put in
// a separated configuration file)
define('DB_HOST',      'localhost' );
define('DB_USER',     'mytestuser');
define('DB_PASSWORD', 'mypassword');
define('DB_DATABASE', 'MyTestDB' );

// Database-connection utility class
class DBConn {
    //
    // Private section
    //
    private $_dbh;           // MySQL connection handle
    private $_result;       // Result resource handle
    private $_anyTableLocked; // Flag to indicate if any table is locked
    private $_inTranscation; // Flag to indicate if currently a transaction is in progress

    private function _closeConnection()
    {
        // If there is no active connection, just return
        if(!$this->_dbh) return;

        // If there is an active result, free it
        if($this->_result) {
            @mysql_free_result($this->_result);
            $this->_result = null;
        }

        // Unlock tables and rollback transaction
        $this->unlockTables();
        $this->rollbackTransaction();

        // Close connection
        @mysql_close($this->_dbh);
        $this->_dbh = null;
    }

    private function _exitError($message)
    {
        // Close connection
        $this->_closeConnection();

        // Exit with HTTP error
        header('HTTP/1.1 500 Internal Server Error');
        echo '<b>HTTP/1.1 500 Internal Server Error</b><br/>';
        echo '<br/>[DBConn]<br/>' . $message . '<br/>';
        exit;
    }

    //
    // Public section
    //
}
```

```

public function __construct()
{
    // Set variables
    $this->_dbh          = null;
    $this->_result       = null;
    $this->_anyTableLocked = false;
    $this->_inTranscation = false;

    // Connect to the database
    $attempt = 0;
    while(true) {
        // Try to connect to the database
        $this->_dbh = @mysql_connect(DB_HOST, DB_USER, DB_PASSWORD);
        // Succeeded
        if($this->_dbh)
            break;
        // Failed
        else {
            // Increment counter
            ++$attempt;
            // Retry a few times for: error 1040 - "Too many connections"
            //                                     error 1203 - "User %s already has more than
            //                                     'max_user_connections' active
            //                                     connections"
            if($attempt < 10 && (mysql_errno() == 1040 || mysql_errno() == 1203))
                usleep(500000);
            // Report error
            else
                self::_exitError('Could not connect to the database server: '
                    . mysql_error());
        }
    }

    // Select database
    if(!mysql_select_db(DB_DATABASE, $this->_dbh)) {
        $msg = mysql_error($this->_dbh);
        mysql_close($this->_dbh);
        self::_exitError('Could not select the game database: ' . $msg);
    }

    // Set timezone to UTC
    $this->sqlExec('SET time_zone = \'+0:00\');
}

public function __destruct()
{ $this->_closeConnection(); }

public function sqlEscapeString($str)
{ return mysql_real_escape_string($str, $this->_dbh); }

public function sqlExec($sql)
{
    // Free previous result (if any)
    if($this->_result) {
        @mysql_free_result($this->_result);
        $this->_result = null;
    }

    // Execute query and check for error
    $this->_result = mysql_query($sql, $this->_dbh);
    if(!$this->_result) {
        $msg = '<em>SQL execution failed:</em><br/>';
        $msg .= mysql_error($this->_dbh);
        $msg .= '<br/><em>Offending statement:</em><br/>';
        $msg .= $sql;
        mysql_close($this->_dbh);
        self::_exitError($msg);
    }

    // If the query does not produce result, set the result to NULL
    if(gettype($this->_result) != 'resource') $this->_result = null;
}

```

```
public function sqlNumberOfColumns()
{ return $this->_result ? mysql_num_fields($this->_result) : 0; }

public function sqlNumberOfRows()
{ return $this->_result ? mysql_num_rows($this->_result) : 0; }

public function sqlNumberOfAffectedRows()
{ return mysql_affected_rows($this->_dbh); }

public function sqlGetLastInsertedID()
{ return mysql_insert_id($this->_dbh); }

public function sqlFetchColumnNames()
{
    $colNames = array();

    $idx = 0;
    while($idx < mysql_num_fields($this->_result)) {
        $meta = mysql_fetch_field($this->_result, $idx);
        $colNames[$idx] = $meta->name;
        ++$idx;
    }

    return $colNames;
}

public function sqlFetchRow()
{ return mysql_fetch_row($this->_result); }

public function lockReadTable($table)
{
    if($this->_inTransaction)
        self::_exitError('A transaction is currently in progress');

    $this->sqlExec('LOCK TABLES ' . $table . ' READ');
    $this->_anyTableLocked = true;
}

public function lockWriteTable($table)
{
    if($this->_inTransaction)
        self::_exitError('A transaction is currently in progress');

    $this->sqlExec('LOCK TABLES ' . $table . ' WRITE');
    $this->_anyTableLocked = true;
}

public function unlockTables()
{
    if(!$this->_anyTableLocked) return;

    $this->sqlExec('UNLOCK TABLES');
    $this->_anyTableLocked = false;
}

public function beginTransaction()
{
    if($this->_anyTableLocked)
        self::_exitError(
            'Cannot start a transaction: one or more tables are currently locked');
    if($this->_inTransaction)
        self::_exitError('A transaction is already in progress');

    $this->sqlExec('START TRANSACTION');
    $this->_inTransaction = true;
}

public function commitTransaction()
{
    if(!$this->_inTransaction) return;
}
```

```
        $this->sqlExec('COMMIT');
        $this->_inTransaction = false;
    }

    public function rollbackTransaction()
    {
        if(!$this->_inTransaction) return;

        $this->sqlExec('ROLLBACK');
        $this->_inTransaction = false;
    }
}
?>
```

dbconn.php: An Example Connection Class using the PHP's MySQL Extension.

Explanation:

- On creation, the class' constructor (the `_construct` method) will directly try to connect to the MySQL server specified in the 'access information'. If the MySQL server reported that there has been too many connections (error 1040) or the specified database user has too many active connections (error 1203), the script will sleep for 500 mS and try again. After 10 failed attempts, the script will report error. For any other error, the script will directly report error. Upon a successful connection, the script will select the specified database and set the connection's timezone to UTC (this would prevent date/time conversion to be done automatically by the MySQL server).
- The class' destructor will just call the `_closeConnection()` method. The method basically free any query result (if any), unlock tables, rollback transaction, and close the MySQL connection.
- If there is any error, the private method `_exitError()` will be used to report errors. The method will close any active MySQL connection before sending 'HTTP/1.1 500 Internal Server Error'.
- The `sqlExec()` method will free any previous query result (if any) before executing the new query and checking for error. If the query does not produce result (the `mysql_query()` function does not return a resource), the `sqlExec()` method will set the internal result variable to `null`.
- The `sqlFetchColumnNames()` method will basically fetch the names of all the columns produced by a query and return them as an array.
- Due to locking tables cannot be done while any transaction is in progress, the `lockReadTable()` and `lockwriteTable()` methods check if a transaction is in progress before actually tries to lock the specified tables. On the other hand, starting a transaction

will automatically unlocks any locked table. Therefore, the `beginTransaction()` method will check if any table is locked before actually tries to start a transaction. Note that these checking are done so that we can write a clean code using the `DBConn` class by not mixing transaction and table locks (MySQL server can actually resolve most conflict automatically).

- The other methods are just wrappers to the PHP's MySQL extension API.

4.2. An Example Model Class

The code snippet below shows an example of `Model` class that uses the previously presented `DBConn` class. To understand the class, assume that this class models a player in a game world. Each player has statuses such as health point, magic point, and spirit point.

```
<?php
/*
    Copyright (C) 2010 AnemoneSoft.com
        Aloysius Indrayanto

    This program is free software and comes with ABSOLUTELY NO WARRANTY.

    This program is licensed under the GNU Lesser General Public License (GNU LGPL)
    either version 3 of the license, or (at your option) any later version as
    published by the Free Software Foundation.
*/

// Include the database connection class
include_once 'dbconn.php';

// Basic-player-status class
class PlayerBasicStatus {
    public $healthPoint;
    public $magicPoint;
    public $spiritPoint;

    public function __construct($hp, $mp, $sp)
    {
        $this->healthPoint = $hp;
        $this->magicPoint = $mp;
        $this->spiritPoint = $sp;
    }
}

// Player-table class
class Table_Player {
    //
    // Private section
    //
    private $_dbc;

    //
    // Public section
    //
    public function __construct($dbc)
    { $this->_dbc = $dbc; }

    public function lockRead()
    { $this->_dbc->lockReadTable('Player'); }

    public function lockWrite()
    { $this->_dbc->lockWriteTable('Player'); }
```

```
public function updateBasicStatuses($id, $hp, $mp, $sp)
{
    $sql = <<<EOT
    UPDATE Player
    SET     HealthPoint = $hp,
           MagicPoint  = $mp,
           SpiritPoint = $sp
    WHERE  PlayerID   = $id
    EOT;
    $this->_dbc->sqlExec($sql);
}

public function getBasicStatuses($id)
{
    $sql = <<<EOT
    SELECT HealthPoint, MagicPoint, SpiritPoint
    FROM   Player
    WHERE  PlayerID = $id
    EOT;
    $this->_dbc->sqlExec($sql);

    $row = $this->_dbc->sqlFetchRow();
    if(!$row) return null;

    return new PlayerBasicStatus((int) $row[0], (int) $row[1], (int) $row[2]);
}
?>
```

model.php: An Example Model Class.

Explanation:

- The PlayerBasicStatus class is a helper class to contain the player's statuses.
- Basically, the Table_Player class job's is to wrap SQL commands in a consistent public API.
 - The constructor takes a DBConn instance to be used with all the queries.
 - The updateBasicStatuses() method generates an SQL UPDATE command using the given parameters and pass it to the sqlExec() method form the DBConn instance.
 - The getBasicStatuses() method generates an SQL SELECT command using the given parameters and pass it to the sqlExec() method form the DBConn instance. In then will convert the returned records to an instance of PlayerBasicStatus class.
 - The lockRead() and lockWrite() methods simply redirect the calls to the corresponding DBConn's methods while passing the correct table name.

5. Session Management and Security

Due to HTTP is a stateless protocol, a method is needed to maintain data between requests. The simplest method is to store the user data in cookies. However, for a bit better

security, using PHP's session would be a good idea. Basically, PHP will generate a special ID (a session ID) and store it as a cookie. Internally, PHP will associate the session ID with the script-supplied data (called session data). Using this method, only the server knows what data the PHP script actually stores (the browser only know the session name and ID). Hence, it may improve the security.

Rather than directly calling the PHP's session management functions, it would be a good idea to implement a class to manage session. Basically, the class should be able to:

- Provides a read-only access to all the session variables. PHP manual states that only one user (script file) can have access to session variables (open a session) at any given time. Hence, if there are multiple requests, they will be queued. If one of the script opens a session and then performs some complicated and lengthy operations, then the other requests can be queued for too long a time. Therefore, it is necessary for a script to open a session, update it, and close it as soon as possible (before doing any complicated/lengthy processing). The problem is that session variables can no longer be read if the script closes the session. The script (or in this case, the session management class) needs to be able to cache/copy the session variables in local variables.
- Provides a mechanism to open, close, and reopen session as needed. Note that opening a session while holding database table lock or transaction may lead to deadlock between requests.
- Manage the credential information of the currently logged-in user. This credentials may includes:
 1. ID of the user;
 2. Login name (user name) of the user;
 3. Visible name (or real name) of the user;
 4. Privilege of the user (such as 'is administrator' flag);
 5. Time zone offset (relative to UTC) of the user's browser.
- Manage some security-related variables, such as:
 1. Current captcha value;
 2. Random number;
 3. Last command time;
 4. Command counter (the number of commands a user has issued to the game).

The code snippet below shows an example of `Session` class that supports the above features.

```

<?php
/*
    Copyright (C) 2010 AnemoneSoft.com
        Aloysius Indrayanto

    This program is free software and comes with ABSOLUTELY NO WARRANTY.

    This program is licensed under the GNU Lesser General Public License (GNU LGPL)
    either version 3 of the license, or (at your option) any later version as
    published by the Free Software Foundation.
*/

// Session class
class Session {
    // Private flag to indicate if the session is open (the session variables are writeable)
    private static $_open;

    // Public data
    public static $currentUserID;           // Current user ID
    public static $currentUserLoginName;    // Current user login name
    public static $currentUserVisibleName;  // Current user visible name
    public static $currentUserIsAdmin;      // Flag to indicate if the current user is an
                                            // administrator
    public static $currentUserTimezoneOffset; // Timezone offset of the current user
                                            // (in seconds)
    public static $captchaValue;            // Captcha value
    public static $randomNumber;            // General purpose random number
    public static $lastCommandTime;         // Last command time (in seconds)
    public static $commandCounter;          // Command counter

    // Open session and read all available variables
    public static function open()
    {
        if(!self::$_open) {
            session_name('_my_game_session_id');
            session_cache_limiter('private_no_expire, must-revalidate');
            session_start();
            self::$_open = true;
        }

        self::$currentUserID           = isset($_SESSION['UID']) ? $_SESSION['UID'] : 0;
        self::$currentUserLoginName     = isset($_SESSION['ULN']) ? $_SESSION['ULN'] : null;
        self::$currentUserVisibleName   = isset($_SESSION['UVN']) ? $_SESSION['UVN'] : null;
        self::$currentUserIsAdmin        = isset($_SESSION['UIA']) ? $_SESSION['UIA'] : false;
        self::$currentUserTimezoneOffset = isset($_SESSION['UTZ']) ? $_SESSION['UTZ'] : 0;
        self::$captchaValue              = isset($_SESSION['CPV']) ? $_SESSION['CPV'] : '';
        self::$lastCommandTime           = isset($_SESSION['LCT']) ? $_SESSION['LCT'] : 0;
        self::$commandCounter            = isset($_SESSION['CMC']) ? $_SESSION['CMC'] : 0;

        if(isset($_SESSION['RNM']))
            self::$randomNumber = $_SESSION['RNM'];
        else {
            self::$randomNumber = mt_rand();
            $_SESSION['RNM'] = self::$randomNumber;
        }
    }

    // Close session (the session variables become read-only)
    public static function close()
    {
        session_write_close();
        self::$_open = false;
    }

    // Destroy session (the session variables become invalid)
    public static function destroy()
    {
        session_regenerate_id(true);
        session_destroy();
        clearstatcache();

        self::$currentUserID           = 0;

```

```

        self::$currentUserLoginName      = null;
        self::$currentUserVisibleName    = null;
        self::$currentUserIsAdmin        = false;
        self::$currentUserTimezoneOffset = 0;
        self::$captchaValue              = '';
        self::$randomNumber               = 0;
        self::$lastCommandTime           = 0;
        self::$commandCounter            = 0;
    }

    // Set current user information
    public static function setUserInfo($uid, $uln, $uvn, $adm, $tzo)
    {
        if(!self::$_open) die('Session handling bug in \'Session::\' . __FUNCTION__ . '()\');

        self::$currentUserID              = $_SESSION['UID'] = $uid;
        self::$currentUserLoginName       = $_SESSION['ULN'] = $uln;
        self::$currentUserVisibleName     = $_SESSION['UVN'] = $uvn;
        self::$currentUserIsAdmin         = $_SESSION['UIA'] = $adm;
        self::$currentUserTimezoneOffset  = $_SESSION['UTZ'] = $tzo;

        session_regenerate_id(true);
    }

    // Set captcha value
    public static function setCaptchaValue($captcha)
    {
        if(!self::$_open) die('Session handling bug in \'Session::\' . __FUNCTION__ . '()\');

        self::$captchaValue = $_SESSION['CPV'] = $captcha;
    }

    // Update last command time and return the time difference with the previous time
    public static function updateLastCommandTime()
    {
        if(!self::$_open) die('Session handling bug in \'Session::\' . __FUNCTION__ . '()\');

        $curTime = time();
        $difTime = $curTime - self::$lastCommandTime;

        self::$lastCommandTime = $_SESSION['LCT'] = $curTime;

        return $difTime;
    }

    // Reset the command counter
    public static function resetCommandCounter()
    {
        if(!self::$_open) die('Session handling bug in \'Session::\' . __FUNCTION__ . '()\');

        self::$commandCounter = $_SESSION['CMC'] = 0;
    }

    // Increment the command counter and return the new value
    public static function incrementCommandCounter()
    {
        if(!self::$_open) die('Session handling bug in \'Session::\' . __FUNCTION__ . '()\');

        ++self::$commandCounter;
        $_SESSION['CMC'] = self::$commandCounter;

        return self::$commandCounter;
    }
};

// Open the session automatically upon including this file
Session::open();

// Set the default timezone to UTC
if(function_exists('date_default_timezone_set')) date_default_timezone_set('UTC');
?>

```

session.php: An Example Session Class.

Explanation:

- The script file will automatically open a session upon inclusion and set the PHP's timezone to UTC (if supported by the installed PHP version).
- Since there will be only one instance of the class ever needed, all the member variables and methods are made static.
- The `open()` method will set the session name before opening the session. It is necessary to use unique session name (cookie variable) for each application/game. After opening a session, it will read all the session variables, or, set them with their defaults values if they are not available yet. If there is no random number defined yet, it will create a new random number using the Mersenne Twister algorithm by calling the PHP's `mt_rand()` function.
- The `close()` method will simply write and close the session. If a script has closed the session, it will still be able to read the content of the session variables by reading the local copies of the values in the class' member variables.
- The `destroy()` method will order PHP to regenerate a new session ID, destroy the current session, clear the cache, and invalidate all the class' member variables. Regenerating session ID can improve the security of the application/game.
- The other methods are used to update the values of the session variables. It is important to check if the script actually has an open session before attempting to update any of the session variables.

Usage of security-related variables:

- The `captchaValue` variable is used for storing the current captcha value (a sequence of random alphanumeric character). In the application/game registration page, it is a very good idea to use captcha to verify if the user that wants to register with the application/game is really a person and not an automated script/bot. Also, displaying and asking the user to enter the captcha once in a while in the gameplay would reduce the possibility that the user played the game using automated script/bot.
- The `randomNumber` variable would contains a different random number everytime the user log in. This random number can be use to mask or encrypt any/some information exchanged between the server and the user's browser. For example: in a game, there is an item store; rather than passing the real item ID, passing the masked ID (using XOR

operation) or encrypted ID (using DES/AES, for example) would increase the security of the game.

- The `lastCommandTime` variable can be used to track how fast a user sends command to the application/game. If a user has used a simple automation script, the time delay between commands may be too short or too uniform. Hence, it can be used for a simple detection mechanism if the user is a real person or not.
- The `commandCounter` basically is used to track the number of commands the user has send to the application/game. After a certain number of command, the application/game can present the user with a captcha page. This would reduce the possibility that the user played the game using automated script/bot. Note that only commands that will update the progress/statuses of the application/game that need to be counted; 'read only' commands that simply query the current status of the application/game should not be counted.

6. Captcha Generator Script

A captcha is a random sequence of character (usually alphanumeric) to challenge a client so that he/she can prove that he/she is a real person and not an automated script/bot. The code snippet below demonstrate a simple way to generate captcha image. Note that the PHP GD extension will need to be installed to run the script.

```
<?php
/*
    Copyright (C) 2010 AnemoneSoft.com
    Aloysius Indrayanto

    This program is free software and comes with ABSOLUTELY NO WARRANTY.

    This program is licensed under the GNU Lesser General Public License (GNU LGPL)
    either version 3 of the license, or (at your option) any later version as
    published by the Free Software Foundation.
*/

// Start session
include_once 'session.php';

// Force no cache for the page
header('Cache-Control: no-store, no-cache, must-revalidate');
header('Cache-Control: post-check=0, pre-check=0', false);
header('Pragma: no-cache');

// Helper function for generating random characters
function generateRandomCharacters()
{
    $sourceChars = array( array(48, 57), // Digits
                          array(97, 122) // Lowercase characters
                        );

    srand((double) microtime() * 1000000);

    $randomStr = '';
```

```

    for($i=0; $i < 6; ++$i) {
        $pos      = rand(0, sizeof($sourceChars) - 1);
        $randomStr .= chr(rand($sourceChars[$pos][0], $sourceChars[$pos][1]));
    }

    return $randomStr;
}

// Load the base image
$cmg = @imagecreatefromjpeg('image/' . rand(0, 4) . '.jpg');
if(!$cmg) {
    $cmg = imagecreatetruecolor(128, 32);
    imagefilledrectangle($cmg, 0, 0, 128, 32, imagecolorallocate($cmg, 0, 0, 0));
}

// Generate random characters and store them in the session variable
$rstr = generateRandomCharacters();
Session::setCaphcaValue($rstr);
Session::close();

// Determine the color
$colorMode = rand(0, 2);

// Prepare colors
if($colorMode == 0) {
    $darkColor  = imagecolorallocate($cmg, 32, 64, 64);
    $brightColor = imagecolorallocate($cmg, 128, 255, 255);
}
else if($colorMode == 1) {
    $darkColor  = imagecolorallocate($cmg, 64, 64, 32);
    $brightColor = imagecolorallocate($cmg, 255, 255, 128);
}
else if($colorMode == 2) {
    $darkColor  = imagecolorallocate($cmg, 64, 32, 64);
    $brightColor = imagecolorallocate($cmg, 255, 128, 255);
}

// Load font
$fid = @imageloadfont('anonymous.gdf');
if(!$fid) $fid = 5;

// Determine the printing mode
$printMode = rand(1, 4);

// Print the characters based on the mode
if($printMode == 1 || $printMode == 2) {
    $posX = 25;
    $posY = ($printMode == 1) ? 5 : 13;
    for($i = 0; $i < strlen($rstr); ++$i) {
        $chr = $rstr[$i];
        imagestring($cmg, $fid, $posX - 2, $posY - 2, $chr, $darkColor);
        imagestring($cmg, $fid, $posX + 2, $posY + 2, $chr, $darkColor);
        imagestring($cmg, $fid, $posX, $posY, $chr, $brightColor);
        $posX += 37;
        $posY = ($posY == 5) ? 13 : 5;
    }
}
else if($printMode == 3 || $printMode == 4) {
    $posX = 25;
    $posY = ($printMode == 3) ? 0 : 15;
    $incY = ($printMode == 3) ? 3 : -3;
    for($i = 0; $i < strlen($rstr); ++$i) {
        $chr = $rstr[$i];
        imagestring($cmg, $fid, $posX - 2, $posY - 2, $chr, $darkColor);
        imagestring($cmg, $fid, $posX + 2, $posY + 2, $chr, $darkColor);
        imagestring($cmg, $fid, $posX, $posY, $chr, $brightColor);
        $posX += 37;
        $posY += $incY;
    }
}

// Send the image
header('Content-type: image/jpeg');

```

```
imagejpeg($cimg, NULL, 100);  
imagedestroy($cimg);  
?>
```

captcha.php: A Simple Captcha Image Generator.

Explanation:

- The function `generateRandomCharacters()` is a helper function used for generating six random alphanumeric characters.
- The script will generate six random alphanumeric characters, save them as the captcha value in session variable and then close the session.
- The script will load a random background image using the `imagecreatefromjpeg()` function from PHP GD. If the loading failed, an empty image (black color) of 128x32 pixels will be generated.
- The script will then determine the colors to be used to print the captcha value.
- After loading the font and determining the printing mode, the script will print the captcha value to the image using the already determined color.
- Finally, the script will send the complete captcha image as a JPEG stream.

References

<http://php.net/index.php>, June 09, 2010

<http://www.mysql.com>, June 09, 2010

<http://php.net/manual/en>, June 09, 2010

<http://en.wikipedia.org/wiki/JSON>, June 12, 2010

http://en.wikipedia.org/wiki/Online_game, June 15, 2010

http://en.wikipedia.org/wiki/Flash_player, June 15, 2010

<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>, June 16, 2010

http://en.wikipedia.org/wiki/Data_Encryption_Standard, June 16, 2010

http://en.wikipedia.org/wiki/Advanced_Encryption_Standard, June 16, 2010

<http://en.wikipedia.org/wiki/CAPTCHA>, June 16, 2010

<http://en.wikipedia.org/wiki/Model-view-controller>, June 16, 2010