



# MTV Networks Game Services Client Programming Manual

Version 3.1.3 – February 10,2009

**NOTICE:** The information contained in this manual is confidential to MTV Networks. As a developer hired by MTV Networks you are to use this information only for the purpose of specifying, developing, integrating and testing specific projects as requested to conduct the business relationship and fulfill your requirements for delivery. You are to keep the information contained in this manual, as well as all other information provided by MTV Networks, strictly confidential, in accordance with the terms of the binding agreements in place.

## OVERVIEW

MTV Networks Game Services (herein referred to as GS) is a platform of services supporting online gaming.

GS is a comprehensive web-server based application service optimized for online games. GS supplies the common community services most online games require, such as player profiles, score submit, leader boards, badges, achievements, point accumulation, multiplayer services and much more. GS is a service implementing the business rules of online gaming – your application is responsible for all user interface and GUI display. A particular game can selectively choose to expose any subset or all of the GS services to its users.

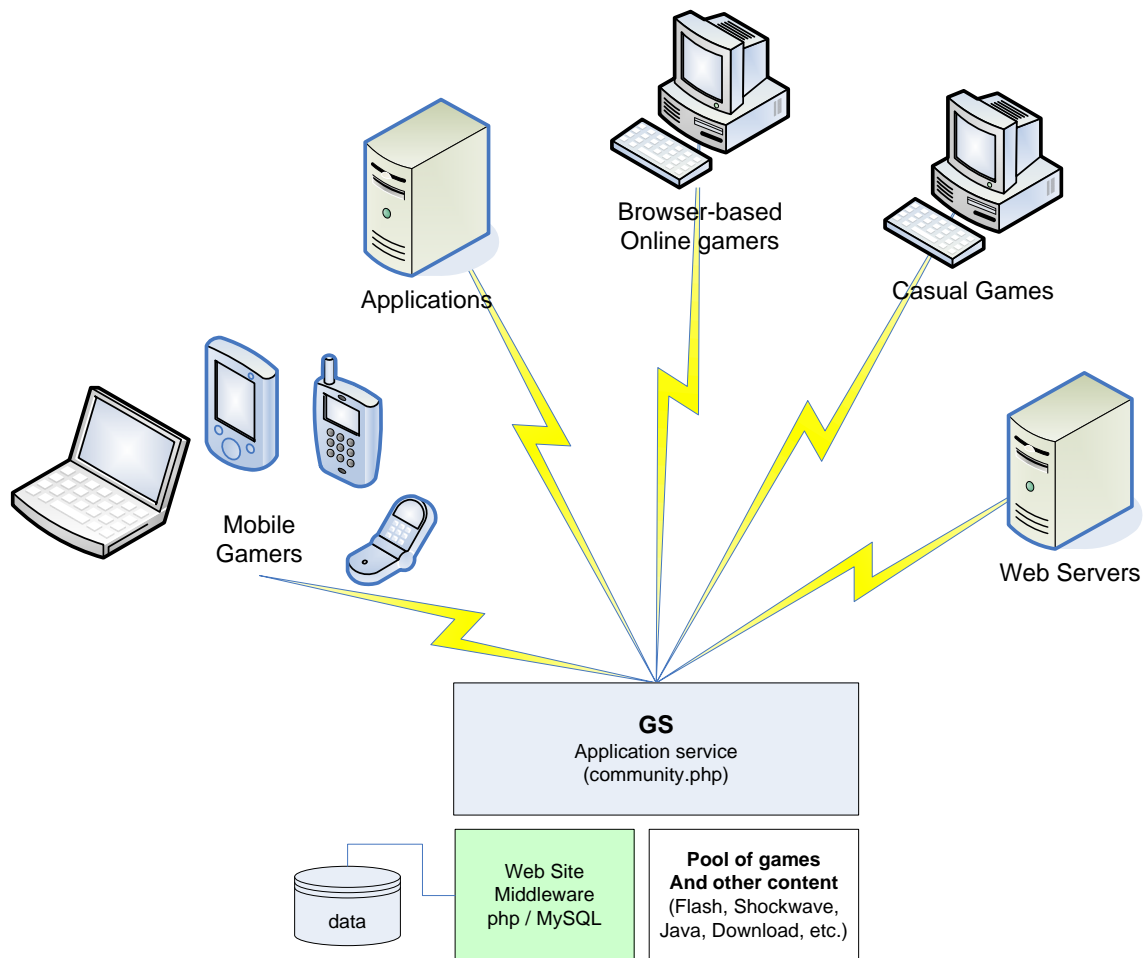


Figure 1: GS Platform Access Diagram

GS operates using a request/response model typical of client-server architecture. The client issues a service request containing a specific service and the necessary parameters. The server processes this request and replies with the status of the operation and any generated results. The purpose of the GS Client SDK is to make this process simple and straightforward for the developer.

## ***Overview of Services***

The services provided by GS:

- Send To Friend;
- In-game Event Tracking;
- Score submit, leader boards, ranking, badging, achievements and experience points;
- Custom avatars;
- User registration, profiles, authentication;
- User ratings for multiplayer games;
- Buddy list management;
- Voting;

- Notifications;
- Challenges;
- In-game Advertising.

In all of these cases your game is responsible for providing a GUI for the user, verifying user input. GS takes care of the business rules of each service and proper recording of the data.

### ***Affected Web Sites***

Any client communicating with GS to perform any function must do so on behalf of a specific site.

Web site currently utilizing GS functionality:

- arcade.mtv.com
- games.comedycentral.com
- games.vh1.com
- games.cmt.com

### ***Game Requirements for GS Integration***

All online games built for MTV Networks properties must follow these integration rules:

- Games must be built in Flash, Shockwave, Java or JavaScript. Current Flash support is ActionScript 2.0, Flash 8 and ActionScript 3.0, Flash 9.
- The game must load the GS SDK first and if the SDK cannot be loaded the game should fail with a user-friendly error message. The specific details about how to load the SDK are included in the source code samples.
- All game components must reference subordinate assets relative to a dynamic path determined at runtime. This is because the component is loaded from a content management system originating in a different folder than where the invoking file resides. We provide a function to help determine the path to your game folder:  
`GameServices.getBaseDir(null)`; For example, if your game needed to access the file `gamesettings.xml` and expected that file to be in the same folder as the loaded game file (i.e. the current directory), then your code is required to do the following:

```
var myFile = GameServices.getbaseDir(null) +  
            "gamesettings.xml";
```

Note the function supplies the trailing /.

- Start-up parameters are communicated into the game from the server from a CMS. How this happens is different based on the plug-in technology. We cannot support game-specific items passed into Flashvars or the URL to the game. It is best to use external files (XML) or HTTP requests to drive parameterized initialization data.

Please note: all requests for all files and external data must originate and terminate on MTV servers. We do not support references to external servers.

- All games larger than 1 megabyte must provide a file pre-loader that loads the game files with some animation indicating load progress.
- Games may not reference any resources on external web sites without the consent from the MTV game producer.
- We do not support any server-based services other than those described in this document. If your game requires specific services or data access not provided by GS it must first be approved by the MTV tech team. We will not deploy any CGI, php, Java etc. or databases developed by third party developers.
- All references to external assets should be parameterized as is best suited to the game such that URI references can be changed without rebuilding the game. All externally referenced assets must be stored in the same folder as the first file loaded or a subfolder therein. For example, store parameters in an XML file or a text file.
- Game tracking functions must be called in every game. Consult the MTV producer to determine where the proper calls are to be placed and the required data values. Game tracking functions are found in section Game Tracking on page 13.
- Stage sizes may not be larger than 780 pixels wide and 580 pixels high. If your online game must be larger than this please verify your size requirements with your site producer.
- All games must provide user sound control such that all sounds may be muted and unmuted.

- All string data intended for the GUI must be separated out into separate string tables and loaded by the game code. In-line code containing text is not acceptable.
- All graphical elements with embedded language, including text movie clips or graphics with embedded language, must be collected together in the library under a folder designated by the country code. All of our games are targeted to multiple countries and we need to make the job of localizing as simple as possible.
- All source code and necessary components to build a game must be delivered with the final release delivery of the game. MTV will not sign off on a completed project until the game can be built by MTV staff.

## Flash Programming Interface

We provide an object interface for the purpose of simplifying the interface to GS. The runtime code supporting all client-side GS services are provided in a compiled swf. A set of action script include files describe the services.

### File Manifest

The files contained in the GS Flash Package include:

testGS.html	Sample HTML to stage the tester application. This provides test configuration data through the flashvars parameter.
testGS.swf	Compiled version of the test movie.
testGS fla	Source to the test movie.
Sample_Game.html	Sample HTML to stage the sample game. This provides test configuration data through the flashvars parameter.
Sample_Game.swf	Compiled version of the sample game movie.
Sample_Game fla	Source to the sample game.
GSMain.as	Game Services start up functionality intended to be #included in frame 1 of your movie (or somewhere before any GS calls are required). You may be required to make changes to this code depending on how you develop your game.
gs1.swf	Game Services module implementing the GS version 1.x interface. (This is provided for backward compatibility with older games. It should not be used for new development.)
gs2.swf	Game Services module implementing the GS version 2.x interface.
as_metrics.swf	Game Services module.

omni_config.xml	Test configuration for tracking module.
mtv_gs/	Subfolder containing all the MTV GS SDK interface classes. You can move this folder where it makes sense in your development environment and then alter Publish Settings accordingly.
MTVN Game Services.pdf	This manual.
GameServices.as	Game Services class containing helper functions.
GSGameTracking.as	Game Tracking interface class.
GSScoreSubmit.as	Score Submit interface class.
GSSendToFriend.as	Send to Friend interface class.
GSRequest.as	Request object interface class.
GSResult.as	Result object interface class. The result object contains information from the server as a result of a previous request.

None of these files should be included in the delivery of the game's runtime assets.

## Getting Started

Look over the structure of testGS.fla and sampleGame.fla projects. These are test programs used to demonstrate some of the methods and may not be indicative of your needs. But it should suffice to help you get started.

The first thing we recommend you do in your game is load the gs module (gs1.swf, gs2.swf or gs3.swf). The gs module contains all our runtime services, objects and interfaces. When we distribute the SDK the gs module is located in the same folder as the testGS.fla. However, when your game is integrated in our live environment the gs module is located in a common area. Therefore we coded a conditional path assembly to accommodate local development and the live environment. We use a parameter passed in via Flashvars to detect where the module is loaded from. If this variable is not provided we assume we are developing locally and get the module from the current directory.

```

var g_isDev:Boolean = (_root.GSPath == null);
var g_mtvnGSPath:String = "gs1.swf";
if (g_isDev) {
    _root.game_id = "6";
    _root.site_id = "10";
    _root.SubmitURL = "http://gs.mtv-q.mtvi.com/community.php";
} else {
    g_mtvnGSPath = _root.GSPath + g_mtvnGSPath;
}

```

The `g_isDev` global variable can be used throughout your code to determine if you are integrated on a GS server or you are running in your local development environment.

The other variables noted in the code, `game_id`, `site_id` and `submitURL`, are passed in via Flashvars when the game is running on our servers. However, when developing locally these variables are undefined. Therefore the sample code sets them to generic values so you can test. You can ask your MTV producer for real values for your game but in most testing cases the sample values will work. You should never set these values in code or the game will not work properly when integrated in the live environment.

Some of the services may require immediate initialization. These include Send To Friend and Game Tracking. Use the code in frame 1 of `testGS.fla` as a guide.

## Callback Functions

You must set up call back functions based on the services you choose in your application. Almost all of the GS services are asynchronous, meaning they send a message to the server and return immediately. Sometime later the server replies with the result of your request.

There are two callback function fingerprints, one for the send to friend service and one for everything else. The reason for this is a legacy problem – early version of send to friend worked this way and we cannot upgrade these older games.

```
function stfCallback(action:String):Void
```

The action parameter can have one of the following values:

**nodata** – upon module initialization a request for game data is sent to the server. If there is no data this is the call back response.

**load** - upon module initialization a request for game data is sent to the server. This is the call back response when data is available.

**send** – when send is called this is the server response.

```
function gameServicesCallback(gsResponseInfo:GSResult,  
whichCommand:String):Void
```

A `GSResult` object is passed in to your callback function. This contains information about the result of the service call. Refer to each API for the expected results and refer to the section `GSResult Object` below. With `whchCommand` parameter is a string containing the service call requested in order for the callback function to identify what the result is referring to.

All object constructors accept the required callback function reference as a parameter. Usually when you construct a GS object you will need the callback readily available. However, you may pass `null` to the constructor and set the callback function later.

Sample call backs are provided in frame 1 of the sample code. You may use this code as a template or write your own version according to your needs.

## ***Score Submit***

When your game decides it is time to submit a score it should have a `GSScoreSubmit` instance ready. Construct this object with a callback function.

```
gs_scoreSubmit = new GSScoreSubmit(scoreSubmitCallback);
```

The call back function is notified when the server has replied with the result of the submission. There are 8 different score submit methods to choose from. Which one you use will depend on your game and how the MTV producer decides score submit should be handled. The choices are:

<code>submitScore</code>	Submit a score for the current authenticated user. Server replies with the result of the submission.
<code>submitScoreGetRank</code>	Submit a score for the current authenticated user and return the player's rank on the leader board. Server replies with the result of the submission. If successful the reply also contains the rank.
<code>submitScoreGetScores</code>	Submit a score for the current authenticated user and return a leader board. Server replies with the result of the submission. If successful the requested leader board is also returned.
<code>unauthSubmitScore</code>	Submit a score with a user-supplied screen name. Server replies with the result of the submission.
<code>unauthSubmitScoreGetRank</code>	Submit a score with a user-supplied screen name and return the player's rank on the leader board. Server replies with the result of the submission. If successful the reply also contains the rank.
<code>unauthSubmitScoreGetScores</code>	Submit a score with a user-supplied screen name and return a leader board. Server

	replies with the result of the submission. If successful the requested leader board is also returned
leaderBoard	Return the requested leader board.
unauthLeaderBoard	Return the requested leader board.
setUserIntermediateScore	Submit an intermediate score.
getUserIntermediateScores	Return intermediate scores for a given time period.

In all of these methods the game identifier is implied so they will only work for the current game set by the CMS.

### **Additional Game Data**

The Score Submit API contains a method for games to communication additional information about the score and the game play session. This additional data can be unstructured and only the game is responsible for interpreting it, or the data can be organized into key/value pairs and GS can process it for badge awarding and experience points.

An example of the unstructured data method we used is in the Redneck game called Deer Stacker. In this game we not only stored each user's score, but also the resulting configuration of the deer stack. The deer stack was a binary data structure that the game could interpret to display the stack when the score was displayed. The key/value pair structure is used in the South Park 10 game suite. Each game has 3 achievements associated with superb game play. Each game communicates its unique events to GS through predetermined keys set in the game data attribute of score submit. In order for this to happen the specific events require configuration by the GS admin team on the servers.

### **Submit Score for Mini-Games and Game Suites**

Some games are combinations of games, where one "shell" references many sub-modules, each its own game. When these games are considered separate games for leader board scoring, a different technique is used to identify the game.

Identifying the game grouping must be decided ahead of time with the MTV producer who will be integrating the game files on the GS servers. A game group is set up on the GS servers containing the information about each game in the group. Then on the web page hosting the game the game group information is passed into the game such that the SDK can handle it accordingly.

To submit a score, first identify which game you want to target. This is done with an index number. This call can happen at any time as the SDK holds the value until it is updated by the caller.

```
GSScoreSubmit.SetGameId(gameId);
```

The parameter gameId is an index number, starting at 1, indicating which game in the game group is targeted. Then call the score submit or leader board functions required for your game.

## **Authenticated vs. Unauthenticated Leader Boards**

GS provides two categories of score submission.

Authenticated leader boards are used when users login and verify their identity with the GS servers. Using this method a given user may appear on a specific leader board only once. If they submit a score that does not beat the current score they have recorded previously, no update is performed. If their new score beats the old score then the new score replaces the old score. Any game data and time played information is only recorded when the score is updated.

Unauthenticated scores simulate the arcade game leader boards where players cannot be identified so each time a user submits their score they also provide their name. Duplicate names are allowed as we are not able to authenticate the user and every score submit is recorded. In this scenario it is possible the entire leader board can be filled up by the same really good player who submits often.

## **Time Periods**

Game Services supports a concept called time periods for leader boards. This is all done on the server and little needs to be coded in the game to support this.

A game may be set up to support multiple periods of time for player ranking on the leader board. Any interval of time may be configured, but usually common calendar intervals make most sense for users. For example, a given game may support daily, weekly and monthly time periods. This allows multiple leader boards, or views, based on the indicated interval of time. When a user submits a score, GS checks each active time period set for that specific game to determine if the player should appear on that view.

For all games there is a "default" time period called all-time. GS tracks the all-time best score for all players without regard to time period. Common time periods are:

Time Period Type	Interval
0	All-time
10	Daily
20	Weekly
30	Monthly

Table 1: Time Period Types

All intervals are calendar time in the Eastern time zone (GMT-5).

Time periods other than all-time require configuration by the GS admin team on the servers.

### Intermediate Scores

Certain game mechanics, trivia games in particular, allow players to participate on a regular defined interval and then consider their final score as an aggregate sum of the individual game plays over that period of time. For example, a game is played once per day and the final leader board is the sum of each player's daily score over the calendar month.

### Experience Points

As users interact with various services they gain "experience." Currently experience is automatically calculated by the server from score values in the score submit process. There could also be data elements contained in the extra score data parameter that would trigger experience points. For example in a fighting game, beating the boss may trigger the addition of experience points for that player. There are other ways to earn experience, for example interacting with specific content on the hosting web site. There is no way to directly affect a user's experience points total from the SDK: all operations are read only.

GS supports three levels of experience point accumulation: game-based, game-group, and site-wide. Game based experience points counts experience points earned for each user while playing a specific game. Game group experience is earned for a group of games while playing any of the games in that group. And site-wide experience is earned while playing any game on the site as well as other site-based (non-game) interactions.

Experience points always accumulate. They are never taken away from users. For redeemable accumulators see the Token service (not currently documented.)

There are a variety of functions that work with experience points:

## Badges

Badges are associated to a user id when that user accomplishes some predetermined task. Usually badges are based on stellar game play based on score (e.g. beating a certain score), but a badge can also be earned for special in-game events (e.g. beating a specific boss, advancing past a particular level, finding an Easter-egg, and so on.)

Badge rules are defined on the server. All the necessary information to indicate a user has earned a badge is contained in the score submit. GS will either sense the score value or look into the extra game data supplied to determine if the user has earned the badge.

Certain rules apply to GS Badges: A user can only earn a specific badge once. Once a user has earned a badge the badge cannot be taken away. There can only be one badge for a specific event. Scores can be divided into ranges but only one badge can apply to any given range.

Functions that apply to badges:

<code>getUserBadges</code>	Return a list of badges earned by the current logged in user for the current game.
<code>getUserBadgesByTime</code>	Return a list of badges earned by the current logged in user for the current game in the specified time frame.
<code>getUserBadgesForGame</code>	Return a list of badges earned by the current logged in user for the specified game.
<code>getGameBadges</code>	Return a list of all available badges for the current game.

## ***Send To Friend***

The send to friend interface encapsulates the service of sending a message to another user. The message may contain game-specific data as is typical in the functionality of e-cards and sound boards. The message template and sending logic are all implemented on the server; your game is responsible for collecting and validating the send-to parameters.

Construct this object with a callback function.

```
gs_sendToFriend = new GSSendToFriend(stfCallback);
```

See section `GSSendToFriend` on page 40 for all the send-to-friend functions and parameters.

## Send To Friend Callback

The call back function will be notified when the server has replied with the result of the send. The call back function is also notified at start-up if there is data waiting for this instance. For example, if the user clicked a link in an email to view the composition from another user, that data is available at start-up. The call back function is notified and the app is able to get the data and reconstitute the original composition.

```
function stfCallback ( action:String ) {
    if (action == "nodata") {
        // we get here if there is no playbackid at startup
    } else if (action == "load") {
        // we get here if there is a playbackid at start up
        // get the data by calling
        // GSSendToFriend::getGameData();
    } else {
        // we get here after a SendToFriend is called and
        // the result comes back. Check for error by
        // calling GSSendToFriend::isError()
        // and GSSendToFriend::getError();
    }
}
```

## User Data

The send to friend service accepts game specific user data and captures that with the submission. Any time that specific record is recalled the user data is sent back exactly as it was recorded. Up to 8 kilobytes of data are accepted.

## Game Tracking

The tracking interface is designed to communicate certain game milestones to the tracking server. This allows metrics to be recorded for important game events used to determine the effectiveness of the game with the intended audience. The game milestones are:

**Load** This event is triggered when the game load is complete and the game is ready to play. This event is different than the load request event on the web server (triggered when the web server initiates the download of the content). Typically there is only one load event per game session.

The load event takes an additional parameter used to indicate which game component was loaded. In the case of games without multi-phased loading this parameter can be empty or null. In the case of multiple loading the parameter indicates which piece of the game was loaded (for example mini-game). This is a text string.

**Play** The game play event is triggered each time a new game is begun. Usually this is triggered by the user clicking a *start game* button or similar GUI element but some games automatically start play once loaded. It is

up to the game developer to call this milestone at the proper time. Depending on game design, there is at least one play event per session, possible more if the game design allows the player to start a new game.

The play event takes an additional parameter used to indicate which game component was played. In the case of games without mini-games this parameter can be empty or null. This is a text string.

**Game Over** The game over event is triggered when the game ends. This signals a completed game and is very important to gauge game play behavior. There should be a play event to match any game over event, but there may not be a game over event matching a play event if the player doesn't finish the game.

The game over event takes an additional string data parameter. The value of this parameter is completely up to the game as to how it is used. In most cases set it to an empty string. In games where a win/lose scenario makes sense this parameter should be set to either "WIN" or "LOSE".

**Zone Change** The zone change event is game specific. A zone change is a game event such as a new level or a new environment or some other significant event. This event requires an additional parameter to uniquely identify the event to the tracking system. The data parameter to this event uniquely defines the milestone such that the tracking team can query the event and build reports.

To set up game tracking perform the following steps:

- Declare a global `GSGameTracking` object in your project.

```
gs_gameTracking = new GSGameTracking();
```

This variable should be available from any code that will call game tracking services.

- Once you have a reference to the object call `initGameTracking` to set everything up.

```
gs_gameTracking.initGameTracking(null, false);
```

Very Important Note: The `omni_config.xml` file is provided for testing purposes only. It is not used in the live environment.

- After the game is fully loaded but before the game play begins (this will depend on the game design) call `sendHit` to indicate the load milestone was reached.

```
gs_gameTracking.sendHit(GSGameTracking.LOAD, "");
```

or in the case of a game with mini-games or multiple components:  
`gs_gameTracking.sendHit(GSGameTracking.LOAD, "MINIGAME1");`

- Any time a new game is started (e.g. the *Play* button is clicked) call *sendHit* as follows:

```
gs_gameTracking.sendHit(GSGameTracking.PLAY, "L1");
```

or:

```
gs_gameTracking.sendHit(GSGameTracking.PLAY, "MINIGAME1");
```

use the data parameter to indicate which level is beginning in level based games or which mini-game in game suites.

- Depending on game design and tracking requirements set by the game production team, mark any significant game milestones such as level up, pickup, ad view, etc. by calling *sendHit* with the zone change event. The data parameter to this event should be agreed upon by the metrics team.

```
gs_gameTracking.sendHit(GSGameTracking.ZONECHANGE, "L2");
```

- Any time the user completes a game call *sendHit* as follows:

```
gs_gameTracking.sendHit(GSGameTracking.GAMEOVER, "");
```

If applicable, provide the zone (level) the user attained when the game ended. This tag must be identical to the tag used in the zone change event. Call *sendHit* as follows:

```
gs_gameTracking.sendHit(GSGameTracking.GAMEOVER, "L2");
```

In a win/lose type game call *sendHit* as follows:

```
gs_gameTracking.sendHit(GSGameTracking.GAMEOVER, "WIN");
```

or

```
gs_gameTracking.sendHit(GSGameTracking.GAMEOVER, "LOSE");
```

Use the data parameter to indicate if the game ended in a winning or losing result.

## **Avatar**

User avatars may be loaded as movieclips or images. The GS services provide avatar images rendered in .png format in three sizes. The following APIs are available:

<code>loadAvatar</code>	Create a movieclip containing the current logged in user's avatar.
<code>loadAvatarForUser</code>	Create a movieclip containing the specified user's avatar.
<code>loadDefaultAvatar</code>	Create a movieclip containing the specified site default avatar.

In addition, your application is able to load the avatar image directly using this method:

```
GameServices.getGSWebRoot() +  
"games/avatarBuilder/composite.php?site_id="+site_id+"&size=0&user_id="  
+ user_id;
```

## ***Buddies***

Game Services provides a set of buddy management features. Buddy lists are used in chat and multiplayer games where a user can have access to a list of the users they frequently interact with. Buddy function are:

addUserBuddy	Add a user to the current logged in user's buddy list.
acceptuserBuddy	Accept a buddy request by another user.
getUserBuddies	Return the list of buddies assigned to the current logged in user.
getBuddyUsers	Return the list of users assigned as buddies to a given user.
deleteUserBuddy	Remove a buddy from the current logged in user's buddy list.
moveUserBuddy	Reorder the buddies in the list for the current logged in user.

## ***Voting and Polling***

The voting functions allow client interfaces to vote on objects in the game services database. For example, users can vote on games, other users or just about any category of content in GS. A vote is a numeric value that is ranged and interpreted by the application. For example a game voting application may allow vote values to range from 1 to 5 meaning 1 to 5 stars for given game\_id's. Then the application can use the aggregate functions to get the average for that object.

addOrUpdateVote	Add a vote value to the given item. User must be logged in. If the current user already voted on the intended object the vote is changed to the new value.
getNumberOfVotesPerValue	Return the number of votes for each value assigned to the given object.
getAverageVote	Return the average vote for the given object.
getTotalVotesByUser	Return the total number of objects the given user voted on.
getVoteResponse	Returns the total number of votes and the average vote value for the given object.
getUserVotesForGroup	Returns the total number of votes grouped by group key.

getUserVoteStatistics Returns vote statistics for the indicated user.

## **Notifications**

GameServices provides a notification system where short categorized messages are managed by user id. For example, an application may want to alert users about pending events related to their account, such as the addition of a new comment or a new buddy relationship. GS leaves it up to the application to determine the content and handing of notifications and only provides the API to manage the content and relationships to users. Notifications are private: users may view only the notifications they authored or received. All notification APIs only operate on behalf of the current logged in user.

createNotification	Create a new notification of a specific category for a given user. The notification is sent by the current logged in user.
getNotifications	Return attributes of all notifications sent to the current logged in user.
getNotificationsByUser	Return attributes of all notifications generated by the current logged in user.
countNotifications	Return a count of all notifications sent to the current logged in user.
acceptNotification	Mark a given notification as accepted.
deleteNotification	Remove the specified notification. Only the current logged in user who is the author or the recipient may remove a notification.

## **Notification Categories**

Notifications are categorized by GS subsystem. The following table enumerates the notification categories:

Notification Category Id	Description
1	<i>Private message:</i> user to user private message.
2	<i>Game event:</i> a notification event generated by a specific game_id containing information specific to that game. Currently we have not defined any such events.
3	<i>Comment:</i> another user added a comment.
4	<i>Buddy:</i> Another user added a buddy.
5	<i>Leader board:</i> a leader board event occurred. For

	example, you were on the top-10 and got knocked out.
6	<i>System event:</i> The system is sending a message. The "system" in this case being the application or specific site instance.

Table 2: Notification Categories

## **Challenges**

Users may challenge other users to play a single game or a group of games and get a private leader board (the game group must already exist.). A challenge has a fixed period of time where each participant should submit a score and earn a ranking.

As users participate in the challenge a challenge token is passed into the game alerting GS this is a challenge participant. Challenge tokens are not transferable, they may only be used by the user in the challenge they are assigned to. All challenge methods are members of the GameServices object.

isChallenge	Determine if the game is running in challenge mode.
createGameChallenge	Create a new challenge for a specific game. A user must be logged in to use this service.
createGameGroupChallenge	Create a new challenge for a game group. A user must be logged in to use this service.
addUserToChallenge	Add a single user to an existing challenge.
addUserListToChallenge	Add a list of users to an existing challenge.
addUserBuddiesToChallenge	Add all the user's buddies to an existing challenge.
addSiteUserToChallenge	Add a 3 <sup>rd</sup> party site user id to an existing challenge.
removeUserFromChallenge	Remove a single user from an existing challenge.
getChallengeInfo	Return information about an existing challenge.
getChallengeUsers	Return a list of users in an existing challenge and their status in the challenge.
getChallengeScoreRanks	Return a leader board of an existing challenge.

listChallenges	Return a list of all challenges for a given game.
listChallenges	Return a list of all challenges a given user is a member of.
declineChallenge	Decline (the current logged in user) to participate in the challenge.
closeChallenge	Closing a challenge will no longer allow scores to be submitted, but the challenge information and leader board are still available.
deleteChallenge	Delete the challenge and all data associated with it.
testChallengeToken	Verify the specified challenge token is authentic.
getChallengeToken	Returns the challenge token when the game is running in challenge mode. Maybe useful during testing but in production there isn't much you can do with the token.
setChallengeToken	Assign the challenge token, effectively putting the game in challenge mode. You would never do this in a production game. It is really only useful during development and testing.

## ***XML Result Stream***

All GS services respond with an XML stream. This XML stream is formatted as follows:

```
<?xml version='1.0' encoding='uft-8' ?>
<results>
  {<result>
  </result>}
  <status>
    <success>{1 or 0}</success>
    <message>{error_code}</message>
    <extended_info>{optional message explanation}</extended_info>
  </status>
  {<outparams>
    ...Output parameters vary based on function called...
  </outparams>}
  <passthru><fn>{function_name}</fn><state_seq>{#}</state_seq></passthru>
</results>
```

### **<result>**

This section contains the result data. The contents depend on the function request. Generally this will be treated as an array where each row of result data is an entry in the array and the entry is itself an array indexed by column name.

The result section may not appear if the operation produces no results or there was an error.

### **<status>**

The status section is mandatory. This tells you the status of your request. <success> will be a 1 indicating the operation succeeded. A 0 indicates failure and additional information is provided in <message> and <extended\_info>.

### **<outparams>**

Certain operations produce additional information regarding how to treat the result. For example, GetScoreRanks, a leader board query, returns additional information indicating the direction of the leader board (scores are ascending or descending order) and how to format the score number.

### **<passthru>**

A copy of the function request that generated this response. This section also includes a sequencing number. This is useful in the case when multiple calls to the same services are made and the results need to be matched to specific out-bound requests.

## **Programming Interface (the API)**

There are three objects in the Game Services architecture: GameServices, GSScoreSubmit and GSSendToFriend.

<b>GameServices</b>	contains most of the methods needed to interact with the Game Services platform.
<b>GSScoreSubmit</b>	contains all methods for dealing with score submission, leader boards, badges and experience points.
<b>GSSendToFriend</b>	handles the send to friend service. Note this object will be folded into GameServices in a future release.

### ***Objects & Methods***

#### **GameServices**

This object has most general services dealing with MTV Game Services.

### **GameServices ( callbackFn : Function )**

Constructor. Takes an optional parameter call back function reference. The call back function is called when an asynchronous function response arrives from the game services server.

### **setCallback ( callbackFn : Function ) : Boolean**

Set the call back function reference. The call back function is called when an asynchronous function response arrives from the game services server. Use this function to set the call back function when not done with the constructor.

The fingerprint of the call back function is:

```
function gameServicesCallback (gsResponseInfo:GSResult,  
whichCommand:String ):Void
```

`gsResponseInfo` contains the result of the operation. This is described in further detail in GSResult Object.

`whichCommand` contains the API that was requested and generated this response.

This function returns *true* when the call back function is set, *false* if the parameter is not valid.

### **setSendToFriendCallback(callbackfn:Function):Boolean**

The send to friend service works a little different that other game service APIs, therefore the call back function has a different fingerprint.

```
function stfCallback ( action:String ):Void
```

The action parameter can have one of the following values:

**nodata** – upon module initialization a request for game data is sent to the server. If there is no data this is the call back response.

**load** - upon module initialization a request for game data is sent to the server. This is the call back response when data is available.

**send** – when send is called this is the server response.

This is the same function as `GSSendToFriend.setCallback()`. It is provided here because the `GSSendToFriend` object will be phased out in the future.

**getVersionString() : String**

Returns the version of the game services build.

**getSubmitURL() : String**

Returns the URL of the game services server. For example  
`http://gs.mtv.com/community.php`

**getLoginURL() : String**

Returns the URL where to call login on the game services server. For example  
`http://gs.mtv.com/communitylogin.php`

**getGSWebRoot() : String**

Returns the root of the game services web server. For example  
`http://gs.mtv.com/`

**getSiteId() : Number**

Returns the current site id.

**getBaseDir( mc : MovieClip ) : String**

Returns the path on the game services server where the current .swf is actually hosted. If the .swf is trying to access assets relative to itself (for example, in subfolders in the same folder) the developer must call this function to locate the .swf folder. It is NOT in the current directory. For example, if your game needed to access the file `gamesettings.xml` and expected that file to be in the same folder as the loaded game file, then your code will do the following:

```
var myFile = GameServices.getbaseDir(null) + "gamesettings.xml";
```

If it is determined (by looking at the Flashvar `GSPATH`) that the game does not appear to be running from a GS server, this function will return an empty string to signify "user the current directory".

**isChallenge () : Boolean**

Returns *true* if the game is operating as a challenge. The GS servers pass a token into the game when it is running as a challenge. See the section Challenges on page 18.

The game should act the same in challenge mode, but you may want to show the challenge leader board or user status within the challenge.

```
public static function setChallengeToken(challengeToken:String):Boolean;
```

**getChallengeToken () : String**

Returns the challenge token. This is provided for testing purposes only. In production there isn't anything useful you can do with the challenge token. See the section Challenges on page 18.

**setChallengeToken (challenge\_token:String) : Boolean**

Assigns the challenge token to the game, effectively putting the game in challenge mode. This is provided for testing purposes only. In production you should never do this as the GS servers assign the token. See the section Challenges on page 18.

Functions that initiate a transaction to the GS server and communicate the result with a GSResult object through the call-back function:

**login ( username: String, password : String ) : Boolean**

Attempt to login the user using the credentials supplied. The call back function is called with the result of the operation. The results include the same response data as *getRegistrationStatus* when successful.

Normally we would not expect login to occur from within individual games.

**logout () : Boolean**

Attempt to logout the current logged in user.

**getRegistrationStatus() : Boolean**

Get information about the current logged in user. The call back function is called with the results. If there is a user logged in you will get the following items:

<i>user_id</i>	the logged in user's id
<i>user_name</i>	the logged in user name
<i>date_created</i>	the date the account was created
<i>date_updated</i>	the date this user's profile was last updated
<i>last_login</i>	the date and time of the last login
<i>login_count</i>	the number of times this user logged in
<i>reg_confirmed</i>	true if this user's registration is confirmed
<i>view_count</i>	number of times the user's profile was viewed by other users.
<i>site_experience_points</i>	Number of site-wide experience points earned by this user.
<i>site_user_id</i>	the site identity of the co-registered hosting site.

### **getRegistrationStatusEx() : Boolean**

Extended version of `getRegistrationStatus`, adding a few user-specific counters. Note this function is considerably less efficient than `getRegistrationStatus`. If there is a user logged in you will get the following items:

<code>user_id</code>	the logged in user's id
<code>user_name</code>	the logged in user name
<code>date_created</code>	the date the account was created
<code>date_updated</code>	the date this user's profile was last updated
<code>last_login</code>	the date and time of the last login
<code>login_count</code>	the number of times this user logged in
<code>reg_confirmed</code>	true if this user's registration is confirmed
<code>view_count</code>	number of times the user's profile was viewed by other users.
<code>site_experience_points</code>	Number of site-wide experience points earned by this user.
<code>buddy_count</code>	Number of accepted user buddies.
<code>comment_count</code>	Number of accepted user comments.
<code>notification_count</code>	Total number of notifications (both accepted and pending).
<code>site_user_id</code>	the site identity of the co-registered hosting site.

### **initGameTracking ( configFileName:String, useDebug:Boolean ):Boolean**

Turn on the in-game tracking functionality. This function must be called before any calls to `sendHit`.

`configFileName` should normally be an empty string. While testing from your local environment (a non-MTV server) you should use the XML file supplied by MTV. However, when delivering the any build for integration on MTV servers this parameter must be an empty string.

`useDebug` should be set to false. This is set to true to produce debug output (trace messages) related to game tracking.

### **sendHit( hitType:String, hitData:String ):Boolean**

#### **Report an in-game event. Refer to the section User Data**

The send to friend service accepts game specific user data and captures that with the submission. Any time that specific record is recalled the user data is sent back exactly as it was recorded. Up to 8 kilobytes of data are accepted.

Game Tracking on page 13.

Example:

```
GameServices.sendHit(GSGameTracking.LOAD, loadDetails);
```

### **getUserBuddies( accepted:Boolean ):Boolean**

Return a list of other users this current logged in user chose as buddies. The accepted parameter controls the results returned by this function. When accepted is true the function only returns buddy assignments for the current logged in user that have been accepted. When accepted is false this function returns buddy assignments for the current logged in user that have not been accepted.

The result from this operation is an array of zero or more the following data items:

<i>buddy_user_id</i>	The buddy's user id.
<i>user_name</i>	The buddy user name
<i>position</i>	Indicates how the list is sorted.
<i>group_id</i>	Group identifier.
<i>date_created</i>	the date the assignment was made.

### **addUserBuddy( buddyUserId:Number, groupId:Number ):Boolean**

Assigns the supplied user id to the buddy list for the current logged in user. When a new buddy is assigned it is initially marked as "not-accepted" such that the target user would have to accept the assignment. See acceptBuddyUser.

The group identifier is used to group buddy by some site-defined category. Typically you would supply 0, but you could for instance set up categories such as Friends, Opponents, Family, Classmates, etc. But these groupId values would have to be assigned on a site-wide basis such that all users grouped by the same values.

This function returns via the callback only an indication if the assignment was made or if it failed.

### **acceptUserBuddy( buddyUserId:Number ):Boolean**

Accepts the buddy assignment.

### **moveUserBuddy( buddyUserId:Number, newPosition:Number ):Boolean**

Moves the indicated user to the indicated position, shifting all users in between the new position and the original position. Buddy positions are zero-based.

**deleteUserBuddy( buddyUserId:Number ):Boolean**

Deletes the indicated user from the buddy list, shifting all users after the indicated buddy's position.

**getBuddyUsers( buddyUserId:Number ):Boolean**

Returns all the users the indicated user is a buddy of. The result information comes by the callback and is an array of zero or more elements containing the following attributes:

<i>user_id</i>	The buddy's user id.
<i>user_name</i>	The buddy's user name.
<i>date_added</i>	The date the buddy was added to the list.

**createNotification( categoryId:Number, categoryKey:Number recipientId:Number, msg:String ):Boolean**

Create a new notification from the current logged in user sent to the notification queue of the indicated recipient.

<i>categoryId</i>	The notification category id (see table Notification Categories).
<i>categorykey</i>	The id of the object that is the subject of the categoryId. Usually this is 0.
<i>recipientId</i>	The intended recipient.
<i>msg</i>	The content of the notification.

**getNotifications( categoryId:Number, firstItem:Number numItems:Number ):Boolean**

Return the content of notifications queued to the current logged in user.

<i>categoryId</i>	Return only notifications in the indicated category id (see table Notification Categories). Pass 0 for all categories.
<i>firstItem</i>	For paging, first item to return. Use 0 for first item.
<i>numItems</i>	Number of items from firstItem to return.

For each notification, the result object will contain the following elements:

<i>notification_id</i>	Unique notification identifier. You will need this id later to call acceptNotification or deleteNotification.
<i>date_submitted</i>	Date and time notification was originally submitted.
<i>category_id</i>	Category id assigned to the notification (see table Notification Categories).

<i>category_key</i>	Object id associated with category id.
<i>creator_id</i>	User id of the user who created (sent) this notification.
<i>recipient_id</i>	User id of the recipient (the current logged in user).
<i>date_viewed</i>	Date and time notification was accepted.
<i>msg</i>	The content of the notification.
<i>user_name</i>	The user name matching the user id of the person who created the notification.

Notifications are grouped by category id and sorted by date created.

**loadAvatar( size:Number, containerClip:MovieClip clipName:String, depth:Number ):MovieClip**

Return a MovieClip object containing the requested avatar for the current logged in user.

<i>size</i>	An avatar size request. Check with your site producer for the available sizes. Typically there are three avatar sizes but this could change by site. 0=full size, 1=medium size, 2=thumbnail.
<i>containerClip</i>	The parent movie clip containing this clip.
<i>clipName</i>	Name to assign to the avatar movie clip.
<i>depth</i>	Movie clip depth.

**loadAvatarForUser( userId:Number, size:Number, containerClip:MovieClip clipName:String, depth:Number ):MovieClip**

Return a MovieClip object containing the requested avatar for the indicated user.

<i>userId</i>	GameServices user id of the user you want to load an avatar on behalf of.
<i>size</i>	An avatar size request. Check with your site producer for the available sizes. Typically there are three avatar sizes but this could change by site. 0=full size, 1=medium size, 2=thumbnail.
<i>containerClip</i>	The parent movie clip containing this clip.
<i>clipName</i>	Name to assign to the avatar movie clip.
<i>depth</i>	Movie clip depth.

**loadDefaultAvatar( genderId:Number, size:Number, containerClip:MovieClip clipName:String, depth:Number ):MovieClip**

Return a MovieClip object containing the requested site-wide default avatar.

<i>genderId</i>	Gender identifier used to indicate which default avatar you want to load. Most sites would have two genders: 1=male, 2=female. However, some sites also offer other options.
<i>size</i>	An avatar size request. Check with your site producer for the available sizes. Typically there are three avatar sizes but this could change by site. 0=full size, 1=medium size, 2=thumbnail.
<i>containerClip clipName depth</i>	The parent movie clip containing this clip. Name to assign to the avatar movie clip. Movie clip depth.

Each site has a default avatar assigned to users if they do not choose to set up their own avatar.

#### **findUserId( userName:String ):Boolean**

Find a user id matching the specified userName. Returns only one item if a match is made, otherwise returns no results.

<i>userName</i>	User name string to match. Performs exact match only.
-----------------	---

#### **findUsers( searchFor:String, matchBeginningOfNameOnly:Boolean ):Boolean**

Find the user id(s) of user names matching the specified search string. Returns up to 100 items if a match is made. Returns an empty result if no match is found.

<i>searchFor</i>	User name string to search for.
<i>matchBeginningOfNameOnly</i>	When <i>true</i> , search only names that begin with the search string. When <i>false</i> , find the search string anywhere in the user names. Specifying <i>false</i> is a much less efficient search.

#### **findUsersEx( searchFor:String ):Boolean**

Find user id(s) with profile data matching the specified search string. Returns up to 100 items if a match is found. Returns no results if no match is found.

<i>searchFor</i>	String to search for. Searches the following profile data items: user_name, tagline, about_me and additional_info.
------------------	--

Please note this function is very slow.

### **getUserBadges():Boolean**

Return a list of all badges earned by the current logged in user for all games.

In the GSResult object the following attributes are returned if the user has one or more badges:

<i>badge_id</i>	Unique id of the badge.
<i>badge_name</i>	Name of the badge.
<i>large_img_url</i>	URL to the large image of the badge.
<i>small_img_url</i>	URL to the small badge image.
<i>short_desc</i>	Badge description.
<i>long_desc</i>	More detailed description.
<i>game_id</i>	game_id associated with the badge.
<i>position</i>	Ordering of badge with respect to the game.

### **getUserBadgesByTime( start\_date:Date, end\_date:Date ):Boolean**

Return a list of all badges earned by the current logged in user in the given time frame for all games.

<i>start_time</i>	Beginning date (on or after).
<i>end_time</i>	Ending date (up to and including).

Returns the same result set as **getUserBadges**.

### **getUserBadgesForGame():Boolean**

Return a list of all badges earned by the current logged in user for only the current game.

Returns the same result set as **getUserBadges**.

### **getGameBadges():Boolean**

Return a list of all badges available to the current game.

Returns the same result set as **getUserBadges**.

### **getChallengeInfo (challengeId:Number):Boolean**

Given a specific challenge id return the information about that challenge.

<i>challengeId</i>	which challenge to request info.
--------------------	----------------------------------

### **getChallengeUsers (challengeId:Number):Boolean**

Given a specific challenge id return the list of users who are members of that challenge.

*challengeId*                      which challenge to request info.

### **createGameChallenge (start\_time:String, end\_time:String, num\_submits\_per\_player:Number, public\_results:Boolean, title:String, list\_of\_user\_ids:String):Boolean**

Create a new challenge for the current game.

<i>start_time</i>	Date and time when the challenge begins.
<i>end_time</i>	Date and time when the challenge ends.
<i>num_submits</i>	Number of score submits each user is allowed.
<i>public_results</i>	Set to true if the results of the challenge are public.
<i>list_of_user_ids</i>	Comma separated list of GS user_id's who are to be made members of the challenge.
<i>title</i>	A title to identify the challenge.

### **createGameChallengeForBuddies (game\_id:Number, start\_time:String, end\_time:String, num\_submits\_per\_player:Number, public\_results:Boolean, title:String):Boolean**

Create a new challenge for the specified game id and include all the current logged in user's buddies.

### **createGameChallengeForSiteUsers (game\_id:Number, start\_time:String, end\_time:String, num\_submits\_per\_player:Number, public\_results:Boolean, title:String, list\_of\_site\_user\_ids:String):Boolean**

Create a new challenge for the specified game id and include all the listed site user ids. Site user id's is a comma separated list of co-registered user id's from the site. If a given site id is not currently an existing GS user then a new GS user is created.

**createGameGroupChallenge (game\_group\_id:Number, start\_time:String, end\_time:String, num\_submits\_per\_player:Number, public\_results:Boolean, title:String, list\_of\_user\_ids:String):Boolean**

Create a new challenge for the specified game group.

**createGameGroupChallengeForBuddies (game\_group\_id:Number, start\_time:String, end\_time:String, num\_submits\_per\_player:Number, public\_results:Boolean, title:String):Boolean**

Create a new challenge for the specified game group and include all the current logged in user's buddies.

**createGameGroupChallengeForSiteUsers (game\_group\_id:Number, start\_time:String, end\_time:String, num\_submits\_per\_player:Number, public\_results:Boolean, title:String, list\_of\_site\_user\_ids:String):Boolean**

Create a new challenge for the specified game group and include all the listed site user ids. Site user id's is a comma separated list of co-registered site id's. If a given site id is not currently an existing GS user then a new GS user is created.

**addUserToChallenge (challenge\_id:Number, user\_id:Number):Boolean**

Add the specified user to the challenge. Only the owner of the challenge or a member of the challenge may add users to the challenge.

*challenge\_id*            The challenge in question.

*user\_id*                Which user to add to the challenge.

**addUserListToChallenge (challenge\_id:Number, user\_id\_list:String):Boolean**

Add the specified users to the challenge. *user\_id\_list* is a string of comma separated GS user id's. Only the owner of the challenge or a member of the challenge may add users to the challenge.

*challenge\_id*            The challenge in question.

*user\_id\_list*            List of comma separated user id's to add to the challenge.

### **addUserBuddiesToChallenge (challenge\_id:Number):Boolean**

Add the current logged in user's buddies to the challenge. Only the owner of the challenge or a member of the challenge may add users to the challenge.

*challenge\_id*            The challenge in question.

### **addSiteUserToChallenge (challenge\_id:Number, site\_user\_id:String):Boolean**

Add the indicated 3<sup>rd</sup> party site user id to the challenge. Only the owner of the challenge or a member of the challenge may add users to the challenge.

*challenge\_id*            The challenge in question.

*site\_user\_id*            The user id from the 3<sup>rd</sup> party site.

This function supports direct adding of co-registered users to a challenge. If the user is unknown to GS then a new internal account is automatically created.

### **addSiteUserListToChallenge (challenge\_id:Number, site\_user\_id\_list:String):Boolean**

Add the indicated 3<sup>rd</sup> party site user id's to the challenge. Only the owner of the challenge or a member of the challenge may add users to the challenge.

*challenge\_id*            The challenge in question.

*site\_user\_id\_list*        A comma separated list of user id's from the 3<sup>rd</sup> party site.

This function supports direct adding of co-registered users to a challenge. If the user is unknown to GS then a new internal account is automatically created.

### **getChallengeScoreRanks (challenge\_id:Number, first\_rank:Number, num\_ranks:Number):Boolean**

Return the leader board of the indicated challenge.

*challenge\_id*            The challenge in question.

*first\_rank*              First rank to return. Use 1 to return the top position.

*num\_ranks*              Number of ranks to return from the *first\_rank*.

**listChallenges ():Boolean**

Return a list of all challenges the current user is a member of.

**removeUserFromChallenge (challenge\_id:Number, remove\_user\_id:Number):Boolean**

Remove the specified user from the indicated challenge.

*challenge\_id*            The challenge in question.

*remove\_user\_id*        User to remove.

**declineChallenge (challenge\_id:Number):Boolean**

The current logged in user declines participation in the specified challenge.

*challenge\_id*            The challenge in question.

**deleteChallenge (challenge\_id:Number):Boolean**

Delete the specified challenge.

*challenge\_id*            The challenge in question.

**closeChallenge (challenge\_id:Number):Boolean**

Close the specified challenge. Once closed, no more scores may be submitted, but the data is still available such as leader boards and challenge info.

*challenge\_id*            The challenge in question.

**testChallengeToken (challenge\_token:String):Boolean**

Test the challenge token to determine its validity.

*challenge\_token*        The challenge token to test.

**GSScoreSubmit**

The score submit object, GSScoreSubmit, encapsulates all functionality related to submitting scores to the server, rendering leader boards, player ranking, and experience points.

## Local functions that do not initiate a transaction to the GS server:

### **GSScoreSubmit(callbackFn:Function)**

Construct the GSScoreSubmit object with the call back function. Call back function is optional here but if not set with the construction you must call *setCallback* before any server-side service is called.

This can be the same call back function as set with the GameServices object.

### **setCallback(callbackfn:Function):Boolean;**

You must set the call back function before calling any server-side service.

This can be the same call back function as set with the GameServices object.

### **getGameId():Number**

You can get the game id assigned to the game, if needed. Under normal circumstances you do not need this. When working with game groups this will return the game id referenced by the game index number set with *setGameTargetId()*.

Never hard-code game id's in your game. Game id's are dynamically assigned by the server and are subject to change without notice.

### **setGameId(gameId:Number):Boolean**

When working with game groups (a set of games acting together as a game suite) you can tell game services which game id is now active.

For example, if a game group contains game id's [34, 66, 22, 397] then *setGameId(397)* would indicate to GS the 4<sup>th</sup> game is now the active game. You cannot reference a game id that is not in the game group.

### **setGameTargetId(gameIndex:Number):Boolean**

When working with game groups (a set of games acting together as a game suite) you can tell game services which game is now active. This is a one-based index number from 1 to the number of games in the suite.

For example, if a game group contains game id's [34, 66, 22, 397] then target index 4 refers to game id 397.

### **getGameIdFromTargetId(gameIndex:Number):Number**

Given a game index number return the game id.

**recordScore(score1:Number, score2:Number):Void**

Currently not used.

**setGameDataValue(gameDataKey:String, gameDataValue:String):Boolean**

Adds the key/value pair to the next score submit event. This data is stored with the score and will be returned any time the score entry is returned (for example in leader boards). This information may also be processed by the GS back-end for badges and experience points when properly configured to do so.

**getGameDataValue(gameDataKey:String):Boolean**

Returns the value associated to gameDataKey. Use this to read a value back after it was set or after a leader board entry was set using setGameDataFromString(). Note that immediately after a score submit operation the game data is cleared.

**clearGameData():Boolean**

Resets the game data previously set. This is useful when intermediate values may have been set before a score submit but the user cancelled the game or decided to play again. This way events that occurred for a quit game session are not associated with the wrong session.

**getGameDataCount():Number**

Returns the number of entries in the game data object.

**setGameDataFromString(gameData:String):Boolean**

Allows setting up the object from a formatted data string returned from a leader board query. The reason this is needed is because leader board queries may return hundreds of entries and it makes little sense to create internal game data objects for all of them. In this case the programmer would call this method on the entries of interest.

Functions that initiate a transaction to the GS server and communicate the result with a GSResult object through the call-back function:

**submitScore(score1:Number, score2:Number, extraScoreData:String):Boolean**

Authenticated score submission. A user must be authenticated and logged in with the game services server for this function to succeed.

For now score1 and score2 should be the same value. This is the final score the user earned in the game. Note for games what support a play again

feature the game should submit the best score earned after a series of game plays. This may not necessarily be the most recent score earned.

Extra score data is a string used to record any special data related to the game play. For example, in a golf game you may want to record birdies and eagles and hole-in-ones. This string is specific to the game and can not be longer than 255 characters. However, the value(s) used must be consistent and able to be parsed by simple logic.

**submitScoreGetRank(score1:Number, score2:Number, extraScoreData:String) : Boolean**

Same as submitScore only returns the user's rank on the all-time leader board.

**submitScoreGetScores(score1:Number, score2:Number, extraScoreData:String, startPos:Number, numScores:Number) : Boolean**

Same as submitScore only returns a leader board. Refer to the leaderboard function for information about the leader board.

<i>startPos</i>	one-based index indicating the first rank on the leader board to return.
<i>numScores</i>	how many leader board positions to return after the startPos.

For example, if you wanted the top 100 scores then send 1 for startPos and 100 for numScores. If you wanted the section of the leader board from 150 to 500 then set startPos to 150 and numScores to 350.

**leaderBoard (firstRank:Number, numRanks:Number, timePeriodType:Number, timePeriod:Number):Boolean;**

Get the authenticated leader board for the current game.

<i>firstRank</i>	one-based index indicating the first rank on the leader board to return.
<i>numRanks</i>	how many leader board positions to return after the first rank requested.
<i>timePeriodType</i>	which time period to request. Refer to Table 1: Time Period Types on page 11.
<i>timePeriod</i>	which specific time period to query. To return the leader board for the current time period use -1.

For example, if you wanted the top 50 all-time scores then send 1 for firstRank, 50 for numRanks, 0 for timePeriodType (all-time) and -1 for timePeriod (current time period). If you wanted the section of the leader board from 150 to 500 then set firstRank to 150 and numRanks to 350.

**setUserIntermediateScore(score:Number, extraScoreData:String) : Boolean**

Store an intermediate score for the current game on behalf of the current logged in user.

This function adds one row of intermediate score data for the game in question. Will also add score to any current time periods for the game in question.

Intermediate scores are discussed in Intermediate Scores on page 11.

**unauthSubmitScore(score1:Number, score2:Number, extraScoreData:String, screenName:String):Boolean**

Submit score for an unauthenticated user. Same as submitScore only a user name is provided. This is the name that will appear on the leader board.

**unauthSubmitScoreGetRank(score1:Number, score2:Number, extraScoreData:String, screenName:String) : Boolean**

Submit score for an unauthenticated user and return the resulting rank on the leader board.

**unauthSubmitScoreGetScores(score1:Number, score2:Number, extraScoreData:String, screenName:String, startPos:Number, numScores:Number) : Boolean**

Submit score for an unauthenticated user and return the leader board.

**unauthLeaderBoard (firstRank:Number, numRanks:Number):Boolean**

Get the unauthenticated leader board for the current game.

**getScoreRanks (gameId:Number, timePeriodType:Number, timePeriod:Number, firstRank:Number, numRanks:Number):Boolean**

Same as leaderBoard only this API allows you to specify which game and the time period. Time period specification must match what is set up on the server as it is not specified when submitting a score.

<i>gameId</i>	which game id to request. Pass null to use the current game.
<i>timePeriodType</i>	which time period to request. Refer to Table 1: Time Period Types on page 11.
<i>timePeriod</i>	which specific time period to query. To return the leader board for the current time period use -1.
<i>firstRank</i>	one-based index indicating the first rank on the leader board to return.
<i>numRanks</i>	how many leader board positions to return after the first rank requested.

**getUserScoreRanks (priorRanks:Number, numRanks:Number):Boolean**

Get a leader board relative to the position of the current logged in user.

For example, `getUserScoreRanks(10, 20)` will return the 10 ranks before the current users position on the leader board and the 20 ranks from that position for a total of 20 leader board entries.

**getUserScore (gameId:Number, timePeriodType:Number, timePeriod:Number):Boolean**

Return the user's best score for the given game. Supply 0 for gameId to indicate the current game. Otherwise supply a game id to indicate a specific game.

For example, `getUserScore(0, 0, 0)` will return the current logged in user's best score for the current game for the all-time leader board.

**getUserScoreForUser (userId:Number, gameId:Number, timePeriodType:Number, timePeriod:Number):Boolean**

Return the specified user's best score for the given game. Supply 0 for gameId to indicate the current game. Otherwise supply a game id to indicate a specific game.

For example, `getUserScore(146, 0, 0, 0)` will return the best all-time score in the current game for user 146.

**getUserGameRank(userId:Number, gameId:Number, timePeriodType:Number, timePeriod:Number):Boolean**

Return the user's rank in the given leader board.  
Specify 0 for the userId to return the current logged in user's rank.  
Provide a particular game id to get the rank in that game, or supply 0 for the current game (when using `setGameTargetId()`).

**getSiteUserRank():Boolean**

Return the user's site-wide ranking.

**getUserIntermediateScore(timePeriod:Number, firstItem:Number, numItems:Number) : Boolean**

Return all intermediate scores for the current game in the requested time period. Use -1 to indicate the current time period. Otherwise pass a time period offset.

**getRatingRanksForGame(game\_id:Number, first\_rank:Number, num\_ranks:Number):Boolean**

Return the ranking list of users by rating value for the given game. Supply 0 for the gameId to indicate the current game.

**getRatingRanks(first\_rank:Number, num\_ranks:Number):Boolean**

Return the ranking list of users by rating value for the current game.

**getUserRatingRanksForGame(game\_id:Number, prior\_ranks:Number, num\_ranks:Number):Boolean**

Return the ranking list of users by rating value for the given game relative to the position of the current logged in user.

For example, if the current logged in user is ranked 50<sup>th</sup>, then calling `getUserRatingRanksForGame(0,10,20)` will return positions 40 to 60.

**getUserRatingRanks(prior\_ranks:Number, num\_ranks:Number):Boolean**

Return the ranking list of users by rating value for the current game relative to the position of the current logged in user.

For example, if the current logged in user is ranked 50<sup>th</sup>, then calling `getUserRatingRanksForGame(0,10,20)` will return positions 40 to 60.

**getUserRatingRanksForGameForUser(game\_id:Number, user\_id:Number, prior\_ranks:Number, num\_ranks:Number):Boolean**

Similar to `getUserRatingRanksForGame()` only you can specify a particular user.

**getUserRatingRanksForUser(user\_id:Number, prior\_ranks:Number, num\_ranks:Number):Boolean**

Similar to `getUserRatingRanks()` only you can specify a particular user.

**getUserRating(gameId:Number):Boolean**

Return the rating for the indicated game for the current logged in user.

**getUserRatingForUser(userId:Number, gameId:Number):Boolean**

Return the rating for the indicated game for the specified user.

**getUserSiteExperiencePoints():Boolean**

Returns the value of the current logged in user's site-wide experience points.

**getUserSiteExperiencePointsForUser(user\_id:Number):Boolean**

Returns the value of the specified user's site-wide experience points.

**getUserGameGroupExperiencePointsForUser(user\_id:Number, game\_group\_id:Number):Boolean**

Returns the value of the specified user's experience points for the game group indicated.

**getUserGameGroupExperiencePoints(game\_group\_id:Number):Boolean**

Returns the value of the current logged in user's experience points for the game group indicated.

**getUserGameExperiencePointsForUser(user\_id:Number):Boolean**

Returns the value of the specified user's game experience points for the current game.

**getUserGameExperiencePoints():Boolean**

Returns the value of the current logged in user's game experience points for the current game.

**GSSendToFriend**

The send to friend functionality is separated out into it's own object because the service is a little different than standard game services and the call back operates in a different manner.

**GSSendToFriend ( callbackFn : Function )**

Constructor with optional call back function reference. You must set the call back function so if you do not do it here then use *setCallback()*.

**setCallback ( callbackFn : Function ) : Boolean**

Set the call back function reference. The call back function is called when an asynchronous function response arrives from the game services server. Use this function to set the call back function when not done with the constructor.

```
function stfCallback ( action:String ):Void
```

The action parameter can have one of the following values:

**nodata** – upon module initialization a request for game data is sent to the server. If there is no data this is the call back response.

**load** - upon module initialization a request for game data is sent to the server. This is the call back response when data is available.

**send** – when send is called this is the server response.

The call back function cannot be the same as the Game Services call back as the parameters are different.

**sendToFriend ( toName:String, toEmail:String, fromName:String, fromEmail:String, message:String, gameData:String ):Boolean**

Initiate the send to friend request. No parameter checking is done so we expect the caller to do the minimal validation of at least a valid email address for the *toEmail* and *fromEmail* parameters. *toName*, *fromName*, *message* and *gameData* are all optional.

To send to multiple email addresses separate each address in **toEmail** with semi-colon (;). In this case the **toName** parameter should be empty.

**getGameData():String**

If the .swf start up contained a request to reconstitute game specific data and the call back function was alerted that data has arrived, use this function to get the data. This situation is documented in Send To Friend Callback on page 13.

**getPlaybackId():String**

Returns the key to the game specific data.

**isError():Boolean**

Returns *true* if there was an error, *false* if the most recent operation was successful.

**getError():String**

If *isError()* returns true you can call this to get an error string describing the error.

## **GSRResult Object**

The GSRResult object contains information about the result of a service call. This object is sent to your callback function (see Callback Functions on page 7) upon completion of a service request from the GS servers.

**GSRResult ( node:XMLNode )**

Constructor. The XML Node describing a standard game services transaction must be supplied when constructing this object. Typically these objects are constructed internally and supplied via the call back function (see Callback Functions above).

**isError() : Boolean**

Returns true if the result object indicates the requested service failed. Check the *message* attribute or call *getErrorMessage()* to get more detailed information.

**getErrorMessage() : String**

Returns the full error code when *isError()* returns *true*. These error messages are usually not intended to be viewed by users.

Some notable errors:

`INVALID_XML` or `UNKNOWN_ERROR`: The server did not respond with XML. Chances are you are not connecting with a GS server.

`INVALID_RESPONSE_FORMAT`: The server responded with an XML document but it was not in the proper `GSResult` format. In this case usually the server is replying with a system error.

**getExtendedInfo () : String**

Returns additional information, possibly human readable, about the error code when *isError()* returns *true*.

**getFunctionRequest () : String**

Returns the requested GS service this result is a response to. This is useful when your application sends out multiple requests and you are asynchronously processing multiple `GSResult` objects. Also note that in some cases the function identifier returned here may not exactly match the API in your original request. Sometimes the SDK translates or consolidates certain functions.

**getStateSequence () : Number**

Returns the sequencing number assigned to the original request.

**getResults () : Array**

Returns a reference to the array of results. Most, but not all, service request result in an array of results.

**getOutputParams () : Object**

Returns a reference to the service output parameters. Certain services return scalar values in stead of an array of results.

## Properties

The GSResult object contains the following public properties you may need to query to get result information you require. These properties mirror the XML result sent back from the server as described in the section `getChallengeToken`

Returns the challenge token when the game is running in challenge mode. Maybe useful during testing but in production there isn't much you can do with the token.

`setChallengeToken`

Assign the challenge token, effectively putting the game in challenge mode. You would never do this in a production game. It is really only useful during development and testing.

XML Result Stream on page 19. These properties are accessible with the accessor functions described above. Future versions of the SDK may phase out direct access to the properties in this manner.

**outparams** – an associative array containing service output parameters. The items in this array will vary depending on the server call.

**passthru** – an object containing information about the call. Typically there are two items in this object.

*GSResult.passthru["fn"]* – contains the service that was called.

*GSResult.passthru["state\_seq"]* – contains a sequencing number. No two calls in the same session will have the same sequence number.

**results** – an array containing the result set. Contents of this object will depend on the service call.

## Appendix A – Document API Changes

Release Date	Version	Changes Made
September 11, 2007	1.5.4	Added Buddy API
September 28, 2007	1.5.5	Added Voting API
October 4, 2007	1.5.6	Added Notification API
November 4, 2007	1.5.7	Added Avatar API
November 16, 2007	1.5.8	Added findUser functions, getUserRating().
January 2, 2008	1.6.1	Added Challenges.
February 5, 2008	1.6.5	Updated Challenges.
February 18, 2008	1.6.6	Added GameData APIs for score submit. More complete documentation on Challenges. Added GSResult accessor methods.
February 10, 2009	3.1.3	Corrected leaderboard function description.