ARMOR GAMES

# Armor Games AGI

*Adventures in the Flash AGI, Version II*

Document version 1.0

# Table of Contents

agi.scoreboard.retrieveUserScores( params:Object ) : void
agi.scoreboard.retrieveFriendScores( params:Object ) : void

# Quick Start

The Armor Games Interface (AGI) version 2 is an AS3 based suite of services that provide the developer with tools to customize their game to the current user, sell in-game content, implement goals for the user to earn and store user and game data persistently and securely.

The following code illustrates the core elements that are needed to get the AGI-v2 up and running within a game. The AGI is included at runtime by loading an external SWF file during the games loading sequence. Once the AGI SWF is loaded the developer can then connect the current user's session with the remote services using the game API key.

To get started, visit developers.armorgames.com to create your game. You'll be given an API key that you can use to access AGI services. The Armor Games development site also offers a testing environment in which you can "play test" your game and inspect all the various AGI service calls that are being invoked while it is running. There are tools that allow you to easily test Quests, Storage and Premium Content purchasing to ensure that your game is interacting with the AGI properly.

Example:

```
/**********************************************************
 * Quick Setup
 **********************************************************/
import flash.display.*;
import flash.events.*;
import flash.net.*;
import flash.system.*;

var agi:*;
var agiURL:String = "http://agi.armorgames.com/assets/agi/AGI2.swf"
Security.allowDomain( "agi.armorgames.com" );
loader:Loader = new Loader();

// Setup error handler in case the AGI SWF fails to load
loader.contentLoaderInfo.addEventListener(
  IOErrorEvent.IO_ERROR,
  function (e:IOErrorEvent) : void {
    /**
     * Oh noez!
     * AG must be down or someone tripped on a cable
     * in a server room someplace, unplugging the
     * power to the entire internet.
     */
  }
);

// Setup success handler for AGI SWF
```

```
loader.contentLoaderInfo.addEventListener(
  Event.COMPLETE,
  function (e:Event) : void {
    // Connect current game session to remote services using games API key
    agi = e.currentTarget.content;
    agi.connect({
      stage:stage,
      apiKey:"[Put your API Key here]",
      callback:function ( data:Object ):void {
        if ( data.success ) {
          /**
           * All services offered by the AGI are now accessible.
           * You can use agi.isConnected() to check if the AGI is
           * connected to remote services.
           */
        } else {
          /**
           * The AGI failed to connect with the remote services.
           * This can happen when the user is offline or if
           * AG servers are having issues.
           * At this point you have an opportunity to fail gracefully.
           */
          trace( data.error );
        }
      }
    });
  }
);
loader.load( new URLRequest( agiURL ) );
```

# agi : Core AGI Methods

The core AGI object contains methods for configuring and checking the status of remote connections. These core methods, and all further methods, are available after the AGI has successfully loaded.

## agi.connect( params:Object ) : void

The connect method authenticates the given API key and creates a connection to the remote service. Connection settings are passed via the params object.

The properties for the params object are:

- stage (Stage)
- apiKey (String)
- callback (Function)

Example:

```
agi.connect({
  stage:    stage,
  apiKey:   "45EB8AD2-2A10-4BF6-9F32-E04F01C76083",
  callback: function ( data:Object ) : void {
      if ( data.success ) {
        // AGI connected successfully. AGI services are now available
      } else {
        // AGI failed to connect. This occurs when the user is offline or if
        // there are remote service issues. You have an opportunity to handle the
        // condition gracefully.
        trace( data.error );
      }
    }
  }
});
```

## agi.isConnected( void ) : Boolean

The isConnected method will return true if the AGI has successfully authenticated your API key and has connected to remote services.

Example:

```
if ( agi.isConnected() ) {
  // AGI is connected and AGI services are available
} else {
  // AGI is not connected
}
```

# agi.user : User Methods

Methods for accessing information about the current user.

## agi.user.isGuest(void) : Boolean

Returns true if the user is not currently logged in.

Example:

```
if ( agi.user.isGuest() ) {
  // Current user is not logged in
} else {
  // Current user is logged in and may be a standard member, moderator or admin
}
```

## agi.user.isPremium(void) : Boolean

Returns true if the user has purched Armor Games website premium service (or currently known as Ad-Free+ Gaming)

Example:

```
if ( agi.user.isPremium() ) {
  // User is a Premium AG Member!
} else {
  // Current user is not logged in or not a premium AG member
}
```

## agi.user.isModerator(void) : Boolean

Returns true if the user is logged in and they are an Armor Games moderator.

Example:

```
if ( agi.user.isModerator() ) {
  // Current user is either not logged in or is logged in but not a moderator
} else {
  // Current user is logged in and is a moderator
}
```

## agi.user.isAdmin(void) : Boolean

Returns true if the user is logged in and they are an Armor Games administrator.

Example:

```
if ( agi.user.isAdmin() ) {
  // Current is either not logged in or is logged in but not an administrator
} else {
  // Current user is logged in and is an administrator
}
```

## agi.user.getUID(void) : String

A consistent and unique ID for the current user. The value is a 32 character string (letters and numbers) or NULL if the current user is not logged in.

Example:

```
var userID:String = agi.user.getUID();
  // userID = 'd3b07384d113edec49eaa6238ad5ff00' OR NULL
```

## agi.user.getUsername(void) : String

The current user's username on Armor Games or NULL if they are not logged in.

Example:

```
var username:String = agi.user.getUsername();
  // username = 'ferret' OR NULL
```

## agi.user.getAvatarURL(void) : String

A fully qualified URL to the user's avatar image (can be a JPEG, GIF or PNG). The size is 50 x 50 pixels.

Example:

```
var avatarURL:String = agi.user.getAvatarURL();
  // avatarURL = 'http://cache.armorgames.com/image/armatar_426_50.50_c.png' OR NULL
```

# agi.content : Premium Content Methods

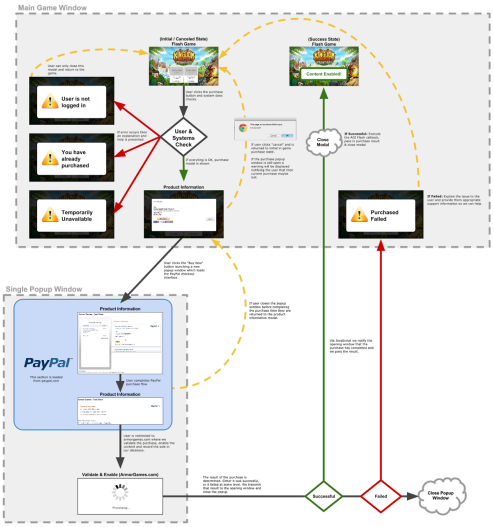
View Full Size

The Premium Content service provides the tools needed for a game developer to sell products and verify purchases within their games. Armor Games currently supports two product types: *Unlockables* and *Consumables*.

**Unlockable products** - player buys item once, can access forever.

**Consumable products** - player buys finite quantity which are consumed during gameplay.

Premium Content v2
**Standard Purchase Flow**




The Armor Games team does the initial product setup for developers. This includes creating new products, associating marketing assets, and setting prices. Once setup, developers can use the following methods to interact with the Premium Content service.

To get started selling products within your game, please contact tasselfoot@armorgames.com.

## agi.content.showStore(params : Object) : void

Displays the HTML Storefront modal window atop the game. This window displays product information and purchase options to the user. The user can choose to purchase a product or simply close the Storefront window.

The developer has basic control over what is displayed in the HTML Storefront modal window depending on the params object provided. For example, if a product SKU is provided then the Storefront will only list the product for the given SKU. If the SKU is omitted, then all active products for the game will be displayed.

The properties for the params object are:

- sku (String) *Optional*
- callback (Function)

After the user finishes interacting with the HTML Storefront modal window, the developer provided callback will be triggered. This callback function is given an object as its only parameter. This object represents the results of the users interactions with the storefront. By testing the values of the response data, the developer can determine the next steps to take within the game. The structure of the response object is:

- success (Boolean) *If the call succeeded or not*
- response (String) *The action the user took*
- purchases (Array) *An array of purchased objects*
    - sku (String)
    - name (String)
    - type (String)
    - quantity (Int)
    - price (Number)
    - totalSales (Int)

Example:

```
agi.content.showStore({
  sku: "cca-siege_weapons", // OPTIONAL
  callback: function( data:Object ):void {
    if ( data.success ) {
      switch ( data.response ) {
        case agi.content.RESPONSE_USER_CANCELLED:
          // User cancelled
          break;
        case agi.content.RESPONSE_PURCHASE_FAILED:
          // The purchase failed or errored out. The user should have been notified
          // within the storefront modal, but it's good to handle the failure
          // here as well.
          break;
        case agi.content.RESPONSE_PURCHASE_SUCCESS:
          // The user has successfully purchased a product.
          for (var i:int = 0; i < data.purchases.length; i++) {
            trace( "User Purchased:", data.purchases[i].name );
          }
          break;
      }
    } else {
      // The call failed to contact Armor Games.
      // You have the opportunity to fail gracefully.
      trace( data.error );
    }
  }
});
```

## agi.content.retrievePurchases(params : Object) : void

The retrievePurchases method fetches all the purchases for the current user and returns an array of data to the provided callback. If a SKU is provided, then it returns purchase information for the given SKU only (if any).

The properties for the params object are:

- sku (String) *Optional*
- callback (Function)

The structure of the response object is:

- success (Boolean) *If the call succeeded or not*
- error (String) *Error message, if any*
- purchases (Array) *An array of purchased objects*
    - sku (String) *Product SKU*
    - name (String) *Product name*
    - description (String) *Product description*
    - type (String) *Either 'consumable' or 'unlockable'*
    - quantity (Int) *Consumable only. Represents remaining product uses for the user*

Example:

```
agi.content.retrievePurchases({
  sku: "cca-siege_weapons", // OPTIONAL
  callback: function( data:Object ):void {
    if ( data.success ) {
      // You may now cycle through the current users purchased content
      for (var i:int = 0; i < data.purchases.length; i++) {
        trace( "User Purchased:", data.purchases[i].name );
      }
    } else {
      // The call failed to contact Armor Games.
      // You have the opportunity to fail gracefully.
      trace( data.error );
    }
  }
});
```

## agi.content.retrieveProducts(params : Object) : void

The retrieveProducts method will fetch all active products for the current game/API key and return an array of data to the provided callback. If a SKU is provided then it returns product information for the given SKU only.

The properties for the params object are:

- sku (String) *Optional*
- callback (Function)

The structure of the response object is:

- success (Boolean) *If the call succeeded or not*
- error (String) *Error message, if any*
- products (Array) *An array of product objects*
    - sku (String) *Product SKU*
    - name (String) *Product name*
    - description (String) *Product description*
    - type (String) *Either 'consumable' or 'unlockable'*
    - price (Number) *Current purchase price in USD*
    - totalSales (Int) *Total number of sales for this product*

Example:

```
agi.content.retrieveProducts({
  sku: "cca-siege_weapons", // OPTIONAL
  callback: function( data:Object ):void {
    if ( data.success ) {
      // You may now cycle through the current products
      for (var i:int = 0; i < data.products.length; i++) {
        trace( "User Purchased:", data.products[i].name );
      }
    } else {
      // The call failed to contact Armor Games.
      // You have the opportunity to fail gracefully.
      trace( data.error );
    }
  }
});
```

## agi.content.consume(params : Object) : void

The consume method will decrease a user's item quantity for the given product SKU. It is only valid for consumable products and will error if called using a SKU for any other type of product. By default the quantity will be decreased by 1. However, an amount property can be passed via the params object to decrease quantity by a greater amount.

The properties for the params object are:

- sku (String)
- amount (Int) *Optional*
- callback (Function)

The structure of the response object is:

- success (Boolean) *If the call succeeded or not*
- error (String) *Error message, if any*
- consumables (Array) *An array of consumable product objects*
    - sku (String) *Product SKU*
    - name (String) *Product name*
    - description (String) *Product description*
    - quantity (Int) *Remaining product uses for the user*

Example:

```
agi.content.consume({
  sku: "cca-siege_weapons",
  amount: 1, // OPTIONAL
  callback: function( data:Object ):void {
    if ( data.success ) {
      // You may now cycle through the consumable products
      for (var i:int = 0; i < data.consumables.length; i++) {
        trace(
          "Consumable Product:", data.consumables[i].name,
          "Remaining Uses:", data.consumables[i].quantity
        );
      }
    } else {
      // The call failed to contact Armor Games.
      // You have the opportunity to fail gracefully.
      trace( data.error );
    }
  }
});
```

# agi.quests : Quest Methods

The Quest service provides developers with the tools to create objectives within their games that users can earn and showcase on their Armor Games profile page. Quests are identified by developers with a unique key value. The Quest service provides methods to update the progress for these given keys.

Quests are configured within the system with a “starting value” and “target value”. For simple Quests, where a user needs to perform an action one time, the starting value would be 0 and the target value would be 1. Calling the submit method in this case would grant the user the Quest immediately.

Quests also have the notion of progress. For example, consider a game with a quest in which a user needs to run 100 meters total. The starting value for this quest can be 0 and the target value could be 10. For every 10 meters run the game, we can increment the progress by 1 unit. If the user leaves the game before earning this quest we are able to display their quest progress throughout the site and encourage them to return to the game and earn it.

### Avoiding Inadvertent Service Abuse

In the previous example why did we not set the target value of the quest to 100 if the goal was to run 100 meters? The reason is that for every quest method called a remote service request must be made. Depending on the frequency of progress updates for a given quest, handling a high frequency of service calls for every user of the game concurrently will degrade performance of the service for all users. There is a balance between how detailed the Quest progress must be versus how often the service needs to be updated to record the progress information. The testing environment within the developers site on Armor Games has the tools you will need to understand the frequency of service calls and the Armor Games team is always available to offer advice and troubleshooting.

To get started using quests within your game please contact tasselfoot@armorgames.com.

## agi.quests.submit(params : Object) : void

The submit method for the Quest service is used to update the current users progress for a given Quest key.

The properties for the params object are:

- key (String)
- progress (Number/Float) *Optional. Treat as progress percentage: a value between 0-1*
- callback (Function)

The method can be used in one of two ways depending on the params object configuration. First, if the progress property is set then the progress value for the user will be updated

accordingly. The progress values is treated as a percentage where the valid range is 0 (0%) and 1 (100%). For example if a user completed half the progress for a given Quest then the progress value would be .5 (50%).

The second approach omits the progress property and just submits a Quest key and callback method. In this case the progress for the current user is incremented by 1 unit. If the target value for a Quest is 1 then calling the submit method without the progress property would immediately grant the quest. If target value for a Quest was 5, then the first call to the submit method for a users would increase the progress value to 1, and the Quest is considered 20% complete. This approach allows the developer to easily record progress without having to worry about maintaining the state of the quest. For example if a Quest requires a user to collect any 5 coins regardless of any specific level or game play session then this approach to Quest submission would work for that by tracking the count for you and immediately award the quest once the target value is reached.

Regardless of the which approach is used to call the submit method, the same data object will be returned to the callback.

The structure of the response object is:

- success (Boolean) *If the call succeeded or not*
- quest (Object) *An array of consumable product objects*
    - key (String) *Quest key*
    - progress (Number) *Number between Quest's "Start Value" & "Target Value"*
    - status (String) "*In Progress", "Completed" or "Revoked"*

### Note: Users Are Prevented From Losing or Going Backwards in Quest Progress

When submitting the progress property please note that the service will not update a user's progress for a quest if its new value is less than its current progress value. This prevents issues where local storage is used for saving game state and users switch machines, causing their data to reset and lower progress values to be reported (another way to avoid this problem is to use the AGI Storage service for saving user's game state information). There are times when you may actually want to reset the progress value for a user or set it to a lower number. In this case you would use the reset method to set the current users progress value to the quests start value. Then use the submit method to set it to the appropriate value.

### Note: Quests Can Only Be Earned Once And Never Taken Away

Once a user has earned a Quest it is theirs forever and only an Armor Games Administrator can update it. After a quest is earned, any submit or rest calls will have no affect on that particular Quests progress value (it will stay at 100%).

Example:

```
agi.quests.submit({
  key: "killed_10_enemies",
  progress: .7, // OPTIONAL
  callback: function( data:Object ):void {
    if ( data.success ) {
      // You may now access updated Quest data
      trace( "Users Current Progress:", data.quest.progress );
      trace( "User Has Earned:", data.quest.status.toLowerCase()=="completed" );
    } else {
      // The call failed to contact Armor Games.
      // You have the opportunity to fail gracefully.
      trace( data.error );
    }
  }
});
```

## agi.quests.reset(params : Object) : void

The reset method resets the users Quest progress to its starting value.  This method can not revoke an achievement a user has already earned, it merely resets the value on the server so that that the current progress is 0%.

The properties for the params object are:

- key (String)
- callback (Function)

The structure of the response object is:

- success (Boolean) *If the call succeeded or not*
- quest (Object) *An array of consumable product objects*
    - key (String) *Quest key*
    - progress (Number) *Number between Quest's "Start Value" & "Target Value"*
    - status (String) "*In Progress", "Completed" or "Revoked"*

Example:

```
agi.quests.reset({
  key: "killed_10_enemies",
  callback: function( data:Object ):void {
    if ( data.success ) {
      // This property should now be 0 (or whatever is the start value for this quest)
      trace( "Users Current Progress:", data.quest.progress );
```

```
      // This property can not be altered by this method
      trace( "User Has Earned:", data.quest.status.toLowerCase()=="completed" );
    } else {
      // The call failed to contact Armor Games.
      // You have the opportunity to fail gracefully.
      trace( data.error );
    }
  }
});
```

# agi.storage : Storage Methods

The storage service (aka GameSave) provides developers the ability to store and retrieve game data. The value to users is that their game progress is consistent across different browsers and computers. The value to developers is that there is no storage limit, data is encrypted and much more difficult to hack, and game data can be tied to a single user or stored in a public game space where all current users can access.

The storage service is divided into two parts. Each part has a different namespace but share the same method calls:

- agi.storage.user (Private) *Data stored and viewable by the current user only*
- agi.storage.game (Public) *Data can be stored and viewed by any game user*

The storage system uses a key/value model. You associate a key (String) with a value (String, Number, int, Boolean, Null, Array, Object). You can then read update or delete the value based on the given key string.

## agi.storage.[user|game].submit( params:Object ) : void

Stores or updates the value for the given key.

The properties for the params object are:

- key (String)
- value (*)
- callback (Function)

The structure of the response object is:

- success (Boolean) *If the call succeeded or not*
- error (String) *Error message, if any*
- key (String) *Key specified in the params object*

Example:

```
// agi.storage.game.submit({...});
agi.storage.user.submit({
  key: "stats_level_1",
  value: {time:532,kills:27,completed:true},
  callback: function( data:Object ):void {
    if ( data.success ) {
      // Data was successfully saved
    } else {
      // The call failed to contact Armor Games.
      // You have the opportunity to fail gracefully.
```

```
      trace( data.error );
    }
  }
});
```

## agi.storage.[user|game].retrieve( params:Object ) : void

Retrieves the value for a given key. If the key is omitted then the method will return all key/values stored for the current user (limited to 100 key/values).

The properties for the params object are:

- key (String) *Optional*
- callback (Function)

The structure of the response object is:

- success (Boolean) *If the call succeeded or not*
- error (String) *Error message, if any*
- keys (Object) *An associative array of keys to values*
    - .key_name_1 (String) = Value (*)
    - .key_name_2 (String) = Value (*)

Example:

```
// agi.storage.game.retrieve({...});
agi.storage.user.retrieve({
  key: "stats_level_1",
  callback: function( data:Object ):void {
    if ( data.success ) {
      // You can now access the keys value
      trace( "Key Value: " + data.keys.stats_level_1 );
    } else {
      // The call failed to contact Armor Games.
      // You have the opportunity to fail gracefully.
      trace( data.error );
    }
  }
});
```

## agi.storage.[user|game].erase( params:Object ) : void

Removes a the given key and its associated value.

The properties for the params object are:

- key (String)
- callback (Function)

The structure of the response object is:

- success (Boolean) *If the call succeeded or not*
- error (String) *Error message, if any*
- key (String) *Key specified in the params object*

Example:

```
// agi.storage.game.erase({...});
agi.storage.user.erase({
  key: "stats_level_1",
  callback: function( data:Object ):void {
    if ( data.success ) {
      // Data was successfully removed
    } else {
      // The call failed to contact Armor Games.
      // You have the opportunity to fail gracefully.
      trace( data.error );
    }
  }
});
```

# agi.gameshare : Game Share / Player Created Content & Levels

The GameShare service allows players to save and share content created within the developer's game, which is then voted upon by other players. Some examples of its uses are:

- Platformer or Puzzle game where players build their own levels using a level editor
- Racing game that saves a players track run and then allows other players to race against his 'ghost'
- Obstacle Course game where players can build their own courses
- Multiplayer games can use gameshares as 'lobbies', where an individual gameshare acts as a room

Developers can retrieve gameshares with a variety of grouping options -- by a timeframe, sorting by different fields, for a specific user, or that user's friends, or by how popular a gameshare is. Individual gameshares do not have a set size for their data storage, but note that the bigger the amount of data, the slower the service will be. All data stored in a gameshare is encrypted.

Each gameshare may have a thumbnail associated with it as well, which will be shown to user as he browses through the gameshares for your game. (note: as of this writing there is no gameshare UI or gameshare section of the site on Armor Games, but it is planned)

Each gameshare has a 'Share ID' that identifies it from the rest. You may use this ID when rating or updating a gameshare.

The data stored in a gameshare has to be a string, so if you have an object you wish to store in gameshare, you will need to serialize it first.

Note that each constant below, designated in all caps, comes from agi.gameshare, for instance: agi.gameshare.ORDER_DESCENDING

As part of a retrieve, you get a "totalSharesCount" variable back as part of the object. This is useful for showing how many pages are available. This number may change depending on the request type -- for instance, if you specify a different "option", or a different "timeFrame", you will get different total counts.

## agi.gameshare.submit( params:Object ) : void

Creates a new gameshare.

The properties for the params object are:

- name (String)
- data (String)
- callback (Function)
- thumbnail (DisplayObject, optional but highly recommended)

- description (String, optional)

The structure of the response object is:

- success (Boolean) *If the call succeeded or not*
- error (String) *Error message, if any*
- shareID (String) *Unique Key/Identifier for this gameshare*
- longURL (String) *The Long URL for this gameshare*
- shortURL (String) *The Short URL for this gameshare (recommended use)*
- thumbnailURL (String) *The Thumbnail URL, if thumbnail data was provided*
- createdDate (String) *Date in format: 2014-02-27 10:25:04*

Example:

```
agi.gameshare.submit({
  name: "User typed in this name!",
  description: "Short description written by the user",
  thumbnail: <<a provided DisplayObject>>,
  data: <<a serialized data string>>,
  callback: function( data:Object ):void {
    if ( data.success ) {
      // Data was successfully saved
    } else {
      // The call failed to contact Armor Games.
      // You have the opportunity to fail gracefully.
      trace( data.error );
    }
  }
});
```

## agi.gameshare.update( params:Object ) : void

Updates a gameshare, for which you will need to provide the shareID.  You can update the name, data, thumbnail, or description.

The properties for the params object are:

- name (String)
- data (String)
- callback (Function)
- thumbnail (DisplayObject, optional but highly recommended)
- description (String, optional)

The structure of the response object is:

- success (Boolean) *If the call succeeded or not*
- error (String) *Error message, if any*
- shareID (String) *Unique Key/Identifier for this gameshare*
- longURL (String) *The Long URL for this gameshare*
- shortURL (String) *The Short URL for this gameshare (recommended use)*
- thumbnailURL (String) *The Thumbnail URL, if thumbnail data was provided*
- createdDate (String) *Date in format: 2014-02-27 10:25:04*

Example:

```
agi.gameshare.update({
  shareID: "92f748ac4b40934622980a36d2910773",
  name: "User typed in this name!",
  description: "Short description written by the user",
  thumbnail: <<a provided DisplayObject>>,
  data: <<a serialized data string>>,
  callback: function( data:Object ):void {
    if ( data.success ) {
      // Data was successfully saved
    } else {
      // The call failed to contact Armor Games.
      // You have the opportunity to fail gracefully.
      trace( data.error );
    }
  }
});
```

## agi.gameshare.retrieve( params:Object ) : void

Retrieve a list of gameshares, or a specific gameshare. You will only get the data associated with a gameshare if you request it specifically via the ShareID. You can get the ShareID in two ways: a user visited a the page via a gameshare link and you call the agi.getGameShareID() function, or when you retrieve a list of gameshares.

The properties for the params object are:

- callback (Function)
- shareID (String) *NOTE: if you supply shareID, the rest of these options are ignored*
- timeFrame (String, default TIME_FRAME_ALLTIME)
- sortOrder (String, default ORDER_DESCENDING)
- sortBy (String, default BY_DATE)
- pageNum (Number, default 1)
- sharesPerPage (Number, default 32)
- option (String, optional)
- random (Boolean, optional) *Return a random gameshare*

Property constants, which you may get via agi.gameshare.<constant>:
timeFrame - TIME_FRAME_ALLTIME, TIME_FRAME_WEEKLY, TIME_FRAME_DAILY
sortOrder - ORDER_DESCENDING, ORDER_ASCENDING
sortBy - BY_DATE, BY_RATING, BY_VIEWS
option:

- OPTION_USER - all gameshares for the provided UID
- OPTION_FRIENDS - all gameshares for the provided UID's friends
- OPTION_LIKED - all gameshares the UID liked
- OPTION_FRIENDS_LIKED - all gameshares the UID's friends liked
- OPTION_RATINGS - each gameshare has a 'rating' which is how the user rated the gameshare, which may be: none, liked, disliked.

The structure of the response object is an array of objects with these values:

- success (Boolean) *If the call succeeded or not*
- error (String) *Error message, if any*
- sortOrder (String) *Same as ORDER_DESCENDING or ORDER_ASCENDING*
- sharesPerPage (Number) *How many shares are being returned per page*
- pageNum (Number) *Current page number*
- totalSharesCount (Number) *Total number of gameshares for this request type*
- gameshares (Object array) Note: if requesting a single gameshare, this is still an array
    - shareID (String) *Unique Key/Identifier for this gameshare*
    - name (String) *User provided name of gameshare*
    - description (String) *User provided description of gameshare*
    - data (String) *Data of gameshare (ONLY if ShareID was provided)*
    - longURL (String) *The Long URL for this gameshare*
    - shortURL (String) *The Short URL for this gameshare (recommended use)*
    - thumbnailURL (String) *The Thumbnail URL, if thumbnail data was provided*
    - uid (String) *Creator UID*
    - username (String) *Username of the creator*
    - rating (String|Null) *How the user rated the gameshare. only populated with option OPTION_RATINGS. if null, then you did not specify the option*
    - profileURL (String) *Profile URL of the creator*
    - avatarURL (String) *User's chosen Avatar in a default size (Tiny)*
    - avatarSizes (Object array) *Different sizes of the player's avatar*
        - avatarURL (String)
        - tiny (String)
        - small (String)
        - medium (String)
        - large (String)
    - stats (Object:) *Different statistics about the gameshare*
        - likes_daily (Number)
        - likes_weekly (Number)

        - likes_monthly (Number)
        - likes_alltime (Number)
        - dislikes_daily (Number)
        - dislikes_weekly (Number)
        - dislikes_monthly (Number)
        - dislikes_alltime (Number)
        - views_daily (Number)
        - views_weekly (Number)
        - views_monthly (Number)
        - views_alltime (Number)
    - createdDate (String) *Date in format: 2014-02-27 10:25:04*
    - updatedDate (String) *Date in format: 2014-02-27 10:25:04*
    - createdRelativeTime (String) *In relative time, when this was created, ie "3 weeks ago"*
    - updatedRelativeTime (String) *In relative time, when this was updated, ie "3 weeks ago"*

Example:

```
// Get a list of gameshares created by the player in the last week sorted
// ascending by their rating
agi.gameshare.retrieve({
  timeFrame: agi.gameshare.TIME_FRAME_WEEKLY,
  sortOrder: agi.gameshare.SORT_ASCENDING,
  sortBy: agi.gameshare.BY_RATING,
  option: agi.gameshare.OPTION_USER,
  uid: agi.user.getUID(),
  callback: function( data:Object ):void {
    if ( data.success ) {
      // Data was successfully loaded
    } else {
      // The call failed to contact Armor Games.
      // You have the opportunity to fail gracefully.
      trace( data.error );
    }
  }
});

// Load a single gameshare
agi.gameshare.retrieve({
  shareID: "some share ID",
  callback: function( data:Object ):void {
    if ( data.success ) {
      // Data was successfully loaded
    } else {
      // The call failed to contact Armor Games.
      // You have the opportunity to fail gracefully.
```

```
      trace( data.error );
    }
  }
});

// Check if the page was loaded via a GameShare URL and load it
if(agi.isGameShareLoad()) {
  agi.gameshare.retrieve({
    shareID: agi.getGameShareID(),
    callback: function( data:Object ):void {
      if ( data.success ) {
        // Data was successfully loaded
      } else {
        // The call failed to contact Armor Games.
        // You have the opportunity to fail gracefully.
        trace( data.error );
      }
    }
  });
}
```

## agi.gameshare.random( params:Object ) : void

Get a random gameshare for the game.

The properties for the params object are:

- callback (Function)

The structure of the response object are exactly the same as retrieve() above.

Example:

```
agi.gameshare.random({
  callback: function( data:Object ):void {
    if ( data.success ) {
      // Data was successfully saved
    } else {
      // The call failed to contact Armor Games.
      // You have the opportunity to fail gracefully.
      trace( data.error );
    }
  }
});
```

## agi.gameshare.rate( params:Object ) : void

Rate a gameshare.  A player may like, dislike, or remove their rating of a gameshare.

The properties for the params object are:

- callback (Function)
- rating (String) like, dislike, remove

The structure of the response object is an array of objects with these values:

- success (Boolean) *If the call succeeded or not*
- error (String) *Error message, if any*

Example:

```
// Rate a gameshare
agi.gameshare.rate({
  shareID: "92f748ac4b40934622980a36d2910773",
  rating: "like",
  callback: function( data:Object ):void {
    if ( data.success ) {
      // Data was successfully saved
    } else {
      // The call failed to contact Armor Games.
      // You have the opportunity to fail gracefully.
      trace( data.error );
    }
  }
});
```

# agi.scoreboard : Scoreboards / High Scores / Leaderboards

Scoreboards allow users to submit scores to compete against each other.  A game may have any number of specific scoreboards, identified by a name.

A score may be floating point, and may range from -9999999999999.9999 to 9999999999999.9999.  When you submit a score, you will specify a scoreboardName.  If the scoreboard doesn't exist yet, it will automatically be created.

You may retrieve scores in ascending or descending order, and by any of the standard AGI timeframes: daily, weekly, alltime.  You may sort the scores by date, or (the default) by score. There are a few calls to make for a scoreboard -- retrieve all the public scores for the scoreboard, retrieve the scores for the scoreboard compared to your Armor Games friends, or retrieve your personal scores for the scoreboard.

Note that each constant below, designated in all caps, comes from agi.scoreboard, for instance: agi.scoreboard.ORDER_DESCENDING

As part of a retrieve, you get a "totalScoresCount" variable back as part of the object.  This is useful for showing how many pages are available.

## agi.scoreboard.submit( params:Object ) : void

Adds a score to a scoreboard.

The properties for the params object are:

- scoreboardName (String)
- score (Number)
- callback (Function)

The structure of the response object is:

- success (Boolean) *If the call succeeded or not*
- prevScore (Number) *User's previous score, if any*
- prevScoreTime (Number) *Unix timestamp of previous score*
- prevScoreRelativeTime (String) *Previous score in a readable format, ie. "3 days ago"*

Example:

```
agi.scoreboard.submit({
  scoreboardName: "easy",
  score: 111,
  callback: function( data:Object ):void {
    if ( data.success ) {
      // Data was successfully saved
    } else {
      // The call failed to contact Armor Games.
      // You have the opportunity to fail gracefully.
      trace( data.error );
    }
  }
});
```

## agi.scoreboard.retrievePublicScores( params:Object ) : void

Retrieve a list of public scores for a scoreboard.

The properties for the params object are:

- timeFrame (String) agi.scoreboard.*TIME_FRAME_[DAILY | WEEKLY | ALL_TIME]*
- scoreboardName (String)
- sortOrder (String) *agi.scoreboard.[ORDER_DESCENDING | ORDER_ASCENDING]*
- sortBy (String) *agi.scoreboard.[BY_SCORES | BY_DATE]*
- pageNum (Number)
- scoresPerPage (Number)
- callback (Function)

The structure of the response object is:

- success (Boolean) *If the call succeeded or not*
- timeFrame (String) *alltime, weekly, monthly*
- sortOrder (String) *descending, ascending*
- totalScoresCount (Number) *How many scores exist for this scoreboard & timeFrame*
- averageScore (Number) *Average score for this scoreboard & timeFrame*
- pageNum (Number) *Current page number*
- scoresPerPage (Number) *How many scores are listed per page*
- rank (Number) *Users rank on the scoreboard*
- scores (Array of Objects)
    - score (Number)
    - time (Number) *Unix timestamp of created date*
    - relativeTime (String) *Readable format of created date, ie: "3 days ago"*
    - username (String) *Username of who achieved this score*
    - avatarURL (String) *User's avatar URL*

    - avatarSizes (Object) *Various urls for the user's avatar*
        - tiny (String)
        - medium (String)
        - large (String)
    - profileURL (String) *Profile URL for user*

Example:

```
agi.scoreboard.retrievePublicScores({
  scoreboardName: "easy",
  timeFrame: agi.scoreboard.TIME_FRAME_ALL_TIME,
  callback: function( data:Object ):void {
    if ( data.success ) {
      // Data was successfully saved
    } else {
      // The call failed to contact Armor Games.
      // You have the opportunity to fail gracefully.
      trace( data.error );
    }
  }
});
```

## agi.scoreboard.retrieveUserScores( params:Object ) : void

Retrieve a list of public scores a user has submitted to a scoreboard.

The properties for the params object are:

- scoreboardName (String)
- sortOrder (String) *agi.scoreboard.[ORDER_DESCENDING | ORDER_ASCENDING]*
- sortBy (String) *agi.scoreboard.[BY_SCORES | BY_DATE]*
- pageNum (Number)
- scoresPerPage (Number)
- callback (Function)

The structure of the response object is:

- success (Boolean) *If the call succeeded or not*
- avatarURL (String) *User's avatar URL*
- avatarSizes (Object) *Various urls for the user's avatar*
    - tiny (String)
    - medium (String)
    - large (String)
- profileURL (String) *Profile URL for user*
- sortOrder (String) *descending, ascending*

- totalScoresCount (Number) *How many scores exist for this scoreboard & timeFrame*
- averageScore (Number) *Average score for this scoreboard & timeFrame*
- pageNum (Number) *Current page number*
- scoresPerPage (Number) *How many scores are listed per page*
- topScore (Number) *Users top score for this scoreboard*
- scores (Array of Objects)
    - score (Number)
    - time (Number) *Unix timestamp of created date*
    - relativeTime (String) *Readable format of created date, ie: "3 days ago"*
- ranks (Array of Objects) *Indexed by the different TIME_FRAMEs*
    - playerCount (Number) *Number of players in this time frame*
    - rank (Number) *Player's rank for this time frame*
    - score (Number) *Player's top score for this time frame*

Example:

```
agi.scoreboard.retrieveUserScores({
  scoreboardName: "easy",
  sortBy: agi.scoreboard.BY_DATE,
  callback: function( data:Object ):void {
    if ( data.success ) {
      // Data was successfully saved
    } else {
      // The call failed to contact Armor Games.
      // You have the opportunity to fail gracefully.
      trace( data.error );
    }
  }
});
```

## agi.scoreboard.retrieveFriendScores( params:Object ) : void

Retrieve a list of scores a user's friends has submitted to a scoreboard and rank them.  The list includes the user as well.

The properties for the params object are:

- scoreboardName (String)
- sortOrder (String) *agi.scoreboard.[ORDER_DESCENDING | ORDER_ASCENDING]*
- sortBy (String) *agi.scoreboard.[BY_SCORES | BY_DATE]*
- pageNum (Number)
- scoresPerPage (Number)
- callback (Function)

The structure of the response object is:

- success (Boolean) *If the call succeeded or not*
- sortOrder (String) *descending, ascending*
- totalScoresCount (Number) *How many scores exist for this scoreboard & timeFrame*
- averageScore (Number) *Average score for this scoreboard & timeFrame*
- pageNum (Number) *Current page number*
- scoresPerPage (Number) *How many scores are listed per page*
- scores (Array of Objects)
    - score (Number)
    - time (Number) *Unix timestamp of created date*
    - relativeTime (String) *Readable format of created date, ie: "3 days ago"*
    - username (String) *Username of who achieved this score*
    - profileURL (String) *Profile URL for user*
    - avatarURL (String) *User's avatar URL*
    - avatarSizes (Object) *Various urls for the user's avatar*
        - tiny (String)
        - medium (String)
        - large (String)

Example:

```
agi.scoreboard.retrieveFriendScores({
  scoreboardName: "easy",
  sortOrder: agi.scoreboard.SORT_ASCENDING,
  callback: function( data:Object ):void {
    if ( data.success ) {
      // Data was successfully saved
    } else {
      // The call failed to contact Armor Games.
      // You have the opportunity to fail gracefully.
      trace( data.error );
    }
  }
});
```