The External API

Lassie Player ActionScript Interface

The Lassie Player is built with a collection of external ActionScript classes that are available for third-party Flash media to implement and utilize. Using this library, third-party Flash media can establish a direct link of communication with the Lassie Player using Flash-native ActionScript. These resources allow custom third-party media that is programmed using Flash ActionScript to access and control the Lassie Player. For example, while developing a custom mini-game a developer can implement Lassie's external interface, then use the ActionScript controls to have the mini-game adjust configuration within the main Lassie Player environment. The mini-game Flash media can then be placed as the sole layer within a room layout, which will make the mini game available within Lassie through standard room layout integration.

The external ActionScript API is an extremely powerful tool for interfacing Flash media with the Lassie Player. However, the system does require some preexisting technical knowledge of using ActionScript in order to implement. A developer should be comfortable with implementing and extending ActionScript classes from an external library before attempting to integrate the external API with their project.

For complete, up-to-date documentation on the Lassie Player External API, see the ActionScript interface at **com.lassie.external.ILassiePlayer**.

The Interface Methods

Methods

The following is a list of methods available through the Lassie Player external API.

processXML

```
/**
* Processes an XML script.
* @param xml  The XML script to process.
* @param branch  Specifies if the XML should be run as a branch process.
* When false, the script runs as the trunk process which will overwrite any existing queue of trunk actions.
* When true, the script runs as a branch independent of the trunk.
*/
function processXML($xml:XML, $processAsBranch:Boolean=false):void;
```

---

newGame

```
/**
* Starts a new game within the player.
* @param index  The save slot index (zero-based) in which to create the new game.
* @param name  The name to assign to the saved game.
*/
function newGame($index:uint, $name:String):void;
```

---

loadGame

```
/**
* Loads a saved game.
* @param index  The save slot index from which to load a saved game.
* @param name  The name to assign to the newly loaded game.
* Generally this name should correspond to the saved game's existing name.
*/
function loadGame($index:uint, $name:String):void;
```

---

saveGame

```
/**
* Saves the currently active game.
* @param index  The save slot index in which to save the current game data.
* @param name  The name to assign to the game data.
* Generally this name should correspond to whatever name the game was loaded with.
```

*/
function saveGame($index:uint, $name:String):void;

---

clearGame

/**
* Clears data from a saved game slot.
* @param index The save slot index to purge data from.
*/
function clearGame($index:uint):void;

---

importGame

/**
* Loads a new game using raw data imported from a foreign source.
* @param data The data to import. This data should originally have been acquired through export.
* @param index The save slot index in which to load the game.
* @param name The name to assign to the newly loaded game's save index.
*/
function importGame($data:Object, $index:uint=0, $name:String=""):void;

---

exportGame

/**
* Exports all current data within the game cache.
*/
function exportGame():Object;

---

hasCacheVar

/**
* Tests if a field exists within the game cache.
* @param field The name of the field (variable) to test for.
* @return Returns true if the field exists.
*/
function hasCacheVar($field:String):Boolean;

---

getCacheVar

/**
* Gets the value of a field within the game cache.
* @param field The name of the field to pull data from.
* @return Returns the named field value as a string, or null if the field does not exist.

*/
function getCacheVar($field:String):String;

---

setCacheVar

/**
* Sets a value field within the cache, or clears a field when set to empty.
* @param field Name of the field to write data to.
* @param value The value to write into the field. Write an empty string ("") to clear a field from the cache.
*/
function setCacheVar($field:String, $value:String):void;

---

logicEval

/**
* Evaluates an expression using the Logic API and returns a true/false outcome.
* @param expression The expression string to evaluate.
* @return Returns the outcome of the evaluated expression.
*/
function logicEval($expression:String):Boolean;

---

logicParse

/**
* Parses an expression using the Logic API and returns the rendered textual result.
* @param expression The expression string to parse.
* @return Returns the parsed text with all variables and methods evaluated.
*/
function logicParse($expression:String):String;

---

playDialogue

/**
* Sends an XML dialogue description to the engine for playback.
* @param xml The XML dialogue description to play.
*/
function playDialogue($xml:XML):void;

---

getActionTargetInfo

/**
* Gets an object with properties detailing the object currently targeted by the action selector.

* If no object is currently targeted within the action selector, then a blank object is returned.
* While an object is being targeted by the action selector, the information object contains the following:
* - "type" property: specifies the type of object being targeted. This value will be either "puppet" or "item".
* - "target" porperty: specifies the ID name of the target puppet or item.
* - informational object will also contain a list of properties specified by the layer's vars.
*/
function getActionTargetInfo():Object;

---

getInventorySlot

/**
* Gets an inventory slot DisplayObject which can have an inventory item loaded into it.
* Use this method to create additional inventory item displays throughout game media.
* Upon getting a new inventory slot display object, you can load an item into the slot through it's API.
* Cast the return of this method as a DisplayObject to add the object to a display list.
* @return Returns an inventory slot DisplayObject, cast as the ILPInventorySlot interface.
*/
function getInventorySlot():ILPInventorySlot;

Properties

The following is a list of properties available through the Lassie Player external API.

savedGames

/**
* [read-only]
* Returns an array of saved game names that are stored within the project's save directory.
*/
function get savedGames():Array;

---

fullScreen

/**
* [read-write]
* Toggles full-screen display. When in full-screen mode, the Lassie Player leaves the pixel dimensions of a game fixed, and just blacks out the screen around the game window.
*/
function get fullScreen():Boolean;
function set fullScreen($enable:Boolean):void;

---

language

```
/**
* [read-write]
* Specifies the language key that the Lassie Player will attempt to access from all XML game texts.
*/
function get language():String;
function set language($key:String):void;
```

---

voiceEnabled

```
/**
* [read-write]
* Specifies if voice playback is enabled within the engine. This command is currently not implemented, but is included in the interface for future integration.
*/
function get voiceEnabled():Boolean;
function set voiceEnabled($enable:Boolean):void;
```

---

subtitleEnabled

```
/**
* [read-write]
* Specifies if dialogue subtitles are enabled. This property currently has no effect, given that voice features are not implemented so subtitles cannot be disabled.
*/
function get subtitleEnabled():Boolean;
function set subtitleEnabled($enable:Boolean):void;
```

---

voiceVolume

```
/**
* [read-write]
* Specifies the percentage of volume levels for the Lassie Player's voice channel, expressed as a number between 0 (min) and 1 (max).
*/
function get voiceVolume():Number;
function set voiceVolume($percent:Number):void;
```

---

soundfxVolume

```
/**
* [read-write]
* Specifies the percentage of volume levels for the Lassie Player's sound effects channel,
```

expressed as a number between 0 (min) and 1 (max).
*/
function get soundfxVolume():Number;
function set soundfxVolume($percent:Number):void;

---

soundtrackVolume

/**
* [read-write]
* Specifies the percentage of volume levels for the Lassie Player's soundtrack channels, expressed as a number between 0 (min) and 1 (max).
*/
function get soundtrackVolume():Number;
function set soundtrackVolume($percent:Number):void;

Implementing the External API

All resources for implementing the Lassie Player ActionScript interface are available within the "external" package provided in the Lassie Shepherd SDK's "com" script library. The external API's class path is:

```
com.lassie.external
```

Within the "external" package, the primary class that a Lassie developer should be concerned with is the "LPMovieClip" class. All media placed into the Lassie Player must be a MovieClip (not Sprite), given that the Lassie Player uses an image clip's timeline for graphical object states. When creating a new piece of Flash media that needs to hook into Lassie's ActionScript interface, you should subclass the "LPMovieClip" class, like so:

```
package
{
    import com.lassie.external.LPMovieClip;

    public class MyCustomMedia extends LPMovieClip
    {
        public function MyCustomMedia():void {
            super();
        }
    }
}
```

After setting up your extension class, you are free to program your ActionScript media however you'd like. Your media will inherit a property called "lassiePlayer" from the LPMovieClip super-class, which provides all properties and methods of the Lassie Player ActionScript API. You can access properties or call methods like so:

```
var score = lassiePlayer.getCacheVar("gameScore");
lassiePlayer.setCacheVar("gameScore", parseInt(score)+10);
```

In the above example, a game score cache variable is retrieved, then written back into the cache with its value increased by ten. Note: all variables placed into the cache are stored and returned as String objects. When performing numeric operations on cache values, be sure to parse the value as an integer before treating it as a number.

The Life Cycle of Lassie Player Media

In its most basic form, the Lassie Player ActionScript API opens up direct communication between Flash media and the Lassie Player. However, this channel of communication is not available from the time of creation to the time of destruction of the Flash media. Display objects extending from LPMovieClip are subject to a life cycle, each stage dictating the media's current capabilities. There are four main stages in a LPMovieClip's life cycle, they are:

- Creation
- Activation (add to stage)
- Deactivation (remove from stage)
- Destruction

Creation

Creation is the point at which the LPMovieClip is created by its constructor. At this point, the object exists but has not yet established a connection with the Lassie Player, therefor accessing the "lassiePlayer" object property will return null, and member calls will result in an error. Therefor, calls to the "lassiePlayer" object may NOT be made within the object constructor.

Activation

Activation occurs each time that the LPMovieClip instance is added to the main Flash Player's stage. Once added to stage, the LPMovieClip object can transverse up its display parentage until it locates the main Lassie Player object, at which time it establishes a connection to the Lassie Player API. This connection will remain available until the object is destroyed.

Deactivation

Deactivation occurs each time that the LPMovieClip instance is removed from the main Flash Player's stage. By default, all LPMovieClip media is automatically destroyed upon being removed from stage. If you plan to re-parent LPMovieClip media and allow it to be added and removed from the stage multiple times, you'll need to set "autoDestroy" to false, then manually call the LPMovieClip's "destroy()" method upon being done with the LPMovieClip.

Destruction

Destruction occurs when the LPMovieClip's "destroy()" command is called. When an LPMovieClip instance is destroyed, its connection to the Lassie Player API is severed and all life cycle event listeners are removed. Once a LPMovieClip has been destroyed, all references to the object have been nullified and the object will become eligible for garbage

collection. Failure to destroy a LPMovieClip will result in the instance becoming a memory leak with no way to be cleaned up. By default, all LPMovieClip instances will be destroyed upon first deactivation - meaning that they'll be destroyed upon being removed from stage unless "autoDestroy" is specifically disabled.

Tapping LPMovieClip Events

In order to perform actions immediately upon the Lassie Player API becoming available or just prior to it becoming unavailable, you can tap into LPMovieClip events. Events can be captured in one of two ways: by assigning function delegates, or by overriding initialization properties.

Assign Function Delegates

A LPMovieClip instance has an "onActivate" and "onDeactivate" property. These two properties can each be assigned a function reference, at which time their respective functions will be called upon activation and deactivation of the instance. "onActivate" is called each time the media is added to stage, and after a connection to the Lassie Player has been established. "onDeactivate" is called each time the instance is about to be removed from stage, just before the connection to the Lassie Player is severed (assuming "autoDestroy" is enabled). Assigning event functions would look like this:

```
package
{
    import com.lassie.external.LPMovieClip;

    public class MyCustomMedia extends LPMovieClip
    {
        public function MyCustomMedia():void {
            super();
            onActivate = _activate;
            onDeactivate = _deactivate;
        }

        private function _activate():void {
            // do stuff!
        }

        private function _deactivate():void {
            // do stuff!
        }
    }
}
```

Override initialization properties

If you'd prefer to avoid function references, you're free to override the LPMovieClip class' initialize and un-initialize methods. Initialize is called each time the media is added to stage, and after a connection to the Lassie Player has been established. Un-initialize is called each time the instance is about to be removed from stage, just before the connection to the Lassie Player is severed (assuming "autoDestroy" is enabled). Overriding initialization properties looks like this:

```
package
{
    import com.lassie.external.LPMovieClip;

    public class MyCustomMedia extends LPMovieClip
    {
        public function MyCustomMedia():void {
            super();
        }

        override protected function _init():void {
            // do stuff!
        }

        override proctected function _uninit():void {
            // do stuff!
        }
    }
}
```