SERVER-SIDE ACTIONSCRIPT LANGUAGE REFERENCE FOR ADOBE® FLASH® MEDIA INTERACTIVE SERVER

Server-Side ActionScript Language Reference for Adobe® Flash® Media Interactive Server

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.

Contents

Server-Side ActionScript Language Reference

Use Server-Side ActionScript to write server-side code for an Adobe Flash Media Interactive Server application. You can use Server-Side ActionScript to control login procedures, control events, communicate with other servers, allow and disallow users access to various server-side application resources, and let users update and share information.

Server-Side ActionScript is Adobe's name for JavaScript 1.5. Flash Media Interactive Server has an embedded JavaScript engine that compiles and executes server-side scripts. This *Server-Side ActionScript Language Reference* documents the Flash Media Interactive Server host environment classes and functions. You can also use core JavaScript classes, functions, statements, and operators. For more information, see the *Core JavaScript 1.5 Reference* at http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference. For more information about JavaScript, see "About JavaScript" in the Mozilla Developer Center at http://developer.mozilla.org/en/docs/About_JavaScript.

Server-Side ActionScript is similar, but not identical, to ActionScript 1.0. Both languages are based on ECMAScript (ECMA-262) edition 3 language specification. Server-Side ActionScript runs in the Mozilla SpiderMonkey engine embedded in Flash Media Interactive Server. ActionScript 1.0 runs in AVM1 (ActionScript Virtual Machine 1) in Adobe Flash Player. SpiderMonkey implemented the ECMAScript specification exactly and Flash Player AVM1 did not. The biggest difference between Server-Side ActionScript and ActionScript 1.0 is that Server-Side ActionScript is case-sensitive.

Global functions

The following functions are available anywhere in a server-side script:

Signature	Description
`clearInterval()`	Stops a call to the `setInterval()` method.
`getGlobal()`	Provides access to the global object from the secure.asc file while the file is loading.
`load()`	Loads a Server-Side ActionScript file (ASC) or JavaScript file (JS) into the main.asc file.
`protectObject()`	Protects the methods of an object from application code.
`setAttributes()`	Prevents certain methods and properties from being enumerated, writable, and deletable.
`setInterval()`	Calls a function or method at a specified time interval until the `clearInterval()` method is called.
`trace()`	Evaluates an expression and displays the value.

clearInterval()

```
clearInterval(intervalID)
```

Stops a call to the `setInterval()` method.

Availability

Flash Communication Server 1

Parameters

`intervalID` An identifier that contains the value returned by a previous call to the `setInterval()` method.

Example

The following example creates a function named `callback()` and passes it to the `setInterval()` method, which is called every 1000 milliseconds and outputs the message `"interval called."` The `setInterval()` method returns a number that is assigned to the `intervalID` variable. The identifier lets you cancel a specific `setInterval()` call. In the last line of code, the `intervalID` variable is passed to the `clearInterval()` method to cancel the `setInterval()` call.

```
function callback(){trace("interval called");}
var intervalID;
intervalID = setInterval(callback, 1000);
// sometime later
clearInterval(intervalID);
```

getGlobal()

`getGlobal()`

Provides access to the global object from the secure.asc file while the file is loading. Use the `getGlobal()` function to create protected system calls.

Availability

Flash Media Server 2

Details

Flash Media Interactive Server has two script execution modes: secure and normal. In secure mode, only the secure.asc file (if it exists) is loaded and evaluated—no other application scripts are loaded. The `getGlobal()` and `protectObject()` functions are available only in secure mode. These functions are very powerful because they provide complete access to the script execution environment and let you create system objects. Once the secure.asc file is loaded, the server switches to normal script execution mode until the application is unloaded.

To prevent inadvertent access to the global object, always hold its reference in a temporary variable (declared by `var`); do not hold its reference in a member variable or a global variable.

Example

The following code gets a reference to the global object:

```
var global = getGlobal();
```

load()

`load(filename)`

Loads a Server-Side ActionScript file (ASC) or JavaScript file (JS) into the main.asc file. Call this function to load ActionScript libraries. The loaded file is compiled and executed after the main.asc file is successfully loaded, compiled, and executed, but before `application.onAppStart()` is called. The path of the specified file is resolved relative to the main.asc file.

Availability

Flash Communication Server 1

Parameters

`filename` A string indicating the relative path to a script file from the main.asc file.

Example

The following example loads the myLoadedFile.asc file:

```
load("myLoadedFile.asc");
```

protectObject()

```
protectObject(object)
```

Protects the methods of an object from application code. Application code cannot access or inspect the methods directly. You can use this function only in the secure.asc file.

Availability

Flash Media Server 2

Parameters

`object` An object to protect.

Returns

An Object.

Details

After an object is protected, don't reference it in global variables or make it a member of an accessible object. The object returned by `protectObject()` dispatches all method invocations to the underlying object but blocks access to member data. As a result, you can't enumerate or modify members directly. The protected object keeps an outstanding reference to the underlying object, which ensures that the object is valid. The protected object follows normal reference rules and exists while it is referred to.

Flash Media Interactive Server has two script execution modes: secure and normal. In secure mode, only the secure.asc file (if it exists) is loaded and evaluated—no other application scripts are loaded. The `getGlobal()` and `protectObject()` functions are available only in secure mode. These functions are very powerful because they provide complete access to the script execution environment and let you create system objects. Once the secure.asc file is loaded, the server switches to normal script execution mode until the application is unloaded.

For more information, see *Adobe Flash Media Server Developer Guide*.

Example

After secure.asc is executed, calls to `load()` are directed through the user-defined system call, as shown in the following example:

```
var sysobj = {};
sysobj._load = load; // Hide the load function
load = null; // Make it unavailable unpriviliged code.
sysobj.load = function(fname){
    // User-defined code to validate/modify fname
    return this._load(fname);
}
// Grab the global object.
var global = getGlobal();

// Now protect sysobj and make it available as
// "system" globally. Also, set its attributes
```

```
// so that it is read-only and not deletable.

global["system"] = protectObject(sysobj);

setAttributes(global, "system", false, true, true);

// Now add a global load() function for compatibility.
// Make it read-only and nondeletable.

global["load"] = function(path){
    return system.load(path);
}

setAttributes(global, "load", false, true, true);
```

See also

LoadVars class

setAttributes()

`setAttributes(object, propName, enumerable, readonly, permanent)`

Prevents certain methods and properties from being enumerated, writable, and deletable. In a server-side script, all properties in an object are enumerable, writable, and deletable by default. Call `setAttributes()` to change the default attributes of a property or to define constants.

Availability

Flash Media Server 2

Parameters

`object` An Object.

`propName` A string indicating the name of the property in the `object` parameter. Setting attributes on nonexistent properties has no effect.

`enumerable` One of the following values: `true`, `false`, or `null`. Makes a property enumerable if `true` or nonenumerable if `false`; a `null` value leaves this attribute unchanged. Nonenumerable properties are hidden from enumerations (`for var` *`i`* `in` *`obj`*).

`readonly` One of the following values: `true`, `false`, or `null`. Makes a property read-only if `true` or writable if `false`; a `null` value leaves this attribute unchanged. Any attempt to assign a new value is ignored. Typically, you assign a value to a property while the property is writable and then make the property read-only.

`permanent` One of the following values: `true`, `false`, or `null`. Makes a property permanent (nondeletable) if `true` or deletable if `false`; a `null` value leaves this attribute unchanged. Any attempt to delete a permanent property (by calling `delete` *`obj.prop`*) is ignored.

Example

The following code prevents the `resolve()` method from appearing in enumerations:

```
Object.prototype.__resolve = function(methodName){ ... };
setAttributes(Object.prototype, "__resolve", false, null, null);
```

The following example creates three constants on a Constants object and makes them permanent and read-only:

```
Constants.KILO = 1000;
setAttributes(Constants, "KILO", null, true, true);
Constants.MEGA = 1000*Constants.KILO;
```

```
setAttributes(Constants, "MEGA", null, true, true);
Constants.GIGA = 1000*Constants.MEGA; setAttributes(Constants, "GIGA", null, true, true);
```

setInterval()

```
setInterval(function, interval[, p1, ..., pN])
setInterval(object.method, interval[, p1, ..., pN])
```

Calls a function or method at a specified time interval until the `clearInterval()` method is called. This method allows a server-side script to run a routine. The `setInterval()` method returns a unique ID that you can pass to the `clearInterval()` method to stop the routine.

Note: *Standard JavaScript supports an additional usage for the `setInterval()` method, `setInterval(stringToEvaluate, timeInterval)`, which is not supported by Server-Side ActionScript.*

Availability

Flash Communication Server 1

Parameters

`function` A Function object.

`object.method` A method to call on `object`.

`interval` A number indicating the time in milliseconds between calls to `function`.

`p1, ..., pN` Optional parameters passed to `function`.

Returns

An integer that provides a unique ID for this call. If the interval is not set, returns -1.

Example

The following example uses an anonymous function to send the message `"interval called"` to the server log every second:

```
setInterval(function(){trace("interval called");}, 1000);
```

The following example also uses an anonymous function to send the message "interval called" to the server log every second, but it passes the message to the function as a parameter:

```
setInterval(function(s){trace(s);}, 1000, "interval called");
```

The following example uses a named function, `callback1()`, to send the message `"interval called"` to the server log:

```
function callback1(){trace("interval called"); }
setInterval(callback1, 1000);
```

The following example also uses a named function, `callback2()`, to send the message `"interval called"` to the server log, but it passes the message to the function as a parameter:

```
function callback2(s){
    trace(s);
}
setInterval(callback2, 1000, "interval called");
```

The following example uses the second syntax:

```
var a = new Object();
a.displaying=displaying;
setInterval(a.displaying, 3000);
```

```
displaying = function(){
    trace("Hello World");
}
```

The previous example calls the `displaying()` method every 3 seconds and sends the message `"Hello World"` to the server log.

See also

`clearInterval()`

trace()

`trace(expression)`

Evaluates an expression and displays the value. You can use the `trace()` function to debug a script, to record programming notes, or to display messages while testing a file. The `trace()` function is similar to the `alert()` function in JavaScript.

The expression appears in the Live Log panel of the Administration Console; it is also published to the application.xx.log file located in a subdirectory of the *RootInstall*\logs folder. For example, if an application is called myVideoApp, the application log for the default application instance would be located here: *RootInstall*\logs_defaultVHost_\myVideoApp_definst_.

Availability

Flash Communication Server 1

Parameters

`expression` Any valid expression. The values in `expression` are converted to strings if possible.

Application class

Every instance of a Flash Media Server application has an Application object, which is a single instance of the Application class. You don't need to use a constructor function to create an Application object; it is created automatically when an application is instantiated by the server.

Use the Application object to accept and reject client connection attempts, to register and unregister classes and proxies, and to manage the life cycle of an application. The Application object has callback functions that are invoked when an application starts and stops and when a client connects and disconnects.

For more information about the life cycle of an application, see *Adobe Flash Media Server Developer Guide*.

Availability

Flash Communication Server 1

Property summary

Property	Description
application.allowDebug	A boolean value that lets administrators access an application with the Administration API approveDebugSession() method (true) or not (false).
application.clients	Read-only; an Array object containing a list of all the clients connected to an application.
application.config	Provides access to properties of the ApplicationObject element in the Application.xml configuration file.
application.hostname	Read-only; the host name of the server for default virtual hosts; the virtual host name for all other virtual hosts.
application.name	Read-only; the name of the application instance.
application.server	Read-only; the platform and version of the server.

Method summary

Method	Description
application.acceptConnection()	Accepts a connection call from a client to the server.
application.broadcastMsg()	Broadcasts a message to all clients connected to an application instance.
application.clearSharedObjects()	Deletes persistent shared objects files (FSO files) specified by the soPath parameter and clears all properties from active shared objects (persistent and nonpersistent).
application.clearStreams()	Clears recorded streams files associated with an application instance.
application.disconnect()	Terminates a client connection to the application.
application.gc()	Invokes the garbage collector to reclaim any unused resources for this application instance.
application.getStats()	Returns statistics about an application.
application.redirectConnection()	Rejects a connection and provides a redirect URL.
application.registerClass()	Registers a constructor function that is used when deserializing an object of a certain class type.
application.registerProxy()	Maps a method call to another function.
application.rejectConnection()	Rejects the connection call from a client to the server.
application.shutdown()	Unloads the application instance.

Event handler summary

Event handler	Description
application.onAppStart()	Invoked when the server loads an application instance.
application.onAppStop()	Invoked when the server is about to unload an application instance.
application.onConnect()	Invoked when NetConnection.connect() is called from the client.
application.onConnectAccept()	Invoked when a client successfully connects to an application; for use with version 2 components only.
application.onConnectReject()	Invoked when a connection is rejected in an application that contains components.
application.onDisconnect()	Invoked when a client disconnects from an application.

Event handler	Description
`application.onPublish()`	Invoked when a client publishes a stream to an application.
`application.onStatus()`	Invoked when the server encounters an error while processing a message that was targeted at this application instance.
`application.onUnpublish()`	Invoked when a client stops publishing a stream to an application.

application.acceptConnection()

```
application.acceptConnection(clientObj)
```

Accepts a connection call from a client to the server.

Availability

Flash Communication Server 1

Parameters

`clientObj` A Client object; a client to accept.

Details

When `NetConnection.connect()` is called from the client side, it passes a Client object to `application.onConnect()` on the server. Call `application.acceptConnection()` in an `application.onConnect()` event handler to accept a connection from a client. When this method is called, `NetConnection.onStatus()` is invoked on the client with the `info.code` property set to `"NetConnection.Connect.Success"`.

You can use the `application.acceptConnection()` method outside an `application.onConnect()` event handler to accept a client connection that had been placed in a pending state (for example, to verify a user name and password).

When you call this method, `NetConnection.onStatus()` is invoked on the client with the `info.code` property set to `"NetConnection.Connect.Success"`. For more information, see the `NetStatusEvent.info` property in the *ActionScript 3.0 Language and Components Reference* or the `NetConnection.onStatus()` entry in the *Adobe Flash Media Server ActionScript 2.0 Language Reference*.

Note: *When you use version 2 components, the last line (in order of execution) of the `onConnect()` handler should be either `application.acceptConnection()` or `application.rejectConnection()` (unless you're leaving the application in a pending state). Also, any logic that follows `acceptConnection()` or `rejectConnection()` must be placed in the `application.onConnectAccept()` and `application.onConnectReject()` handlers, or it will be ignored.*

Example

The following server-side code accepts a client connection and traces the client ID:

```
application.onConnect = function(client){
    // Accept the connection.
    application.acceptConnection(client);
    trace("connect: " + client.id);
};
```

Note: *This example shows code from an application that does not use components.*

application.allowDebug

```
application.allowDebug
```

A boolean value that lets administrators access an application with the Administration API `approveDebugSession()` method (`true`) or not (`false`). A debug connection lets administrators view information about shared objects and streams in the Administration Console.

The default value for this property is `false` and is set in the Application.xml file:

```
<Application>
    ...
    <Debug>
        <AllowDebugDefault>false</AllowDebugDefault>
    </Debug>
    ...
</Application>
```

Setting `application.allowDebug` to `true` in a server-side script overrides the value in the Application.xml file.

Availability

Flash Media Server 2

application.broadcastMsg()

```
application.broadcastMsg(cmd [, p1,..., pN])
```

Broadcasts a message to all clients connected to an application instance. To handle the message, the client must define a handler on the NetConnection object with the same name as the `cmd` parameter.

Availability

Flash Media Server 2

Parameters

`cmd` A string; a message to broadcast. To handle the message, define a handler with the same name as `cmd` on the client-side NetConnection object.

`p1,..., pN` A string; additional messages to broadcast.

Example

The following server-side code sends a message to the client:

```
application.broadcastMsg("testMessage", "Hello World");
```

The following client-side code catches the message and outputs “Hello World”:

```
nc = new NetConnection();
nc.testMessage = function(msg){
    trace(msg);
};
```

application.clearSharedObjects()

```
application.clearSharedObjects(soPath)
```

Deletes persistent shared objects files (FSO files) specified by the `soPath` parameter and clears all properties from active shared objects (persistent and nonpersistent). Even if you have deleted all the properties from a persistent shared object, unless you call `clearSharedObjects()`, the FSO file still exists on the server.

Availability

Flash Communication Server 1

Parameters

`soPath` A string indicating the Uniform Resource Identifier (URI) of a shared object.

The `soPath` parameter specifies the name of a shared object, which can include a slash (/) as a delimiter between directories in the path. The last element in the path can contain wildcard patterns (for example, a question mark [?] and an asterisk [*]) or a shared object name. The `application.clearSharedObjects()` method traverses the shared object hierarchy along the specified path and clears all the shared objects. Specifying a slash (/) clears all the shared objects that are associated with an application instance.

If `soPath` matches a shared object that is currently active, all its properties are deleted, and a `clear` event is sent to all subscribers of the shared object. If it is a persistent shared object, the persistent store is also cleared.

The following values are possible for the `soPath` parameter:

- `/` clears all local and persistent shared objects associated with the instance.
- `/foo/bar` clears the shared object `/foo/bar`; if bar is a directory name, no shared objects are deleted.
- `/foo/bar/*` clears all shared objects stored under the instance directory /foo/bar. If no persistent shared objects are in use within this namespace, the bar directory is also deleted.
- `/foo/bar/XX??` clears all shared objects that begin with `XX`, followed by any two characters. If a directory name matches this specification, all the shared objects within this directory are cleared.

Returns

A boolean value of `true` if the shared object at the specified path was deleted; otherwise, `false`. If wildcard characters are used to delete multiple files, the method returns `true` only if all the shared objects that match the wildcard pattern were successfully deleted; otherwise, it returns `false`.

Example

The following example clears all the shared objects for an instance:

```
function onApplicationStop(){
    application.clearSharedObjects("/");
}
```

application.clearStreams()

```
application.clearStreams(streamPath)
```

Clears recorded streams files associated with an application instance. You can use this method to clear a single stream, all streams associated with the application instance, just those streams in a specific subdirectory of the application instance, or just those streams whose names match a specified wildcard pattern.

If the `clearStreams()` method is invoked on a stream that is currently recording, the recorded file is set to length 0 (cleared), and the internal cached data is also cleared.

A call to `application.clearStreams()` invokes the `Stream.onStatus()` handler and passes it an information object that contains information about the success or failure of the call.

Note: *You can also use the Administration API `removeApp()` method to delete all the resources for a single application instance.*

Availability

Flash Communication Server 1

Parameters

`streamPath` A string indicating the Uniform Resource Identifier (URI) of a stream.

The `streamPath` parameter specifies the location and name of a stream relative to the directory of the application instance. You can include a slash (/) as a delimiter between directories in the path. The last element in the path can contain wildcard patterns (for example, a question mark [?] and an asterisk [*]) or a stream name. The `clearStreams()` method traverses the stream hierarchy along the specified path and clears all the recorded streams that match the given wildcard pattern. Specifying a slash clears all the streams that are associated with an application instance.

To clear FLV, MP4, or MP3 files, precede the stream path with `flv:`, `mp4:`, or `mp3:`. When you specify `flv:` or `mp3:` you don't have to specify a file extension; .flv and .mp3 are implied. However, when you call `application.clearStreams("mp4:foo")`, the server deletes any file with the name "foo" in an mp4 container; for example, foo.mp4, foo.mov, and foo.f4v. To delete a specific file, pass the file extension in the call; for example, `application.clearStreams("mp4:foo.f4v")`.

Note: *If you don't precede the stream path with a file type, only FLV files are deleted.*

The following examples show some possible values for the `streamPath` parameter:

- `flv:/` clears all FLV streams associated with the application instance.
- `mp3:/` clears all MP3 files associated with the application instance.
- `mp4:/` clears all MP4 streams associated with the application instance (for example, foo.mp4, foo.mov, and so on).
- `mp4:foo.mp4` clears the foo.mp4 file.
- `mp4:foo.mov` clears the foo.mov file.
- `mp3:/mozart/requiem` clears the MP3 file named requiem.mp3 from the application instance's /mozart subdirectory.
- `mp3:/mozart/*` clears all MP3 file from the application instance's /mozart subdirectory.
- `/report` clears the report.flv stream file from the application instance directory.
- `/presentations/intro` clears the recorded intro.flv stream file from the application instance's /presentations subdirectory; if `intro` is a directory name, no streams are deleted.
- `/presentations/*` clears all FLV files from the application instance's /presentations subdirectory. The /presentation subdirectory is also deleted if no streams are used in this namespace.
- `/presentations/report??` clears all FLV files that begin with "report," followed by any two characters. If there are directories within the given directory listing, the directories are cleared of any streams that match `report??`.

Returns

A boolean value of `true` if the stream at the specified path was deleted; otherwise, `false`. If wildcard characters are used to clear multiple stream files, the method returns `true` only if all the streams that match the wildcard pattern were successfully deleted; otherwise, it returns `false`.

Example

The following example clears all recorded streams:

```
function onApplicationStop(){
```

```
    application.clearStreams("/");
}
```

The following example clears all MP3 files from the application instance's /disco subdirectory:

```
function onApplicationStop(){
    application.clearStreams("mp3:/disco/*");
}
```

application.clients

`application.clients`

Read-only; an Array object containing a list of all the clients connected to an application. Each element in the array is a reference to the Client object; use the `application.clients.length` property to determine the number of users connected to an application.

Do not use the index value of the `clients` array to identify users between calls, because the array is compacted when users disconnect and the slots are reused by other Client objects.

Availability

Flash Communication Server 1

Example

The following example uses a `for` loop to iterate through each element in the `application.clients` array and calls the `serverUpdate()` method on each client:

```
for (i = 0; i < application.clients.length; i++){
    application.clients[i].call("serverUpdate");
}
```

application.config

`application.config`

Provides access to properties of the `ApplicationObject` element in the Application.xml configuration file. To access properties that you set in the configuration file, use the `application.config` property. For example, to set the value of the `password` element, use the code `application.config.password`.

For more information, see *Adobe Flash Media Server Configuration and Administration Guide*.

Availability

Flash Media Server 2

Example

Use this sample section from an Application.xml file for this example:

```
<Application>
    <JSEngine>
        <ApplicationObject>
            <config>
                <user_name>jdoe</user_name>
                <dept_name>engineering</dept_name>
            </config>
        </ApplicationObject>
    </JSEngine>
</Application>
```

The following lines of code access the `user_name` and `dept_name` properties:

```
trace("I am " + application.config.user_name + " and I work in the " +
application.config.dept_name + " department.");

trace("I am " + application.config["user_name"] + " and I work in the " +
application.config["dept_name"] + " department.");
```

The following code is sent to the application log file and the Administration Console:

```
I am jdoe and I work in the engineering department.
```

application.disconnect()

```
application.disconnect(clientObj)
```

Terminates a client connection to the application. When this method is called, `NetConnection.onStatus()` is invoked on the client with `info.code` set to `"NetConnection.Connect.Closed"`. The `application.onDisconnect()` handler is also invoked.

Availability

Flash Communication Server 1

Parameters

`clientObj` A Client object indicating the client to disconnect. The object must be a Client object from the `application.clients` array.

Returns

A boolean value of `true` if the disconnection was successful; otherwise, `false`.

Example

The following example calls `application.disconnect()` to disconnect all users from an application instance:

```
function disconnectAll(){
    for (i=0; i < application.clients.length; i++){
        application.disconnect(application.clients[i]);
    }
}
```

application.gc()

```
application.gc()
```

Invokes the garbage collector to reclaim any unused resources for this application instance.

Availability

Flash Media Server 2

application.getStats()

```
application.getStats()
```

Returns statistics about an application.

Availability

Flash Communication Server 1

Returns

An Object whose properties contain statistics about the application instance. The following table describes the properties:

Property	Description
bw_in	Total number of kilobytes received.
bw_out	Total number of kilobytes sent.
bytes_in	Total number of bytes sent.
bytes_out	Total number of bytes received.
msg_in	Total number of Real-Time Messaging Protocol (RTMP) messages sent.
msg_out	Total number of RTMP messages received.
msg_dropped	Total number of RTMP messages dropped.
total_connects	Total number of clients connected to an application instance.
total_disconnects	Total number of clients who have disconnected from an application instance.

Example

The following example outputs application statistics to the Live Log panel in the Administration Console:

```
function testStats(){
    var stats = application.getStats();
    for(var prop in stats){
         trace("stats." + prop + " = " + stats[prop]);
    }
}

application.onConnect = function(client){
    this.acceptConnection(client);
    testStats();
};
```

application.hostname

`application.hostname`

Read-only; the host name of the server for default virtual hosts; the virtual host name for all other virtual hosts.

If an application is running on the default virtual host, and if a value is set in the `ServerDomain` element in the Server.xml configuration file, the `application.hostname` property contains the value set in the `ServerDomain` element. If a value has not been set in the `ServerDomain` element, the property is undefined.

If an application is running on any virtual host other than the default, the `application.hostname` property contains the name of the virtual host.

Availability

Flash Communication Server 1.5

application.name

`application.name`

Read-only; the name of the application instance.

Availability

Flash Communication Server 1

Example

The following example checks the `name` property against a specific string before it executes some code:

```
if (application.name == "videomail/work"){
    // Insert code here.
}
```

application.onAppStart()

```
application.onAppStart = function (){}
```

Invoked when the server first loads the application instance. Use this handler to initialize an application state. The `onAppStart()` event is invoked only once during the lifetime of an application instance.

Availability

Flash Communication Server 1

Example

```
application.onAppStart = function (){
    trace ("*** sample_guestbook application start");

    // Create a reference to a persistent shared object.
    application.entries_so = SharedObject.get("entries_so", true);

    // Prevent clients from updating the shared object.
    application.entries_so.lock();

    // Get the number of entries saved in the shared object
    // and save it in application.lastEntry.
    var maxprop = 0;
    var soProperties = application.entries_so.getPropertyNames();
    trace("soProperties:" + soProperties);
    if (soProperties == null) {
        application.lastEntry = 0;
    } else {
        for (var prop in soProperties) {
            maxprop = Math.max (parseInt(prop), maxprop);
            trace("maxprop " + maxprop);
        }
        application.lastEntry = maxprop+1;
    }
    // Allow clients to update the shared object.
    application.entries_so.unlock();
    trace("*** onAppStart called.");
};
```

application.onAppStop()

```
application.onAppStop = function (info){}
```

Invoked when the server is about to unload an application instance. You can use `onAppStop()` to flush the application state or to prevent the application from being unloaded.

Define a function that is executed when the event handler is invoked. If the function returns `true`, the application is unloaded. If the function returns `false`, the application is not unloaded. If you don't define a function for this event handler, or if the return value is not a boolean value, the application is unloaded when the event is invoked.

The Flash Media Server application passes an information object to the `application.onAppStop()` event. You can use Server-Side ActionScript to look at this information object to decide what to do in the function you define. You can also use the `application.onAppStop()` event to notify users before shutdown.

If you use the Administration Console or the Server Administration API to unload a Flash Media Server application, `application.onAppStop()` is not invoked. Therefore you cannot use `application.onAppStop()` to tell users that the application is exiting.

Availability

Flash Communication Server 1

Parameters

`info` An Object, called an *information object*, with properties that explain why the application is about to stop running. The information object has a `code` property and a `level` property.

Code property	Level property	Description
`Application.Shutdown`	`status`	The application instance is about to shut down.
`Application.GC`	`status`	The application instance is about to be destroyed by the server.

Returns

The value returned by the function you define, if any, or `null`. To unload the application, return `true` or any non-`false` value. To refuse to unload the application, return `false`.

Example

The following example flushes the `entries_so` shared object when the application stops:

```
application.onAppStop = function (info){
    trace("*** onAppStop called.");
    if (info==”Application.Shutdown”){
        application.entries_so.flush();
    }
}
```

application.onConnect()

```
application.onConnect = function (clientObj [, p1, ..., pN]){}
```

Invoked when `NetConnection.connect()` is called from the client. This handler is passed a Client object representing the connecting client. Use the Client object to perform actions on the client in the handler. For example, use this function to accept, reject, or redirect a client connection, perform authentication, define methods on the Client object to be called remotely from `NetConnection.call()`, and set the `Client.readAccess` and `Client.writeAccess` properties to determine client access rights to server-side objects.

When performing authentication, all of the information required for authentication should be sent from the `NetConnection.connect()` method to the `onConnect()` handler as parameters (*p1*..., *pN*).

If you don't define an `onConnect()` handler, connections are accepted by default.

If there are several simultaneous connection requests for an application, the server serializes the requests so that only one `application.onConnect()` handler is executed at a time. It's a good idea to write code for the `application.onConnect()` function that is executed quickly to prevent a long connection time for clients.

Note: *When you are using the version 2 component framework (that is, when you are loading the components.asc file in your server-side script file), you must use the* `application.onConnectAccept()` *method to accept client connections.*

Availability

Flash Communication Server 1

Parameters

`clientObj` A Client object. This object contains information about the client that is connecting to the application.

`p1 ..., pN` Optional parameters passed to the `application.onConnect()` handler from the client-side `NetConnection.connect()` method when a client connects to the application.

Returns

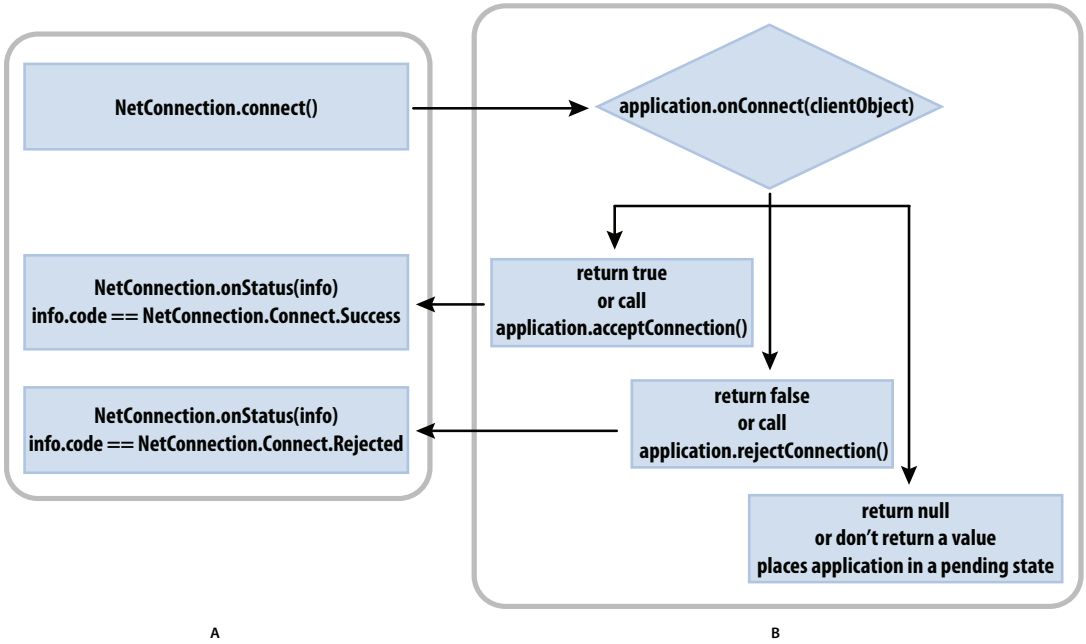
A boolean value; `true` causes the server to accept the connection; `false` causes the server to reject the connection.

When `true` is returned, `NetConnection.onStatus()` is invoked on the client with `info.code` set to `"NetConnection.Connect.Success"`. When `false` is returned, `NetConnection.onStatus()` is invoked on the client with `info.code` set to `"NetConnection.Connect.Rejected"`.

If `null` or no value is returned, the server puts the client in a pending state and the client can't receive or send messages. If the client is put in a pending state, you must call `application.acceptConnection()` or `application.rejectConnection()` at a later time to accept or reject the connection. For example, you can perform external authentication by making a NetConnection call in your `application.onConnect()` event handler to an application server and having the reply handler call `application.acceptConnection()` or `application.rejectConnection()`, depending on the information received by the reply handler.

You can also call `application.acceptConnection()` or `application.rejectConnection()` in the `application.onConnect()` event handler. If you do, any value returned by the function is ignored.

Note: *Returning 1 or 0 is not the same as returning* `true` *or* `false`*. The values 1 and 0 are treated the same as any other integers and do not accept or reject a connection.*



How to use `application.onConnect()` to accept, reject, or put a client in a pending state.
***A.** Client-side ActionScript **B.** Server-Side ActionScript*

Example

The following examples show three ways to accept or reject a connection in the `onConnect()` handler:

```
(Usage 1)
application.onConnect = function (clientObj [, p1, ..., pN]){
    // Insert code here to call methods that do authentication.
    // Returning null puts the client in a pending state.
    return null;
};
(Usage 2)
application.onConnect = function (clientObj [, p1, ..., pN]){
    // Insert code here to call methods that do authentication.
    // The following code accepts the connection:
    application.acceptConnection(clientObj);
};
(Usage 3)
application.onConnect = function (clientObj [, p1, ..., pN])
{
    // Insert code here to call methods that do authentication.
    // The following code accepts the connection by returning true:
    return true;
};
```

The following example verifies that the user has sent the password "XXXX". If the password is sent, the user's access rights are modified and the user can complete the connection. In this case, the user can create or write to streams and shared objects in the user's own directory and can read or view any shared object or stream in this application instance.

```
// This code should be placed in the global scope.

application.onConnect = function (newClient, userName, password){
    // Do all the application-specific connect logic.
    if (password == "XXXX"){
```

```
        newClient.writeAccess = "/" + userName;
        this.acceptConnection(newClient);
    } else {
        var err = new Object();
        err.message = "Invalid password";
        this.rejectConnection(newClient, err);
    }
};
```

If the password is incorrect, the user is rejected and an information object with a `message` property set to `"Invalid password"` is returned to the client side. The object is assigned to `infoObject.application`. To access the `message` property, use the following code on the client side:

```
ClientCom.onStatus = function (info.application.message){
    trace(info.application.message);
    // Prints "Invalid password"
    // in the Output panel on the client side.
};
```

application.onConnectAccept()

```
application.onConnectAccept = function (clientObj [,p1, ..., pN]){}
```

Invoked when a client successfully connects to an application; for use with version 2 components only. Use `onConnectAccept()` to handle the result of an accepted connection in an application that contains components.

If you don't use the version 2 components framework (ActionScript 2.0 components), you can execute code in the `application.onConnect()` handler after accepting or rejecting the connection. When you use the components framework, however, any code that you want to execute after the connection is accepted or rejected must be placed in the `application.onConnectAccept()` and `application.onConnectReject()` event handlers. This architecture allows all of the components to decide whether a connection is accepted or rejected.

Availability

Flash Media Server (with version 2 media components only).

Parameters

`clientObj` A Client object; the client connecting to the application.

`p1, ..., pN` Optional parameters passed to the `application.onConnectAccept()` method. These parameters are passed from the client-side `NetConnection.connect()` method when a client connects to the application; they can be any ActionScript data type.

Example

The following example is client-side code:

```
nc = new NetConnection();
nc.connect("rtmp:/test","jlopes");

nc.onStatus = function(info) {
    trace(info.code);
};

nc.doSomething = function(){
    trace("doSomething called!");
}
```

The following example is server-side code:

```
// When using components, always load components.asc.
load("components.asc");

application.onConnect = function(client, username){
    trace("onConnect called");
    gFrameworkFC.getClientGlobals(client).username = username;
    if (username == "hacker") {
        application.rejectConnection(client);
    }
    else {
        application.acceptConnection(client);
    }
}

// Code is in onConnectAccept and onConnectReject statements
// because components are used.
application.onConnectAccept = function(client, username){
    trace("Connection accepted for "+username);
    client.call("doSomething",null);
}

application.onConnectReject = function(client, username){
    trace("Connection rejected for "+username);
}
```

application.onConnectReject()

```
application.onConnectReject = function (clientObj [,p1, ..., pN]){}
```

Invoked when a connection is rejected in an application that contains components.

If you don't use the version 2 components framework, you can execute code in the `application.onConnect()` handler after accepting or rejecting a connection. When you use the components framework, however, any code that you want to execute after the connection is accepted or rejected must be placed in the `application.onConnectAccept()` and `application.onConnectReject()` framework event handlers. This architecture allows all of the components to decide whether a connection is accepted or rejected.

Availability

Flash Media Server (with version 2 components only)

Parameters

`clientObj` A Client object; the client connecting to the application.

`p1, ..., pN` Optional parameters passed to the `application.onConnectReject()` handler. These parameters are passed from the client-side `NetConnection.connect()` method when a client connects to the application.

Example

The following example is client-side code that you can use for an application:

```
nc = new NetConnection();
nc.connect("rtmp:/test","jlopes");

nc.onStatus = function(info) {
    trace(info.code);
};

nc.doSomething = function(){
    trace("doSomething called!");
```

```
}
```

The following example is server-side code that you can include in the main.asc file:

```
// When using components, always load components.asc.
load( "components.asc" );

application.onConnect = function(client, username){
    trace("onConnect called");
    gFrameworkFC.getClientGlobals(client).username = username;
    if (username == "hacker") {
        application.rejectConnection(client);
    }
    else {
        application.acceptConnection(client);
    }
}

application.onConnectAccept = function(client, username){
    trace("Connection accepted for "+username);
    client.call("doSomething",null);
}

application.onConnectReject = function(client, username){
    trace("Connection rejected for "+username);
}
```

application.onDisconnect()

```
application.onDisconnect = function (clientObj){}
```

Invoked when a client disconnects from an application. Use this event handler to flush any client state information or to notify other users that a user is leaving the application. This handler is optional.

Note: *After a client has disconnected from an application, you cannot use this method to send data back to that disconnected client.*

Availability

Flash Communication Server 1

Parameters

`clientObj` A Client object; a client disconnecting from the application.

Returns

Server ignores any return value.

Example

The following example uses an anonymous function and assigns it to the `application.onDisconnect()` event handler:

```
// This code should be placed in the global scope.
application.onDisconnect = function (client){
    // Do all the client-specific disconnect logic.
    // Insert code here.
    trace("user disconnected");
};
```

application.onPublish()

```
application.onPublish = function (clientObj, streamObj){}
```

Invoked when a client publishes a stream to an application. Use this event handler to send traffic to other servers when you're building a large-scale live broadcasting application; this is called *multipoint publishing*. For example, you can support subscribers in multiple geographic locations by sending traffic from the origin server (Server A) in one city to two origin servers in two different cities (Server B and Server C). The following is the workflow for such a scenario:

1. A client publisher connects to Server A and starts publishing.
2. Server A receives notifications from the event handler `application.onPublish()` in a server-side script.
3. Inside the `onPublish()` handler, create two NetStream objects to Server B and Server C.
4. Call the `NetStream.publish()` method to redirect the publishing data from Server A to Server B and Server C.
5. Subscribers connecting to Server B and Server C get the same live stream.

In this example, the publishing client connects and publishes only to Server A. The rest of the data flow is handled by logic in the server-side script.

Note: *You cannot change Client object properties in this handler.*

Availability

Flash Media Interactive Server 3 and Flash Media Development Server 3

Parameters

`clientObj` A Client object; the client publishing the stream to the application.

`streamObj` A Stream object; the stream being published to the application.

Returns

Server ignores any return value.

Example

For a complete client-side and server-side example of multipoint publishing, see *Publish from server to server* in *Adobe Flash Media Server Developer Guide*.

application.onStatus()

```
application.onStatus = function (infoObject){}
```

Invoked when the server encounters an error while processing a message that was targeted at this application instance. The `application.onStatus()` handler handles any `Stream.onStatus()` or `NetConnection.onStatus()` messages that don't find handlers. Also, there are a few status calls that come only to `application.onStatus()`.

Availability

Flash Communication Server 1

Parameters

`infoObject` An Object with `code` and `level` properties that contain information about the status of an application. Some information objects also have `details` and `description` properties.The following table describes the information object property values:

Code property	Level property	Description
`Application.Script.Error`	`error`	The ActionScript engine has encountered a runtime error. This information object also has the following properties: • `filename`: name of the offending ASC file. • `lineno`: line number where the error occurred. • `linebuf`: source code of the offending line.
`Application.Script.Warning`	`warning`	The ActionScript engine has encountered a runtime warning. This information object also has the following properties: • `filename`: name of the offending ASC file. • `lineno`: line number where the error occurred. • `linebuf`: source code of the offending line.
`Application.Resource.LowMemory`	`warning`	The ActionScript engine is low on runtime memory. This provides an opportunity for the application instance to free some resources or to take suitable action. If the application instance runs out of memory, it is unloaded and all users are disconnected. In this state, the server does not invoke the `application.onDisconnect()` event handler or the `application.onAppStop()` event handler.

Returns

Any value that the callback function returns.

Example

```
application.onStatus = function(info){
    trace("code: " + info.code + " level: " + info.level);
    trace(info.code + " details: " + info.details);
};
// Application.Script.Warning level: warning
```

application.onUnpublish()

```
application.onUnpublish = function (clientObj, streamObj){}
```

Invoked when a client stops publishing a stream to an application. Use this event handler with `application.onPublish()` to send traffic to other servers when you're building a large-scale, live broadcasting application.

Note: *You cannot change Client object properties in this handler.*

Availability

Flash Media Interactive Server 3 and Flash Media Development Server 3

Parameters

`clientObj` A Client object; the client publishing the stream to the application.

`streamObj` A Stream object; the stream being published to the application.

Returns

Server ignores any return value.

Example

For a complete client-side and server-side example, see *Publish from server to server* in *Adobe Flash Media Server Developer Guide*.

application.redirectConnection()

`application.redirectConnection(clientObj, url[, description[, errorObj]])`

Rejects a connection and provides a redirect URL. You must write logic in the `NetConnection.onStatus()` handler that detects redirection and passes the new connection URL to the `NetConnection.connect()` method.

When this method is called, `NetConnection.onStatus()` is invoked on the client and passed an information object with the following values:

Property	Value
`info.code`	`"NetConnection.Connect.Rejected"`
`info.description`	The value passed in the description parameter; if no value is passed in the parameter, the default value is `"Connection failed"`
`info.ex.code`	`302`
`info.ex.redirect`	The new connection URL
`info.level`	`"Error"`

Availability

Flash Media Interactive Server 3 and Flash Media Development Server 3

Parameters

`clientObj` A Client object specifying a client to reject.

`url` A string specifying the new connection URL.

Note: *If you omit this parameter, `rejectConnection()` is called instead.*

`description` A string that lets you provide more information when a connection is redirected.

`errorObj` An object of any type that is sent to the client, explaining the reason for rejection. The `errorObj` object is available in client-side scripts as the `application` property of the information object that is passed to the `NetConnection.onStatus()` call when the connection is rejected.

Example

The following example is server-side code:

```
application.onConnect = function(clientObj, count){
    var err = new Object();
    err.message = "This is being rejected";
    err.message2 = "This is the second message. with number description";
    if (count == 1){
        redirectURI = "rtmp://www.example.com/redirected/fromScript";
        redirectDescription = "this is being rejected via Server Side Script.";
    }
    else if (count == 2){
        redirectURI = "rtmp://www.example2.com/redirected/fromScript";
        redirectDescription = "this is being rejected via Server Side Script.";
    }
    application.redirectConnection(clientObj, redirectURI, redirectDescription, err);
```

```
}
```

The following example is client-side ActionScript 3.0 code:

```
var theConnection:NetConnection;
var theConnection2:NetConnection;
var client:Object = new Object();

function init():void{
    connect_button.label = "Connect";
    disconnect_button.label = "Disconnect";

    connect_button.addEventListener(MouseEvent.CLICK, buttonHandler);
    disconnect_button.addEventListener(MouseEvent.CLICK, buttonHandler);
}

function buttonHandler(event:MouseEvent){
    switch (event.target){
        case connect_button :
            doConnect();
            break;
        case disconnect_button :
            disConnect();
            break;
    }
}
function doConnect(){
    makeConnection(theURI.text);
}


function disConnect(){
    theConnection.close();
}to

function makeConnection(uri:String){
    if (theConnection){
        theConnection.close();
    }
    theConnection = new NetConnection();
    theConnection.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler);
    theConnection.client = client;
    theConnection.connect(uri);
}

function makeConnection2(uri:String){
    if (theConnection2){
        theConnection2.close();
    }
    theConnection2 = new NetConnection();
    theConnection2.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler);
    theConnection2.client = client;
    theConnection2.connect(uri);
}

function netStatusHandler(event:NetStatusEvent):void{
//Check the Redirect code and make connection to redirect URI if appropriate.
    try{
        if (event.info.ex.code == 302){
            var redirectURI:String;
            redirectURI = event.info.ex.redirect;
```

```
            if (redirectURI.charCodeAt(redirectURI.length-1) == 13){
                redirectURI = redirectURI.slice(0,(redirectURI.length-1));
            }
            makeConnection2(redirectURI);
        }
    }
}

init();
```

application.registerClass()

`application.registerClass(className, constructor)`

Registers a constructor function that is used when deserializing an object of a certain class type. If the constructor for a class is not registered, you cannot call the deserialized object's methods. This method is also used to unregister the constructor for a class. This is an advanced use of the server and is necessary only when sending ActionScript objects between a client and a server.

The client and the server communicate over a network connection. Therefore, if you use typed objects, each side must have the prototype of the same objects they both use. In other words, both the client-side and Server-Side ActionScript must define and declare the types of data they share so that there is a clear, reciprocal relationship between an object, method, or property on the client and the corresponding element on the server. You can call `application.registerClass()` to register the object's class type on the server side so that you can use the methods defined in the class.

Constructor functions should be used to initialize properties and methods; they should not be used for executing server code. Constructor functions are called automatically when messages are received from the client and need to be "safe" in case they are executed by a malicious client. You shouldn't define procedures that could result in negative situations, such as filling up the hard disk or consuming the processor.

The constructor function is called before the object's properties are set. A class can define an `onInitialize()` method, which is called after the object has been initialized with all its properties. You can use this method to process data after an object is deserialized.

If you register a class that has its prototype set to another class, you must set the prototype constructor back to the original class after setting the prototype. The second example below illustrates this point.

Note: *Client-side classes must be defined as `function function_name(){}`, as shown in the following examples. If not defined in the correct way, `application.registerClass()` does not identify the class when its instance passes from the client to the server, and an error is returned.*

Availability

Flash Communication Server 1

Parameters

`className` A string indicating the name of an ActionScript class.

`constructor` A constructor function used to create an object of a specific class type during object deserialization. The name of the constructor function must be the same as `className`. During object serialization, the name of the constructor function is serialized as the object's type. To unregister the class, pass the value `null` as the `constructor` parameter. Serialization is the process of turning an object into something that you can send to another computer over the network.

Example

The following example defines a `Color` constructor function with properties and methods. After the application connects, the `registerClass()` method is called to register a class for the objects of type `Color`. When a typed object is sent from the client to the server, this class is called to create the server-side object. After the application stops, the `registerClass()` method is called again and passes the value `null` to unregister the class.

```
function Color(){
    this.red = 255;
    this.green = 0;
    this.blue = 0;
}
Color.prototype.getRed = function(){
    return this.red;
}
Color.prototype.getGreen = function(){
    return this.green;
}
Color.prototype.getBlue = function(){
    return this.blue;
}
Color.prototype.setRed = function(value){
    this.red = value;
}
Color.prototype.setGreen = function(value){
    this.green = value;
}
Color.prototype.setBlue = function(value){
    this.blue = value;
}
application.onAppStart = function(){
    application.registerClass("Color", Color);
};
application.onAppStop = function(){
    application.registerClass("Color", null);
};
```

The following example shows how to use the `application.registerClass()` method with the `prototype` property:

```
function A(){}
function B(){}

B.prototype = new A();
// Set constructor back to that of B.
B.prototype.constructor = B;
// Insert code here.
application.registerClass("B", B);
```

application.registerProxy()

```
application.registerProxy(methodName, proxyConnection [, proxyMethodName])
```

Maps a method call to another function. You can use this method to communicate between different application instances that can be on the same Flash Media Server or on different Flash Media Servers. Clients can execute server-side methods of any application instances to which they are connected. Server-side scripts can use this method to register methods to be proxied to other application instances on the same server or a different server. You can remove or unregister the proxy by calling this method and passing `null` for the `proxyConnection` parameter, which results in the same behavior as never registering the method at all.

Availability

Flash Communication Server 1

Parameters

`methodName` A string indicating the name of a method. All requests to execute `methodName` for this application instance are forwarded to the `proxyConnection` object.

`proxyConnection` A Client or NetConnection object. All requests to execute the remote method specified by `methodName` are sent to the Client or NetConnection object specified in the `proxyConnection` parameter. Any result returned is sent back to the originator of the call. To unregister or remove the proxy, provide a value of `null` for this parameter.

`proxyMethodName` A string indicating the name of a method for the server to call on the object specified by the `proxyConnection` parameter if `proxyMethodName` is different from the method specified by the `methodName` parameter. This is an optional parameter.

Returns

A value that is sent back to the client that made the call.

Example

In the following example, the `application.registerProxy()` method is called in a function in the `application.onAppStart()` event handler and is executed when the application starts. In the function block, a new NetConnection object called `myProxy` is created and connected. The `application.registerProxy()` method is then called to assign the method `getXyz()` to the `myProxy` object.

```
application.onAppStart = function(){
    var myProxy = new NetConnection();
    myProxy.connect("rtmp://xyz.com/myApp");
    application.registerProxy("getXyz", myProxy);
};
```

application.rejectConnection()

```
application.rejectConnection(clientObj[, description[, errObj])
```

Note: *The `description` parameter is supported in Flash Media Interactive Server 3 and Flash Media Development Server 3 and later.*

Rejects the connection call from a client to the server. The `application.onConnect()` handler is invoked when the client calls `NetConnection.connect()`. In the `application.onConnect()` handler, you can either accept or reject the connection. You can also make a call to an application server to authenticate the client before you accept or reject it.

Note: *When you use version 2 components, the last line (in order of execution) of the `onConnect()` handler should be either `application.acceptConnection()` or `application.rejectConnection()` (unless you're leaving the application in a pending state). Also, any logic that follows `acceptConnection()` or `rejectConnection()` must be placed in `application.onConnectAccept()` and `application.onConnectReject()` handlers, or it is ignored. This requirement exists only when you use version 2 components.*

Availability

Flash Communication Server 1

Parameters

`clientObj` A Client object specifying a client to reject.

`description` A string that allows you to provide more information when a connection is redirected.

`errObj` An object of any type that is sent to the client, explaining the reason for rejection. The `errObj` object is available in client-side scripts as the `application` property of the information object that is passed to the `NetConnection.onStatus()` call when the connection is rejected.

Example

In the following example, the client is rejected and sent an error message. This is the server-side code:

```
application.onConnect = function(client){
    // Insert code here.
    var error = new Object();error.message = "Too many connections";
    application.rejectConnection(client, error);
};
```

This is the client-side code:

```
clientConn.onStatus = function (info){
    if (info.code == "NetConnection.Connect.Rejected"){
        trace(info.application.message);
        // Sends the message
        // "Too many connections" to the Output panel
        // on the client side.
    }
};
```

application.server

`application.server`

Read-only; the platform and version of the server.

Availability

Flash Communication Server 1

Example

The following example checks the `server` property against a string before executing the code in the `if` statement:

```
if (application.server == "Flash Media Server-Windows/1.0"){
    // Insert code here.
}
```

application.shutdown()

`application.shutdown()`

Unloads the application instance. If the application is running in vhost or application-level scope, only the application instance is unloaded, but the core process remains running. If the application is running in instance scope, the application instance is unloaded and the core process terminates. This process is done asynchronously; the instance is unloaded when the unload sequence begins, not when the `shutdown()` call returns.

After `shutdown()` is called, `application.onAppStop()` is called, connected clients are disconnected, and `application.onDisconnect()` is called for each client. Calls made after calling `shutdown()` may not be executed.

Availability

Flash Media Server 2

Returns

A boolean value indicating success (`true`) or failure (`false`).

Client class

The Client class lets you handle each user, or *client*, connection to a Flash Media Server application instance. The server automatically creates a Client object when a user connects to an application; the object is destroyed when the user disconnects from the application. Users have unique Client objects for each application to which they are connected. Thousands of Client objects can be active at the same time.

You can use the properties of the Client class to determine the version, platform, and IP address of each client. You can also set individual read and write permissions to various application resources such as Stream objects and shared objects. Use the methods of the Client class to set bandwidth limits and to call methods in client-side scripts.

When you call `NetConnection.call()` from a client-side ActionScript script, the method that is executed in the server-side script must be a method of the Client class. In your server-side script, you must define any method that you want to call from the client-side script. You can also call any methods that you define in the server-side script directly from the Client class instance in the server-side script.

If all instances of the Client class (each client in an application) require the same methods or properties, you can add those methods and properties to the class itself instead of adding them to each instance of a class. This process is called *extending* a class. To extend a class, instead of defining methods in the constructor function of the class or assigning them to individual instances of the class, you assign methods to the `prototype` property of the constructor function of the class. When you assign methods and properties to the `prototype` property, the methods are automatically available to all instances of the class.

The following code shows how to assign methods and properties to an instance of a class. In the `application.onConnect()` handler, the client instance `clientObj` is passed to the server-side script as a parameter. You can then assign a property and method to the client instance.

```
application.onConnect = function(clientObj){
    clientObj.birthday = myBDay;
    clientObj.calculateDaysUntilBirthday = function(){
        // Insert code here.
    }
};
```

The previous example works, but must be executed every time a client connects. If you want the same methods and properties to be available to all clients in the `application.clients` array without defining them every time, assign them to the `prototype` property of the Client class.

There are two steps to extending a built-in class by using the `prototype` property. You can write the steps in any order in your script. The following example extends the built-in Client class, so the first step is to write the function that you will assign to the `prototype` property:

```
// First step: write the functions.

function Client_getWritePermission(){
// The writeAccess property is already built in to the Client class.
    return this.writeAccess;
}

function Client_createUniqueID(){
    var ipStr = this.ip;
```

```
// The ip property is already built in to the Client class.
    var uniqueID = "re123mn"
// You would need to write code in the above line
// that creates a unique ID for each client instance.
    return uniqueID;
}

// Second step: assign prototype methods to the functions.

Client.prototype.getWritePermission = Client_getWritePermission;
Client.prototype.createUniqueID = Client_createUniqueID;

// A good naming convention is to start all class method
// names with the name of the class followed by an underscore.
```

You can also add properties to `prototype`, as shown in the following example:

```
Client.prototype.company = "Adobe";
```

The methods are available to any instance, so within `application.onConnect()`, which is passed a `clientObj` parameter, you can write the following code:

```
application.onConnect = function(clientObj){
    var clientID = clientObj.createUniqueID();
    var clientWritePerm = clientObj.getWritePermission();
};
```

Availability

Flash Communication Server 1

Property summary

Property	Description
`Client.agent`	Read-only; the version and platform of the client.
`Client.audioSampleAccess`	Enables Flash Player to access raw, uncompressed audio data from streams in the specified folders.
`Client.id`	Read-only; a string that uniquely identifies the client.
`Client.ip`	Read-only; A string containing the IP address of the client.
`Client.pageUrl`	Read-only; A string containing the URL of the web page in which the client SWF file is embedded.
`Client.protocol`	Read-only; A string indicating the protocol used by the client to connect to the server.
`Client.readAccess`	A string of directories containing application resources (shared objects and streams) to which the client has read access.
`Client.referrer`	Read-only; A string containing the URL of the SWF file or the server in which this connection originated.
`Client.secure`	Read-only; A boolean value that indicates whether this is an SSL connection (`true`) or not (`false`).
`Client.uri`	Read-only; the URI specified by the client to connect to this application instance.

Property	Description
`Client.videoSampleAccess`	Enables Flash Player to access raw, uncompressed video data from streams in the specified folders.
`Client.virtualKey`	A virtual mapping for clients connecting to the server.
`Client.writeAccess`	Provides write access to directories that contain application resources (such as shared objects and streams) for this client.

Method summary

Method	Description
`Client.call()`	Executes a method on a client or on another server.
`Client.checkBandwidth()`	Call this method from a client-side script to detect bandwidth.
`Client.getBandwidthLimit()`	Returns the maximum bandwidth that the client or the server can use for this connection.
`Client.getStats()`	Returns statistics for the client.
`Client.getStreamLength()`	Returns the length of a stream, in seconds.
`Client.ping()`	Sends a "ping" message to the client and waits for a response.
`Client.remoteMethod()`	Invoked when a client or another server calls the NetConnection.call() method.
`Client.__resolve()`	Provides values for undefined properties.
`Client.setBandwidthLimit()`	Sets the maximum bandwidth for this client from client to server, server to client, or both.

Client.agent

`clientObject.agent`

Read-only; the version and platform of the client.

When a client connects to the server, the format of `Client.agent` is as follows:

`Operating_System Flash_Player_Version`

For example, if Flash Player version 9.0.45.0 is running on Windows, the value of `Client.agent` is:

`"WIN 9,0,45,0".`

When a connection is made to another Flash Media Interactive Server, the format of `Client.agent` is as follows:

`Server_Name/Server_Version Operating_System/Operating_System_Build`

For example, if the server version is 3.0.0 and it's running on Windows Server 2003, the value of `Client.agent` is:

`"FlashCom/3.0.0 WIN/5.1.2600".`

Availability

Flash Communication Server 1

Example

The following example checks the `agent` property against the string `"WIN"` and executes different code depending on whether they match. This code is written in an `onConnect()` function:

```
function onConnect(newClient, name){
    if (newClient.agent.indexOf("WIN") > -1){
        trace ("Window user");
```

```
    }   else {
        trace ("non Window user.agent is" + newClient.agent);
    }
}
```

Client.audioSampleAccess

`clientObject.audioSampleAccess`

Enables Flash Player to access raw, uncompressed audio data from streams in the specified folders.

Call the `SoundMixer.computeSpectrum()` method in client-side ActionScript 3.0 to read the raw sound data for a waveform that is currently playing. For more information, see the `SoundMixer.computeSpectrum()` entry in the *ActionScript 3.0 Language and Components Reference* and "Accessing raw sound data" in *Programming ActionScript 3.0*.

Availability

Flash Media Interactive Server 3 and Flash Media Development Server 3

Example

The following server-side code sets the `audioSampleAccess` directory to `publicdomain`:

```
application.onConnect = function(client) {

    // Anyone can play free content, which is all streams placed under the
    // samples/, publicdomain/ and contrib/ folders.
    client.readAccess = "samples;publicdomain;contrib";

    // Paying customers get to watch more streams.
    if ( isPayingCustomer(client))
        client.readAccess += "nonfree;premium";

    // Content can be saved (user recorded streams) to contrib/ folder.
    client.writeAccess = "contrib";

    // Anyone can gain access to an audio snapshot of the publicdomain/ folder.
    client.audioSampleAccess = "publicdomain";

    // Paying customers can also get a video snapshot of the publicdomain/ folder.
    if (isPayingCustomer(client))
        client.videoSampleAccess = "publicdomain";
}
```

See also

`Client.videoSampleAccess`

Client.call()

`clientObject.call(methodName, [resultObj, [p1, ..., pN]])`

Executes a method on a client or on another server. The remote method may return data to the `resultObj` parameter, if provided. Whether the remote agent is a client or another server, the method is called on the remote agent's NetConnection object.

Availability

Flash Communication Server 1

Parameters

`methodName` A string indicating a method specified in the form `"[objectPath/]method"`. For example, the command `"someObj/doSomething"` tells the client to invoke the `NetConnection.someObj.doSomething()` method on the client.

`resultObj` An Object. This is an optional parameter that is required when the sender expects a return value from the client. If parameters are passed but no return value is desired, pass the value `null`. The result object can be any object that you define. To be useful, it should have two methods that are invoked when the result arrives: `onResult()` and `onStatus()`. The `resultObj.onResult()` event is triggered if the invocation of the remote method is successful; otherwise, the `resultObj.onStatus()` event is triggered.

`p1, ..., pN` Optional parameters that can be of any ActionScript type, including a reference to another ActionScript object. These parameters are passed to the `methodName` parameter when the method is executed on the Flash client. If you use these optional parameters, you must pass in some value for `resultObj`; if you do not want a return value, pass `null`.

Returns

A boolean value of `true` if a call to `methodName` was successful on the client; otherwise, `false`.

Example

The following example shows a client-side script that defines a function called `getNumber()`, which generates a random number:

```
nc = new NetConnection();
nc.getNumber = function(){
    return (Math.random());
};

nc.connect("rtmp:/clientCall");
```

The following server-side script calls `Client.call()` in the `application.onConnect()` handler to call the `getNumber()` method that was defined on the client. The server-side script also defines a function called `randHander()`, which is used in the `Client.call()` method as the `resultObj` parameter.

```
randHandler = function(){
    this.onResult = function(res){
        trace("Random number: " + res);
    }
    this.onStatus = function(info){
        trace("Failed with code:" + info.code);
    }
};
application.onConnect = function(clientObj){
    trace("Connected");
    application.acceptConnection(clientObj);
    clientObj.call("getNumber", new randHandler());
};
```

Note: *This example does not work with version 2 components. For an example of calling `Client.call()` when using version 2 components, see `application.onConnectAccept()`.*

Client.checkBandwidth()

```
clientObject.checkBandwidth()
```

Call this method from a client-side script to detect client bandwidth. If the client is connected directly to the origin server, bandwidth detection occurs on the origin. If the client is connected to the origin server through an edge server, bandwidth detection happens at the first edge to which the client connected.

To use this method to detect client bandwidth, you must also define `onBWDone()` and `onBWCheck()` methods in a client-side script. For more information, see *Detect bandwidth* in *Adobe Flash Media Server Developer Guide*.

Note: *If this function is defined in a server-side script, the client call invokes that instead of the* `checkBandwith()` *definition in the core server code.*

Availability

Flash Media Interactive Server 3 and Flash Media Development Server 3

Client.getBandwidthLimit()

```
clientObject.getBandwidthLimit(iDirection)
```

Returns the maximum bandwidth that the client or the server can use for this connection. Use the `iDirection` parameter to get the value for each direction of the connection. The value returned indicates bytes per second and can be changed with the `Client.setBandwidthLimit()` method. Set the default value for a connection in the Application.xml file of each application.

Availability

Flash Communication Server 1

Parameters

`iDirection` A number specifying the connection direction. The value 0 indicates a client-to-server direction; 1 indicates a server-to-client direction.

Returns

A number.

Example

The following example uses `Client.getBandwidthLimit()` to set the variables `clientToServer` and `serverToClient`:

```
application.onConnect = function(newClient){
    var clientToServer= newClient.getBandwidthLimit(0);var serverToClient=
newClient.getBandwidthLimit(1);
};
```

Client.getStats()

```
clientObject.getStats()
```

Returns statistics for the client.

Availability

Flash Communication Server 1

Returns

An Object with various properties for each statistic returned. The following table describes the properties of the returned object:

Property	Description
bytes_in	Total number of bytes received.
bytes_out	Total number of bytes sent.
msg_in	Total number of RTMP messages received.
msg_out	Total number of RTMP messages sent.
msg_dropped	Total number of dropped RTMP messages.
ping_rtt	Length of time the client takes to respond to a ping message.
audio_queue_msgs	Current number of audio messages in the queue waiting to be delivered to the client.
video_queue_msgs	Current number of video messages in the queue waiting to be delivered to the client.
so_queue_msgs	Current number of shared object messages in the queue waiting to be delivered to the client.
data_queue_msgs	Current number of data messages in the queue waiting to be delivered to the client.
dropped_audio_msgs	Number of audio messages that were dropped.
dropped_video_msgs	Number of video messages that were dropped.
audio_queue_bytes	Total size of all audio messages (in bytes) in the queue waiting to be delivered to the client.
video_queue_bytes	Total size of all video messages (in bytes) in the queue waiting to be delivered to the client.
so_queue_bytes	Total size of all shared object messages (in bytes) in the queue waiting to be delivered to the client.
data_queue_bytes	Total size of all data messages (in bytes) in the queue waiting to be delivered to the client.
dropped_audio_bytes	Total size of all audio messages (in bytes) that were dropped.
dropped_video_bytes	Total size of all video messages (in bytes) that were dropped.
bw_out	Current upstream (client to server) bandwidth for this client.
bw_in	Current downstream (server to client) bandwidth for this client.
client_id	A unique ID issued by the server for this client.

Example

The following example outputs a client's statisitics:

```
function testStats(){
    var stats = client.getStats();
    for(var prop in stats){
         trace("stats." + prop + " = " + stats[prop]);
    }
}

application.onConnect = function(client){
    this.acceptConnection(client);
    testStats();
};
```

Client.getStreamLength()

```
clientObject.getStreamLength(streamObj)
```

Returns the length of a stream, in seconds. Call this method from a client-side script and specify a response object to receive the returned value.

Availability
Flash Media Interactive Server 3 and Flash Media Development Server 3

Parameters
`streamObj` A Stream object.

Returns
A number.

Example
The following client-side code gets the length of the `sample_video` stream and returns the value to `returnObj`:

```
nc.call("getStreamLength", returnObj, "sample_video");
```

Client.id
`clientObject.id`

Read-only; a string that uniquely identifies the client.

Availability
Flash Media Server 3

Example
The following `onConnect()` function traces the ID of the connecting client:

```
application.onConnect(newClient) {
    trace(newClient.id);
}
```

Client.ip
`clientObject.ip`

Read-only; A string containing the IP address of the client.

Availability
Flash Communication Server 1

Example
The following example uses the `Client.ip` property to verify whether a new client has a specific IP address. The result determines which block of code runs.

```
application.onConnect = function(newClient, name){
    if (newClient.ip == "127.0.0.1"){
        // Insert code here.
    } else {
        // Insert code here.
    }
};
```

Client.pageUrl
`clientObject.pageUrl`

Read-only; A string containing the URL of the web page in which the client SWF file is embedded. If the SWF file isn't embedded in a web page, the value is the location of the SWF file. The following code shows the two examples:

```
// trace.swf file is embedded in trace.html.
client.pageUrl: http://www.example.com/trace.html

// trace.swf is not embedded in an html file.
client.pageUrl: http://www.example.com/trace.swf
```

The value can be an HTTP address or a local file address (for example, file:///C:/Flash Media Server applications/example.html).

Availability

Flash Media Server 2

Example

The following example uses the `Client.pageURI` property to verify whether a new client is located at a particular URI. The result determines which block of code runs.

```
application.onConnect = function(newClient){
    if (newClient.pageUrl == "http://www.example.com/index.html"){
        return true;
    } else {
        return false;
    }
};
```

Client.ping()

```
clientObject.ping()
```

Sends a "ping" message to the client and waits for a response. If the client responds, the method returns `true`; otherwise, `false`. Use this method to determine whether the client connection is still active.

Availability

Flash Communication Server 1

Example

The following `onConnect()` function pings the connecting client and traces the results of the method:

```
application.onConnect(newClient) {
    if (newClient.ping()){
        trace("ping successful");
    }
    else {
        trace("ping failed");
    }
}
```

See also

`Client.getStats()`

Client.protocol

```
clientObject.protocol
```

Read-only; A string indicating the protocol used by the client to connect to the server. This string can have one of the following values:

Protocol	Description
`rtmp`	RTMP over a persistent socket connection.
`rtmpt`	RTMP tunneled over HTTP.
`rtmps`	RTMP over an SSL (Secure Socket Layer) connection.
`rtmpe`	An encrypted RTMP connection.
`rtmpte`	An encrypted RTMP connection tunneled over HTTP.

Availability

Flash Communication Server 1

Example

The following example checks the connection protocol used by a client upon connection to the application:

```
application.onConnect(clientObj){
    if(clientObj.protocol == "rtmp") {
        trace("Client connected over RTMP");
    } else if(clientOjb.protocol == "rtmpt") {
        trace("Client connected over RTMP tunneled over HTTP");
    }
}
```

Client.readAccess

`clientObject.readAccess`

A string of directories containing application resources (shared objects and streams) to which the client has read access. To give a client read access to directories that contain application resources, list the directories in a string delimited by semicolons.

Availability

Flash Communication Server 1

Details

By default, all clients have full read access, and the `readAccess` property is set to slash (`/`). To give a client read access, specify a list of access levels (in URI format), delimited by semicolons. Any files or directories within a specified URI are also considered accessible. For example, if `myMedia` is specified as an access level, any files or directories in the myMedia directory are also accessible (for example, myMedia/mp3s). Similarly, any files or directories in the myMedia/mp3s directory are also accessible, and so on.

Clients with read access to a directory that contains streams can play streams in the specified access levels. Clients with read access to a directory that contains shared objects can subscribe to shared objects in the specified access levels and receive notification of changes in the shared objects.

- For streams, `readAccess` controls the streams that the connection can play.
- For shared objects, `readAccess` controls whether the connection can listen to shared object changes.

Although you cannot use this property to control access for a particular file, you can create a separate directory for a file if you want to control access to it.

Note: *You cannot set this property in the* `application.onPublish()` *event.*

Example

The following `onConnect()` function gives a client read access to myMedia/mp3s, myData/notes, and any files or directories within them:

```
application.onConnect = function(newClient, name){
    newClient.readAccess = "myMedia/mp3s;myData/notes";
};
```

Client.referrer

```
clientObject.referrer
```

Read-only; A string containing the URL of the SWF file or the server in which this connection originated.

Availability

Flash Communication Server 1

Example

```
application.onConnect = function(newClient, name){
    trace("New user connected to server from" + newClient.referrer);
};
```

Client.remoteMethod()

```
myClient.remoteMethod = function([p1, ..., pN]){}
```

Invoked when a client or another server calls the `NetConnection.call()` method. A `remoteMethod` parameter is passed to `NetConnection.call()`. The server searches the Client object instance for a method that matches the `remoteMethod` parameter. If the method is found, it is invoked and the return value is sent back to the result object specified in the call to `NetConnection.call()`.

Availability

Flash Communication Server 1

Parameters

`p1, ..., pN` Optional parameters passed to the `NetConnection.call()` method.

Example

The following example creates a method called `sum()` as a property of the Client object `newClient` on the server side:

```
newClient.sum = function(op1, op2){
    return op1 + op2;
};
```

The `sum()` method can then be called from `NetConnection.call()` on the client side:

```
nc = new NetConnection();
nc.connect("rtmp://myServer/myApp");
nc.call("sum", new result(), 20, 50);
function result(){
    this.onResult = function (retVal){
        output += "sum is " + retVal;
    };
    this.onStatus = function(errorVal){
```

```
        output += errorVal.code + " error occurred";
    };
}
```

The `sum()` method can also be called on the server:

```
newClient.sum();
```

The following example creates two functions that you can call from either a client-side or server-side script:

```
application.onConnect = function(clientObj) {
    // The function foo returns 8.
    clientObj.foo = function() {return 8;};
    // The function bar is defined outside the onConnect call.
    clientObj.bar = application.barFunction;
};
// The bar function adds the two values it is given.
application.barFunction = function(v1,v2) {
    return (v1 + v2);
};
```

You can call either of the two functions that were defined in the previous example (`foo` and `bar`) by using the following code in a client-side script:

```
c = new NetConnection();
c.call("foo");
c.call("bar", null, 1, 1);
```

You can call either of the two functions that were defined in the previous example (`foo` and `bar`) by using the following code in a server-side script:

```
c = new NetConnection();
c.onStatus = function(info) {
    if(info.code == "NetConnection.Connect.Success") {
        c.call("foo");
        c.call("bar", null, 2, 2);
    }
};
```

Client.__resolve()

```
Client.__resolve = function(propName){}
```

Provides values for undefined properties. When an undefined property of a Client object is referenced by Server-Side ActionScript code, the Client object is checked for a `_resolve()` method. If the object has a `_resolve()` method, it is invoked and passed the name of the undefined property. The return value of the `_resolve()` method is the value of the undefined property. In this way, `_resolve()` can supply the values for undefined properties and make it appear as if they are defined.

Availability

Flash Communication Server 1

Parameters

`propName` A string indicating the name of an undefined property.

Returns

The value of the property specified by the `propName` parameter.

Example

The following example defines a function that is called whenever an undefined property is referenced:

```
Client.prototype.__resolve = function (name) {
    return "Hello, world!";
};
function onConnect(newClient){
// Prints "Hello World".
    trace (newClient.property1);
}
```

Client.secure

`clientObject.secure`

Read-only; A boolean value that indicates whether this is an SSL connection (`true`) or not (`false`).

Availability

Flash Media Server 2

Client.setBandwidthLimit()

`clientObject.setBandwidthLimit(iServerToClient, iClientToServer)`

Sets the maximum bandwidth for this client from client to server, server to client, or both. The default value for a connection is set for each application in the `Client` section of the Application.xml file. The value specified cannot exceed the bandwidth cap value specified in the Application.xml file. For more information, see *Adobe Flash Media Server Configuration and Administration Guide*.

Availability

Flash Communication Server 1

Parameters

`iServerToClient` A number; the bandwidth from server to client, in bytes per second. Use 0 if you don't want to change the current setting.

`iClientToServer` A number; the bandwidth from client to server, in bytes per second. Use 0 if you don't want to change the current setting.

Example

The following example sets the bandwidth limits for each direction, based on values passed to the `onConnect()` function:

```
application.onConnect = function(newClient, serverToClient, clientToServer){
    newClient.setBandwidthLimit(serverToClient, clientToServer);
    application.acceptConnection(newClient);
}
```

Client.uri

`clientObject.uri`

Read-only; the URI specified by the client to connect to this application instance.

Availability

Flash Media Server 2

Example

The following example defines an `onConnect()` callback function that sends a message indicating the URI that the new client used to connect to the application:

```
application.onConnect = function(newClient, name){
    trace("New user requested to connect to " + newClient.uri);
};
```

Client.videoSampleAccess

`clientObject.videoSampleAccess`

Enables Flash Player to access raw, uncompressed video data from streams in the specified folders.

Calls the `BitmapData.draw()` method in client-side ActionScript 3.0 to read the raw data for a stream that is currently playing. For more information, see the `BitmapData.draw()` entry in *ActionScript 3.0 Language and Components Reference*.

Availability

Flash Media Interactive Server 3 and Flash Media Development Server 3

Example

The following server-side code sets the `videoSampleAccess` directory to `publicdomain` for paying customers:

```
application.onConnect = function(client) {

    // Anyone can play free content, which is all streams placed under the
    // samples/, publicdomain/, and contrib/ folders.
    client.readAccess = "samples;publicdomain;contrib";

    // Paying customers get to watch more streams.
    if ( isPayingCustomer(client))
        client.readAccess += "nonfree;premium";

    // Content can be saved (user recorded streams) to the contrib/ folder.
    client.writeAccess = "contrib";

    // Anyone can gain access to an audio snapshot of the publicdomain/ folder.
    client.audioSampleAccess = "publicdomain";

    // Paying customers can also get a video snapshot of the publicdomain/ folder.
    if (isPayingCustomer(client))
        client.videoSampleAccess = "publicdomain";
}
```

See also

`Client.audioSampleAccess`

Client.virtualKey

`clientObject.virtualKey`

A virtual mapping for clients connecting to the server. When a client connects, it receives a virtual key that corresponds to ranges that you set in the Vhost.xml file. You can use `Client.virtualKey` to change that value in a server-side script. The following is the code in the Vhost.xml file that you must configure:

```
<VirtualKeys>
    <!-- Create your own ranges and key values.-->
```

```
    <!-- You can create as many Key elements as you need.-->
    <Key from="WIN 7,0,19,0" to="WIN 9,0,0,0">A</Key>
</VirtualKeys>
```

Using the previous Vhost.xml file, if a Flash Player 8 client connected to the server, its `Client.virtualKey` value would be `A`.

Note: *A legal key cannot contain the characters "*" and ":".*

Use this property in conjunction with `Stream.setVirtualPath()` to map stream URLs to physical locations on the server. This allows you to serve different content to different versions of Flash Player. For more information, see `Stream.setVirtualPath()`.

Availability

Flash Media Server 2

Client.writeAccess

`clientObject.writeAccess`

Provides write access to directories that contain application resources (such as shared objects and streams) for this client. To give a client write access to directories that contain application resources, list directories in a string delimited by semicolons. By default, all clients have full write access, and the `writeAccess` property is set to slash (`/`). For example, if `myMedia` is specified as an access level, then any files or directories in the myMedia directory are also accessible (for example, myMedia/myStreams). Similarly, any files or subdirectories in the myMedia/myStreams directory are also accessible, and so on.

- For shared objects, `writeAccess` provides control over who can create and update the shared objects.
- For streams, `writeAccess` provides control over who can publish and record a stream.

You cannot use this property to control access to a single file. To control access to a single file, create a separate directory for the file.

Don't precede the stream path with a leading slash (/) on the client side.

Note: *You cannot set this property in the `application.onPublish()` event.*

Availability

Flash Communication Server 1

Example

The following example provides write access to the /myMedia/myStreams and myData/notes directories:

```
application.onConnect = function(newClient, name){
    newClient.writeAccess = "/myMedia/myStreams;myData/notes";
    application.acceptConnection();
};
```

The following example completely disables write access:

```
application.onConnect = function(clientObj){
    clientObj.writeAccess = "";
    return true;
};
```

See also

`Client.readAccess`

File class

The File class lets applications write to the server's file system. This is useful for storing information without using a database server, creating log files for debugging, and tracking usage. Also, a directory listing is useful for building a content list of streams or shared objects without using Flash Remoting.

By default, a script can access files and directories only within the application directory of the hosting application. A server administrator can grant access to additional directories by specifying virtual directory mappings for File object paths. This is done in the `FileObject` tag in the Application.xml file, as shown in the following example:

```
<FileObject>
    <VirtualDirectory>/videos;C:\myvideos</VirtualDirectory>
    <VirtualDirectory>/fmsapps;C:\Program Files\fms\applications</VirtualDirectory>
</FileObject>
```

This example specifies two additional directory mappings in addition to the default application directory. Any path that begins with /videos—for example, /videos/xyz/vacation.flv—maps to c:/myvideos/xyz/vaction.flv. Similarly, /fmsapps/conference maps to c:/Program Files/fms/applications/conference. Any path that does not match a mapping resolves to the default application folder. For example, if c:/myapps/filetest is the application directory, then /streams/hello.flv maps to c:/myapps/filetest/streams/hello.flv.

Note: *You can use an Application.xml file at the virtual host level or at the application level. For more information, see* Adobe Flash Media Server Configuration and Administration Guide.

In addition, the following rules are enforced by the server:

- File objects cannot be created by using native file path specification.
- File object paths must follow the URI convention:

  A slash (/) must be used as the path separator. Access is denied if a path contains a backslash (\), or if a dot (.) or two dots (..) is the only string component found between path separators.
- Root objects cannot be renamed or deleted.

  For example, if a path using a slash (/) is used to create a File object, the application folder is mapped.

Availability

Flash Media Server 2

Property summary

Property	Description
`File.canAppend`	Read-only; a boolean value indicating whether a file can be appended (`true`) or not (`false`).
`File.canRead`	Read-only; A boolean value indicating whether a file can be read (`true`) or not (`false`).
`File.canReplace`	Read-only; A boolean value indicating whether a file was opened in `"create"` mode (`true`) or not (`false`). This property is undefined for closed files.
`File.canWrite`	Read-only; a boolean value indicating whether a file can be written to (`true`) or not (`false`).
`File.creationTime`	Read-only; a Date object containing the time the file was created.
`File.exists`	Read-only; a boolean value indicating whether the file or directory exists (`true`) or not (`false`).
`File.isDirectory`	Read-only; a boolean value indicating whether the file is a directory (`true`) or not (`false`).

Property	Description
File.isFile	Read-only; a boolean value indicating whether the file is a regular data file (true) or not (false).
File.isOpen	Read-only; a boolean value indicating whether the file has been successfully opened and is still open (true) or not (false).
File.lastModified	Read-only; a Date object containing the time the file was last modified.
File.length	Read-only; for a directory, the number of files in the directory, not counting the current directory and parent directory entries; for a file, the number of bytes in the file.
File.mode	Read-only; the mode of an open file.
File.name	Read-only; a string indicating the name of the file.
File.position	The current offset in the file.
File.type	Read-only; a string specifying the type of data or encoding used when a file is opened.

Method summary

Method	Description
File.close()	Closes the file.
File.copyTo()	Copies a file to a different location or copies it to the same location with a different filename.
File.eof()	Returns a boolean value indicating whether the file pointer is at the end of file (true) or not (false).
File.flush()	Flushes the output buffers of a file.
File.list()	If the file is a directory, lists the files in the directory.
File.mkdir()	Creates a directory.
File.open()	Opens a file so that you can read from it or write to it.
File.read()	Reads the specified number of characters from a file and returns a string.
File.readAll()	Reads the file after the location of the file pointer and returns an array with an element for each line of the file.
File.readByte()	Reads the next byte from the file and returns the numeric value of the next byte, or -1 if the operation fails.
File.readln()	Reads the next line from the file and returns it as a string.
File.remove()	Removes the file or directory pointed to by the File object.
File.renameTo()	Moves or renames a file.
File.seek()	Skips a specified number of bytes and returns the new file position.
File.toString()	Returns the path to the File object.
File.write()	Writes data to a file.
File.writeAll()	Takes an array as a parameter and calls the File.writeln() method on each element in the array.
File.writeByte()	Writes a byte to a file.
File.writeln()	Writes data to a file and adds a platform-dependent end-of-line character after outputting the last parameter.

File constructor

fileObject = new File(name)

Creates an instance of the File class.

Availability

Flash Media Server 2

Parameters

name A string indicating the name of the file or directory. The name can contain only UTF-8 encoded characters; high byte values can be encoded by using the URI character-encoding scheme. The specified name is mapped to a system path by using the mappings specified in the FileObject section of the Application.xml file. If the path is invalid, the name property of the object is set to an empty string, and no file operation can be performed.

Returns

A File object if successful; otherwise, null.

Example

The following code creates an instance of the File class:

```
var errorLog = new File("/logs/error.txt");
```

Note that the physical file isn't created on the hard disk until you call File.open().

File.canAppend

fileObject.canAppend

Read only; a boolean value indicating whether a file can be appended (true) or not (false). The property is undefined for closed files.

Availability

Flash Media Server 2.0

File.canRead

fileObject.canRead

Read-only; A boolean value indicating whether a file can be read (true) or not (false).

Availability

Flash Media Server 2

File.canReplace

fileObject.canReplace

Read-only; A boolean value indicating whether a file was opened in "create" mode (true) or not (false). This property is undefined for closed files.

Availability

Flash Media Server 2

File.canWrite

```
fileObject.canWrite
```

Read only; a boolean value indicating whether a file can be written to (`true`) or not (`false`).

Note: *If `File.open()` was called to open the file, the mode in which the file was opened is respected. For example, if the file was opened in read mode, you can read from the file, but you cannot write to the file.*

Availability

Flash Media Server 2

File.close()

```
fileObject.close()
```

Closes the file. This method is called automatically on an open File object when the object is out of scope.

Availability

Flash Media Server 2

Returns

A boolean value indicating whether the file was closed successfully (`true`) or not (`false`). Returns `false` if the file is not open.

Example

The following code closes the /path/file.txt file:

```
if (x.open("/path/file.txt", "read") ){
    // Do something here.
    x.close();
}
```

File.copyTo()

```
fileObject.copyTo(name)
```

Copies a file to a different location or copies it to the same location with a different filename. This method returns `false` if the source file doesn't exist or if the source file is a directory. When this method fails, it invokes the `application.onStatus()` event handler to report errors.

Note: *The user or process owner that the server runs under in the operating system must have adequate write permissions or the call can fail.*

Availability

Flash Media Server 2

Parameters

`name` Specifies the name of the destination file. The name can contain only UTF-8 characters; high byte values can be encoded by using the URI character-encoding scheme. The name specified is mapped to a system path by using the mappings specified in the Application.xml file. If the path is invalid or if the destination file doesn't exist, the operation fails, and the method returns `false`.

Returns

A boolean value indicating whether the file is copied successfully (`true`) or not (`false`).

Example

The following code copies the file set by the `myFileObj` File object to the location provided by the parameter:

```
if (myFileObj.copyTo( "/logs/backup/hello.log")){
    // Do something here.
}
```

File.creationTime

`fileObject.creationTime`

Read-only; a Date object containing the time the file was created.

Availability

Flash Media Server 2

File.eof()

`fileObject.eof()`

Returns a boolean value indicating whether the file pointer is at the end of file (`true`) or not (`false`). If the file is closed, the method returns `true`.

Availability

Flash Media Server 2

Returns

A boolean value.

Example

The following `while` statement lets you insert code that is executed until the file pointer is at the end of a file:

```
while (!myFileObj.eof()){
// Do something here.
}
```

File.exists

`fileObject.exists`

Read-only; a boolean value indicating whether the file or directory exists (`true`) or not (`false`).

Availability

Flash Media Server 2

File.flush()

`fileObject.flush()`

Flushes the output buffers of a file. If the file is closed, the operation fails. When this method fails, it invokes the `application.onStatus()` event handler to report errors.

Availability

Flash Media Server 2

Returns

A boolean value indicating whether the flush operation was successful (`true`) or not (`false`).

File.isDirectory

`fileObject.isDirectory`

Read-only; a boolean value indicating whether the file is a directory (`true`) or not (`false`).

A File object that represents a directory has properties that represent the files contained in the directory. These properties have the same names as the files in the directory, as shown in the following example:

```
myDir = new File("/some/directory");
myFileInDir = myDir.fileName;
trace(myDir.isDirectory) // Outputs true.
```

The following example uses named property lookup to refer to files that do not have valid property names:

```
mySameFileInDir = myDir["fileName"];
myOtherFile = myDir["some long filename with spaces"];
```

Availability

Flash Media Server 2

File.isFile

`fileObject.isFile`

Read-only; a boolean value indicating whether a file is a regular data file (`true`) or not (`false`).

Availability

Flash Media Server 2

File.isOpen

`fileObject.isOpen`

Read-only; a boolean value indicating whether the file has been successfully opened and is still open (`true`) or not (`false`).

Note: *Directories do not need to be opened.*

Availability

Flash Media Server 2

File.lastModified

`fileObject.lastModified`

Read-only; a Date object containing the time the file was last modified.

Availability

Flash Media Server 2

File.length

`fileObject.length`

Read-only; for a directory, the number of files in the directory, not counting the current directory and parent directory entries; for a file, the number of bytes in the file.

Availability

Flash Media Server 2

File.list()

```
fileObject.list(filter)
```

If the file is a directory, lists the files in the directory. Returns an array with an element for each file in the directory.

Availability

Flash Media Server 2

Parameters

`filter` A Function object that determines the files in the returned array.

If the function returns `true` when a file's name is passed to it as a parameter, the file is added to the array returned by `File.list()`. This parameter is optional and allows you to filter the results of the call.

Returns

An Array object.

Example

The following example returns files in the current directory that have 3-character names:

```
var a = x.currentDir.list(function(name){return name.length==3;});
```

File.mkdir()

```
fileObject.mkdir(newDir)
```

Creates a directory. The directory is created in the directory specified by `fileObject`. When this method fails, it invokes the `application.onStatus()` event handler to report errors.

The user or process owner that the server runs under in the operating system must have adequate write permissions or the call can fail.

Note: *You cannot call this method from a File object that is a file (where `isFile` is `true`). You must call this method from a File object that is a directory (where `isDirectory` is `true`).*

Availability

Flash Media Server 2

Parameters

`newDir` A string indicating the name of the new directory. This name is relative to the current File object instance.

Returns

A boolean value indicating success (`true`) or failure (`false`).

Example

The following example creates a logs directory in the `myFileObject` instance:

```
if (myFileObject.mkdir("logs")){
```

```
    // Do something if a logs directory is created successfully.
}
```

File.mode

`fileObject.mode`

Read-only; the mode of an open file. It can be different from the `mode` parameter that was passed to the `open()` method for the file if you have repeating attributes (for example, `"read, read"`) or if some attributes were ignored. If the file is closed, the property is `undefined`.

Availability

Flash Media Server 2

See also

`File.open()`

File.name

`fileObject.name`

Read-only; a string indicating the name of the file. If the File object was created with a invalid path, the value is an empty string.

Availability

Flash Media Server 2

File.open()

`fileObject.open(type, mode)`

Opens a file so that you can read from it or write to it. First use the File constructor to create a File object and then call `open()` on that object. When the `open()` method fails, it invokes the `application.onStatus()` event handler to report errors.

Availability

Flash Media Server 2

Parameters

`type` A string indicating the encoding type for the file. The following types are supported (there is no default value):

Value	Description
`"text"`	Opens the file for text access by using the default file encoding.
`"binary"`	Opens the file for binary access.
`"utf8"`	Opens the file for UTF-8 access.

`mode` A string indicating the mode in which to open the file. The following modes are valid and can be combined (modes are case sensitive and multiple modes must be separated by commas—for example, `"read,write"`; there is no default value):

Value	Description
`"read"`	Opens a file for reading.
`"write"`	Opens a file for writing.
`"readWrite"`	Opens a file for both reading and writing.
`"append"`	Opens a file for writing and positions the file pointer at the end of the file when you attempt to write to the file.
`"create"`	Creates a new file if the file is not present. If a file exists, its contents are destroyed and a new file is created.

Note: *If both `"read"` and `"write"` are set, `"readWrite"` is automatically set. The user or process owner that the server runs under in the operating system must have write permissions to use "create", "append", "readWrite", and "write" modes.*

Returns

A boolean value indicating whether the file opened successfully (`true`) or not (`false`).

Example

The following client-side script creates a connection to an application called file:

```
var nc:NetConnection = new NetConnection();
function traceStatus(info) {
    trace("Level: " + info.level + " Code: " + info.code);
}
nc.onStatus = traceStatus;
nc.connect("rtmp:/file");
```

The following server-side script creates a text file called log.txt and writes text to the file:

```
application.onConnect = function(client){
    this.acceptConnection(client);
    var logFile = new File("log.txt");
    if(!logFile.exists){
        logFile.open("text", "append");
        logFile.write("something", "somethingElse")
    }
};
```

File.position

`fileObject.position`

The current offset in the file. This is the only property of the File class that can be set. Setting this property performs a seek operation on the file. The property is undefined for closed files.

Availability

Flash Media Server 2

File.read()

`fileObject.read(numChars)`

Reads the specified number of characters from a file and returns a string. If the file is opened in binary mode, the operation fails. When this method fails, it invokes the `application.onStatus()` event handler to report errors.

Availability

Flash Media Server 2

Parameters

`numChars` A number specifying the number of characters to read. If `numChars` specifies more bytes than are left in the file, the method reads to the end of the file.

Returns

A string.

Example

The following code opens a text file in read mode and sets variables for the first 100 characters, a line, and a byte:

```
if (myFileObject.open( "text", "read") ){
    strVal = myFileObject.read(100);
    strLine = myFileObject.readln();
    strChar = myFileObject.readByte();
}
```

File.readAll()

`fileObject.readAll()`

Reads the file after the location of the file pointer and returns an Array object with an element for each line of the file. If the file opened in binary mode., the operation fails. When this method fails, it invokes the `application.onStatus()` event handler to report errors.

Availability

Flash Media Server 2

Returns

An Array object.

File.readByte()

`fileObject.readByte()`

Reads the next byte from the file and returns the numeric value of the next byte or -1 if the operation fails. If the file is not opened in binary mode, the operation fails.

Availability

Flash Media Server 2

Returns

A number; either a positive integer or -1.

File.readln()

`fileObject.readln()`

Reads the next line from the file and returns it as a string. The line-separator characters (either \r\n on Windows or \n on Linux) are not included in the string. The character \r is skipped; \n determines the actual end of the line. If the file opened in binary mode, the operation fails.

Availability
Flash Media Server 2

Returns
A string.

File.remove()
`fileObject.remove(recursive)`

Removes the file or directory pointed to by the File object. When this method fails, it invokes the `application.onStatus()` event handler to report errors.

Availability
Flash Media Server 2

Parameters
`recursive` A boolean value specifying whether to do a recursive removal of the directory and all its contents (`true`), or a nonrecursive removal of the directory contents (`false`). If no value is specified, the default value is `false`. If `fileObject` is not a directory, any parameters passed to the `remove()` method are ignored.

Returns
A boolean value indicating whether the file or directory was removed successfully (`true`) or not (`false`). Returns `false` if the file is open, the path points to a root folder, or the directory is not empty.

Example
The following example shows the creation and removal of a file:

```
fileObject = new File("sharedobjects/_definst_/userIDs.fso");
fileObject.remove();
```

File.renameTo()
`fileObject.renameTo(name)`

Moves or renames a file. If the file is open or the directory points to the root directory, the operation fails. When this method fails, it invokes the `application.onStatus()` event handler to report errors.

Availability
Flash Media Server 2

Parameters
`name` The new name for the file or directory. The name can contain only UTF-8-encoded characters; high byte values can be encoded by using the URI character-encoding scheme. The specified name is mapped to a system path by using the mappings specified in the Application.xml file. If the path is invalid or the destination file doesn't exist, the operation fails.

Returns
A boolean value indicating whether the file was successfully renamed or moved (`true`) or not (`false`).

File.seek()
`fileObject.seek(numBytes)`

Skips a specified number of bytes and returns the new file position. This method can accept both positive and negative parameters.

Availability

Flash Media Server 2

Parameters

`numBytes` A number indicating the number of bytes to move the file pointer from the current position.

Returns

If the operation is successful, returns the current position in the file; otherwise, returns -1. If the file is closed, the operation fails and calls `application.onStatus()` to report a warning. The operation returns -1 when called on a directory.

File.toString()

```
fileObject.toString()
```

Returns the path to the File object.

Availability

Flash Media Server 2

Returns

A string.

Example

The following example outputs the path of the File object `myFileObject`:

```
trace(myFileObject.toString());
```

File.type

```
fileObject.type
```

Read-only; a string specifying the type of data or encoding used when a file is opened. The following strings are supported: `"text"`, `"utf8"`, and `"binary"`. This property is undefined for directories and closed files. If the file is opened in `"text"` mode and UTF-8 BOM (Byte Order Mark) is detected, the `type` property is set to `"utf8"`.

Availability

Flash Media Server 2.0

See also

`File.open()`

File.write()

```
fileObject.write(param0, param1,...paramN)
```

Writes data to a file. The `write()` method converts each parameter to a string and then writes it to the file without separators. The file contents are buffered internally. The `File.flush()` method writes the buffer to the file on disk. When this method fails, it invokes the `application.onStatus()` event handler to report errors.

Note: *The user or process owner that the server runs under in the operating system must have write permissions or this call can fail.*

Availability

Flash Media Server 2

Parameters

`param0, param1,...paramN` Parameters to write to the file.

Returns

A boolean value indicating whether the write operation was successful (`true`) or not (`false`).

Example

The following example writes `"Hello world"` at the end of the `myFileObject` text file:

```
if (myFileObject.open( "text", "append") ) {
    myFileObject.write("Hello world");
}
```

File.writeAll()

`fileObject.writeAll(array)`

Takes an Array object as a parameter and calls the `File.writeln()` method on each element in the array. The file contents are buffered internally. The `File.flush()` method writes the buffer to the file on disk.

Note: *The user or process owner that the server runs under in the operating system must have write permissions or this call can fail.*

Availability

Flash Media Server 2

Parameters

`array` An Array object containing all the elements to write to the file.

Returns

A boolean value indicating whether the write operation was successful (`true`) or not (`false`).

File.writeByte()

`fileObject.writeByte(number)`

Writes a byte to a file. The file contents are buffered internally. The `File.flush()` method writes the buffer to the file on disk.

Note: *The user or process owner that the server runs under in the operating system must have write permissions or this call can fail.*

Availability

Flash Media Server 2

Parameters

`number` A number to write.

Returns

A boolean value indicating whether the write operation was successful (`true`) or not (`false`).

Example

The following example writes byte 65 to the end of the `myFileObject` file:

```
if (myFileObject.open("text","append")) {
    myFileObject.writeByte(65);
}
```

File.writeln()

```
fileObject.writeln(param0, param1,...paramN)
```

Writes data to a file and adds a platform-dependent end-of-line character after outputting the last parameter. The file contents are buffered internally. The `File.flush()` method writes the buffer to the file on disk.

Note: *The user or process owner that the server runs under in the operating system must have write permissions or this call can fail.*

Availability

Flash Media Server 2

Parameters

`param0, param1,...paramN` Strings to write to the file.

Returns

A boolean value indicating whether the write operation was successful (`true`) or not (`false`).

Example

The following example opens a text file for writing and writes a line:

```
if (fileObj.open( "text", "append") ) {
    fileObj.writeln("This is a line!");
}
```

LoadVars class

The LoadVars class lets you send all the variables in an object to a specified URL and lets you load all the variables at a specified URL into an object. It also lets you send specific variables, rather than all variables, which can make your application more efficient. You can use the `LoadVars.onLoad()` handler to ensure that your application runs when data is loaded, and not before.

The LoadVars class works much like the XML class; it uses the `load()`, `send()`, and `sendAndLoad()` methods to communicate with a server. The main difference between the LoadVars class and the XML class is that LoadVars transfers ActionScript name-value pairs, rather than an XML Document Object Model (DOM) tree stored in the XML object. The LoadVars class follows the same security restrictions as the XML class.

Availability

Flash Media Server 2

Property summary

Property	Description
LoadVars.contentType	The MIME type sent to the server when you call the LoadVars.send() or LoadVars.sendAndLoad() method.
LoadVars.loaded	A boolean value that indicates whether a LoadVars.load() or LoadVars.sendAndLoad() operation has completed (true) or not (false).

Method summary

Method	Description
LoadVars.addRequestHeader()	Adds or changes HTTP request headers (such as Content-Type or SOAPAction) sent with POST actions.
LoadVars.decode()	Converts the query string to properties of the specified LoadVars object.
LoadVars.getBytesLoaded()	Returns the number of bytes loaded from the last or current LoadVars.send() or LoadVars.sendAndLoad() method call.
LoadVars.getBytesTotal()	Returns the number of total bytes loaded during all LoadVars.send() or LoadVars.sendAndLoad() method calls.
LoadVars.load()	Downloads variables from the specified URL, parses the variable data, and places the resulting variables in the LoadVars object that calls the method.
LoadVars.send()	Sends the variables in the specified object to the specified URL.
LoadVars.sendAndLoad()	Posts the variables in the specified object to the specified URL.
LoadVars.toString()	Returns a string containing all enumerable variables in the specified object, in the MIME content encoding *application/x-www-urlform-encoded*.

Event handler summary

Event handler	Description
LoadVars.onData()	Invoked when data has completely downloaded from the server or when an error occurs while data is downloading from a server.
LoadVars.onHTTPStatus()	Invoked when Flash Media Interactive Server receives an HTTP status code from the server.
LoadVars.onLoad()	Invoked when a LoadVars.send() or LoadVars.sendAndLoad() operation has completed.

LoadVars constructor

```
new LoadVars()
```

Creates a LoadVars object. You can use the methods of the LoadVars object to send and load data.

Availability

Flash Media Server 2

Example

The following example creates a LoadVars object called my_lv:

```
var my_lv = new LoadVars();
```

LoadVars.addRequestHeader()

```
myLoadVars.addRequestHeader(header, headerValue)
```

Adds or changes HTTP request headers (such as Content-Type or SOAPAction) sent with `POST` actions. There are two possible use cases for this method: you can pass two strings, `header` and `headerValue`, or you can pass an array of strings, alternating header names and header values.

If multiple calls are made to set the same header name, each successive value replaces the value set in the previous call.

The following standard HTTP headers cannot be added or changed with this method: Accept-Ranges, Age, Allow, Allowed, Connection, Content-Length, Content-Location, Content-Range, ETag, Host, Last-Modified, Locations, Max-Forwards, Proxy-Authenticate, Proxy-Authorization, Public, Range, Retry-After, Server, TE, Trailer, Transfer-Encoding, Upgrade, URI, Vary, Via, Warning, and WWW-Authenticate.

Availability

Flash Media Server 2

Parameters

`header` A string or an array of strings that represents an HTTP request header name.

`headerValue` A string that represents the value associated with `header`.

Example

The following example adds a custom HTTP header named SOAPAction with a value of `Foo` to the `my_lv` object:

```
var my_lv = new LoadVars();
my_lv.addRequestHeader("SOAPAction", "'Foo'");
```

The following example creates an array named headers that contains two alternating HTTP headers and their associated values. The array is passed as a parameter to the `addRequestHeader()` method.

```
var my_lv = new LoadVars();
var headers = ["Content-Type", "text/plain", "X-ClientAppVersion", "2.0"];
my_lv.addRequestHeader(headers);
```

The following example creates a new LoadVars object that adds a request header called FLASH-UUID. The header contains a variable that the server can check.

```
var my_lv = new LoadVars();
my_lv.addRequestHeader("FLASH-UUID", "41472");
my_lv.name = "Mort";
my_lv.age = 26;
my_lv.send("http://flash-mx.com/mm/cgivars.cfm", "_blank", "POST");
```

LoadVars.contentType

```
myLoadVars.contentType
```

The MIME type sent to the server when you call the `LoadVars.send()` or `LoadVars.sendAndLoad()` method. The default is *application/x-www-urlform-encoded*.

Availability

Flash Media Server 2

Example

The following example creates a LoadVars object and displays the default content type of the data that is sent to the server:

```
application.onConnect = function(client){
    this.acceptConnection(client);
    var my_lv = new LoadVars();
    trace(my_lv.contentType);
};


// Output to Live Log: application/x-www-form-urlencoded
```

LoadVars.decode()

`myLoadVars.decode(queryString)`

Converts the query string to properties of the specified LoadVars object. This method is used internally by the `LoadVars.onData()` event handler. Most users do not need to call this method directly. If you override the `LoadVars.onData()` event handler, you can explicitly call `LoadVars.decode()` to parse a string of variables.

Availability

Flash Media Server 2

Parameters

`queryString` A URL-encoded query string containing name-value pairs.

Example

The following example traces the three variables:

```
application.onConnect = function(client){
    this.acceptConnection(client);
    // Create a new LoadVars object.
    var my_lv = new LoadVars();
    //Convert the variable string to properties.
    my_lv.decode("name=Mort&score=250000");
    trace(my_lv.toString());
    // Iterate over properties in my_lv.
    for (var prop in my_lv) {
trace(prop+" -> "+my_lv[prop]);
    }
};
```

The following is output to the Live Log panel in the Administration Console:

```
name=Mort&score=250000
name -> Mort
score -> 250000
contentType -> application/x-www-form-urlencoded
loaded -> false
```

LoadVars.getBytesLoaded()

`myLoadVars.getByesLoaded()`

Returns the number of bytes loaded from the last or current `LoadVars.load()` or `LoadVars.sendAndLoad()` method call. The value of the `contentType` property does not affect the value of `getBytesLoaded()`.

Availability
Flash Media Server 2

Returns
A number.

See also
`LoadVars.getBytesTotal()`

LoadVars.getBytesTotal()
`myLoadVars.getBytesTotal()`

Returns the total number of bytes loaded into an object during all`LoadVars.load()` or `LoadVars.sendAndLoad()` `LoadVars.load()` or `LoadVars.sendAndLoad()` method calls. Each time a call to `load()` or `sendAndLoad()` is issued, the `getBytesLoaded()` method is reset, but the `getBytesTotal()` method continues to grow.

The value of the `contentType` property does not affect the value of `getBytesLoaded()`.

Availability
Flash Media Server 2

Returns
A number. Returns `undefined` if no load operation is in progress or if a load operation has not been initiated. Returns `undefined` if the number of total bytes can't be determined—for example, if the download was initiated but the server did not transmit an HTTP content length.

See also
`LoadVars.getBytesLoaded()`

LoadVars.load()
`myLoadVars.load(url)`

Downloads variables from the specified URL, parses the variable data, and places the resulting variables into the LoadVars object that calls the method. You can load variables from a remote URL or from a URL in the local file system; the same encoding standards apply to both.

Any properties in the `myLoadVars` object that have the same names as downloaded variables are overwritten. The downloaded data must be in the MIME content type and be *application/x-www-urlform-encoded*.

The `LoadVars.load()` method call is asynchronous.

Availability
Flash Media Server 2

Parameters
`url` A string indicating the URL from which to download variables.

Returns
A boolean value indicating success (`true`) or failure (`false`).

Example

The following code defines an `onLoad()` handler function that signals when data is returned:

```
application.onConnect = function(client){
    this.acceptConnection(client);
    var my_lv = new LoadVars();
    my_lv.onLoad = function(success) {
        if (success) {
            trace(this.toString());
        } else {
            trace("Error loading/parsing LoadVars.");
        }
    };
    my_lv.load("http://www.helpexamples.com/flash/params.txt");
};
```

LoadVars.loaded

`myLoadVars.loaded`

A boolean value that indicates whether a `LoadVars.load()` or `LoadVars.sendAndLoad()` operation has completed (`true`) or not (`false`).

Availability

Flash Media Server 2

Example

The following example loads a text file and writes information to the log file when the operation is complete:

```
var my_lv = new LoadVars();
my_lv.onLoad = function(success) {
trace("LoadVars loaded successfully: "+this.loaded);
};
my_lv.load("http://www.helpexamples.com/flash/params.txt");
```

See also

`LoadVars.onLoad()`

LoadVars.onData()

`myLoadVars.onData(src){}`

Invoked when data has completely downloaded from the server or when an error occurs while data is downloading from a server.

Availability

Flash Media Server 2

Parameters

`src` A string or `undefined`; the raw (unparsed) data from a `LoadVars.load()` or `LoadVars.sendAndLoad()` method call.

Details

This handler is invoked before the data is parsed and can be used to call a custom parsing routine instead of the one built in to Flash Player. The value of the `src` parameter that is passed to the function assigned to `LoadVars.onData()` can be either `undefined` or a string that contains the URL-encoded name-value pairs downloaded from the server. If the `src` parameter is `undefined`, an error occurred while downloading the data from the server.

The default implementation of `LoadVars.onData()` invokes `LoadVars.onLoad()`. You can override this default implementation by assigning a custom function to `LoadVars.onData()`, but `LoadVars.onLoad()` is not called unless you call it in your implementation of `LoadVars.onData()`.

Example

The following example loads a text file and traces the content when the operation is complete:

```
var my_lv = new LoadVars();
my_lv.onData = function(src) {
if (src == undefined) {
trace("Error loading content.");
return;
}
    trace(src);
};
my_lv.load("content.txt", my_lv, "GET");
```

LoadVars.onHTTPStatus()

`myLoadVars.onHTTPStatus(httpStatus){}`

Invoked when Flash Media Interactive Server receives an HTTP status code from the server. This handler lets you capture and act on HTTP status codes.

Availability

Flash Media Server 2

Parameters

`httpStatus` A number; the HTTP status code returned by the server. For example, a value of 404 indicates that the server has not found a match for the requested URI. HTTP status codes can be found in sections 10.4 and 10.5 of the HTTP specification.

Details

The `onHTTPStatus()` handler is invoked before `onData()`, which triggers calls to `onLoad()` with a value of `undefined` if the load fails. After `onHTTPStatus()` is triggered, `onData()` is always triggered, whether or not you override `onHTTPStatus()`. To best use the `onHTTPStatus()` handler, you should write a function to catch the result of the `onHTTPStatus()` call; you can then use the result in your `onData()` and `onLoad()` handlers. If `onHTTPStatus()` is not invoked, this indicates that Flash Media Interactive Server did not try to make the URL request.

If Flash Media Interactive Server cannot get a status code, or if it cannot communicate with the server, the default value of 0 is passed to your ActionScript code.

Example

The following example shows how to use `onHTTPStatus()` to help with debugging. The example collects HTTP status codes and assigns their value and type to an instance of the LoadVars object. (Notice that this example creates the instance members `this.httpStatus` and `this.httpStatusType` at runtime.) The `onData()` handler uses these instance members to trace information about the HTTP response that can be useful in debugging.

```
var myLoadVars = new LoadVars();

myLoadVars.onHTTPStatus = function(httpStatus) {
    this.httpStatus = httpStatus;
    if(httpStatus < 100) {
        this.httpStatusType = "flashError";
    }
    else if(httpStatus < 200) {
        this.httpStatusType = "informational";
    }
    else if(httpStatus < 300) {
        this.httpStatusType = "successful";
    }
    else if(httpStatus < 400) {
        this.httpStatusType = "redirection";
    }
    else if(httpStatus < 500) {
        this.httpStatusType = "clientError";
    }
    else if(httpStatus < 600) {
        this.httpStatusType = "serverError";
    }
}

myLoadVars.onData = function(src) {
    trace(">> " + this.httpStatusType + ": " + this.httpStatus);
    if(src != undefined) {
        this.decode(src);
        this.loaded = true;
        this.onLoad(true);
    }
    else {
        this.onLoad(false);
    }
}

myLoadVars.onLoad = function(success) {}

myLoadVars.load("http://weblogs.macromedia.com/mxna/flashservices/getMostRecentPosts.cfm")
;
```

LoadVars.onLoad()

```
myLoadVars.onLoad(success){}
```

Invoked when a `LoadVars.load()` or `LoadVars.sendAndLoad()` operation has completed. If the variables load successfully, the `success` parameter is `true`. If the variables were not received, or if an error occurred in receiving the response from the server, the `success` parameter is `false`.

If the `success` parameter is `true`, the `myLoadVars` object is populated with variables downloaded by the `LoadVars.load()` or `LoadVars.sendAndLoad()` operation, and these variables are available when the `onLoad()` handler is invoked.

Availability

Flash Media Server 2

Parameters

`success` A boolean value indicating whether the `LoadVars.load()` operation ended in success (`true`) or failure (`false`).

Example

The following example creates a new LoadVars object, attempts to load variables into it from a remote URL, and prints the result:

```
myLoadVars = new LoadVars();
myLoadVars.onLoad = function(result){
    trace("myLoadVars load success is " + result);
}
myLoadVars.load("http://www.someurl.com/somedata.txt");
```

LoadVars.send()

```
myLoadVars.send(url [, target, method])
```

Sends the variables in the `myLoadVars` object to the specified URL. All enumerable variables in the `myLoadVars` object are concatenated into a string that is posted to the URL by using the HTTP `POST` method.

The MIME content type sent in the HTTP request headers is the value of `LoadVars.contentType`.

Availability

Flash Media Server 2

Parameters

`url` A string; the URL to which to upload variables.

`target` A File object. If you use this optional parameter, any returned data is output to the specified File object. If this parameter is omitted, the response is discarded.

`method` A string indicating the `GET` or `POST` method of the HTTP protocol. The default value is `POST`. This parameter is optional.

Returns

A boolean value indicating success (`true`) or failure (`false`).

See also

`LoadVars.sendAndLoad()`

LoadVars.sendAndLoad()

```
myLoadVars.sendAndLoad(url, target[, method ])
```

Posts the variables in the `myLoadVars` object to the specified URL. The server response is downloaded and parsed as variable data, and the resulting variables are placed in the `target` object. Variables are posted in the same way as `LoadVars.send()`. Variables are downloaded into `target` in the same way as `LoadVars.load()`.

Availability

Flash Media Server 2

Parameters

`url` A string; the URL to which to upload variables.

`target` The LoadVars object that receives the downloaded variables.

`method` A string; the `GET` or `POST` method of the HTTP protocol. The default value is `POST`. This parameter is optional.

Returns

A boolean value indicating success (`true`) or failure (`false`).

LoadVars.toString()

```
myLoadVars.toString()
```

Returns a string containing all enumerable variables in `myLoadVars`, in the MIME content encoding *application/x-www-form-urlencoded*.

Availability

Flash Media Server 2

Returns

A string.

Example

The following example instantiates a new `LoadVars()` object, creates two properties, and uses `toString()` to return a string containing both properties in URL-encoded format:

```
var my_lv = new LoadVars();
my_lv.name = "Gary";
my_lv.age = 26;
trace (my_lv.toString());
//output: age=26&name=Gary
```

Log class

The Log class lets you create a Log object that can be passed as an optional parameter to the constructor for the WebService class. For more information, see WebService constructor.

Availability

Flash Media Server 2

Event handler summary

Event handler	Description
`Log.onLog()`	Invoked when a log message is sent to a log.

Log constructor

```
new Log([logLevel][, logName])
```

Creates a Log object that can be passed as an optional parameter to the constructor for the WebService class.

Availability

Flash Media Server 2

Parameters

`logLevel` One of the following values (if not set explicitly, the level defaults to `Log.BRIEF`):

Value	Description
`Log.BRIEF`	Primary life cycle event and error notifications are received.
`Log.VERBOSE`	All life cycle event and error notifications are received.
`Log.DEBUG`	Metrics and fine-grained events and errors are received.

`logName` An optional parameter that can be used to distinguish between multiple logs that are running simultaneously to the same output.

Returns

A Log object.

Example

The following example creates a new instance of the Log class:

```
newLog = new Log();
```

Log.onLog()

```
myLog.onLog(message){}
```

Invoked when a log message is sent to a log.

Availability

Flash Media Server 2

Parameters

`message` A log message.

NetConnection class

The server-side NetConnection class lets you create a two-way connection between a Flash Media Server application instance and an application server, another Flash Media Server, or another Flash Media Server application instance on the same server. You can use NetConnection objects to create powerful applications; for example, you can get weather information from an application server or share an application load with other servers that are running Flash Media Server or application instances.

Availability

Flash Communication Server 1

Property summary

Property	Description
NetConnection.isConnected	Read-only; a boolean value indicating whether a connection has been made.
NetConnection.objectEncoding	The Action Message Format (AMF) version used to pass binary data between two servers.
NetConnection.uri	Read-only; a string indicating the URI parameter of the NetConnection.connect() method.

Method summary

Method	Description
NetConnection.addHeader()	Adds a context header to the Action Message Format (AMF) packet structure.
NetConnection.call()	Invokes a command or method on another Flash Media Server or an application server to which the application instance is connected.
NetConnection.close()	Closes the connection with the server.
NetConnection.connect()	Connects to another Flash Media Server or to a Flash Remoting server such as Adobe ColdFusion.

Event handler summary

Event handler	Description
NetConnection.onStatus()	Invoked every time the status of the NetConnection object changes.

NetConnection constructor

```
new NetConnection()
```

Creates a new instance of the NetConnection class.

Availability

Flash Communication Server 1.

Returns

A NetConnection object.

Example

The following example creates a new instance of the NetConnection class:

```
newNC = new NetConnection();
```

NetConnection.addHeader()

```
nc.addHeader(name, mustUnderstand, object)
```

Adds a context header to the Action Message Format (AMF) packet structure. This header is sent with every future AMF packet. If you call addHeader() by using the same name, the new header replaces the existing header, and the new header persists for the duration of the NetConnection object. You can remove a header by calling addHeader() with the name of the header to remove and an undefined object.

Availability

Flash Communication Server 1

Parameters

`name` A string; identifies the header and the ActionScript object data associated with it.

`mustUnderstand` A boolean; `true` indicates that the server must understand and process this header before it handles any of the following headers or messages.

`object` An Object.

Example

The following example creates a new NetConnection instance, `nc`, and connects to an application at web server www.foo.com that is listening at port 1929. This application dispatches the service /blag/SomeCoolService. The last line of code adds a header called `foo`.

```
nc=new NetConnection();
nc.connect("http://www.foo.com:1929/blag/SomeCoolService");
nc.addHeader("foo", true, new Foo());
```

NetConnection.call()

```
nc.call(methodName, [resultObj [, p1, ..., pN])
```

Invokes a command or method on another Flash Media Server or an application server to which the application instance is connected. The `NetConnection.call()` method on the server works the same way as the `NetConnection.call()` method on the client: it invokes a command on a remote server.

Note: *To call a method on a client from a server, use the `Client.call()` method.*

Availability

Flash Communication Server 1

Parameters

`methodName` A string indicating a method specified in the form `"[objectPath/]method"`. For example, the `someObj/doSomething` command tells the remote server to invoke the `clientObj.someObj.doSomething()` method, with all the `p1, ..., pN` parameters. If the object path is missing, `clientObj.doSomething()` is invoked on the remote server.

`resultObj` An Object. This optional parameter is used to handle return values from the server. The result object can be any object that you defined and can have two defined methods to handle the returned result: `onResult()` and `onStatus()`. If an error is returned as the result, `onStatus()` is invoked; otherwise, `onResult()` is invoked.

`p1, ..., pN` Optional parameters that can be of any ActionScript type, including a reference to another ActionScript object. These parameters are passed to the `methodName` parameter when the method is executed on the remote application server.

Returns

For RTMP connections, returns a boolean value of `true` if a call to `methodName` is sent to the client; otherwise, `false`. For application server connections, it always returns `true`.

Example

The following example uses RTMP to execute a call from one Flash Media Server to another Flash Media Server. The code makes a connection to the App1 application on server 2 and then invokes the `Sum()` method on server 2:

```
nc1.connect("rtmp://server2.mydomain.com/App1", "svr2",);
nc1.call("Sum", new Result(), 3, 6);
```

The following Server-Side ActionScript code is on server 2. When the client is connecting, this code checks to see whether it has a parameter that is equal to `svr1`. If the client has that parameter, the `Sum()` method is defined so that when the method is called from svr1, svr2 can respond with the appropriate method:

```
application.onConnect = function(clientObj){
    if(arg1 == "svr1"){
        clientObj.Sum = function(p1, p2){
            return p1 + p2;
        }
    }
    return true;
};
```

The following example uses an Action Message Format (AMF) request to make a call to an application server. This allows Flash Media Server to connect to an application server and then invoke the `quote()` method. The Java adaptor dispatches the call by using the identifier to the left of the dot as the class name and the identifier to the right of the dot as a method of the class.

```
nc = new NetConnection;
nc.connect("http://www.xyz.com/java");
nc.call("myPackage.quote", new Result());
```

NetConnection.close()

```
nc.close()
```

Closes the connection with the server. After you close the connection, you can reuse the NetConnection instance and reconnect to an old application or connect to a new one.

Note: *The `NetConnection.close()` method has no effect on HTTP connections.*

Availability

Flash Communication Server 1

NetConnection.connect()

```
nc.connect(URI, [p1, ..., pN])
```

Connects to another Flash Media Server or to a Flash Remoting server such as Adobe ColdFusion.

Call `NetConnection.connect()` to connect to an application server running a Flash Remoting gateway over HTTP or to connect to another Flash Media Server for sharing audio, video, and data over one of the following versions of RTMP:

Protocol	Description
RTMP	Real-Time Messaging Protocol
RTMPS	Real-Time Messaging Protocol over SSL

It is good practice to write an `application.onStatus()` callback function and check the `NetConnection.isConnected` property for RTMP connections to see whether a successful connection was made. For Action Message Format (AMF) connections, check `NetConnection.onStatus()`.

Availability

Flash Communication Server 1

Parameters

`URI` A string indicating a URI to connect to. URI has the following format:

```
[protocol://]host[:port]/appName[/instanceName]
```

The following are legal URIs:

```
http://appServer.mydomain.com/webApp
rtmp://rtserver.mydomain.com/realtimeApp
rtmps://rtserver.mydomain.com/secureApp
rtmp://localhost/realtimeApp
rtmp:/realtimeApp
```

`p1, ..., pN` Optional parameters that can be of any ActionScript type, including references to other ActionScript objects. These parameters are sent as connection parameters to the `application.onConnect()` event handler for RTMP connections. For AMF connections to application servers, RTMP parameters are ignored.

Returns

For RTMP connections, a boolean value of `true` for success; otherwise, `false`. For AMF connections to application servers, `true` is always returned.

Example

The following example creates an RTMP connection to an application instance on Flash Media Server:

```
nc = new NetConnection();
nc.connect("rtmp://tc.foo.com/myApp/myConn");
```

NetConnection.isConnected

```
nc.isConnected
```

Read-only; a boolean value indicating whether a connection has been made. It is set to `true` if there is a connection to the server. It's a good idea to check this property value in an `onStatus()` callback function. This property is always `true` for AMF connections to application servers.

Availability

Flash Communication Server 1

Example

The following example uses `NetConnection.isConnected` in an `onStatus()` handler to check whether a connection has been made:

```
nc = new NetConnection();
nc.connect("rtmp://tc.foo.com/myApp");
nc.onStatus = function(infoObj){
    if (info.code == "NetConnection.Connect.Success" && nc.isConnected){
        trace("We are connected");
    }
};
```

NetConnection.objectEncoding

```
nc.objectEncoding
```

The Action Message Format (AMF) version used to pass binary data between two servers. The possible values are 3 (ActionScript 3.0 format) and 0 (ActionScript 1.0 and ActionScript 2.0 format). The default value is 3. When Flash Media Server acts as a client trying to connect to another server, the encoding of the client should match the encoding of the remote server.

The value of `objectEncoding` is determined dynamically according to the following rules when the server receives a `NetConnection.onStatus()` event with the code property `NetConnection.Connect.Success`:

- If the `onStatus() info` object contains an `objectEncoding` property, its value is used.
- If the `onStatus() info` object does not contain an `objectEncoding` property, 0 is assumed even if the connecting server has set `objectEncoding` to 3.
- Once the NetConnection instance is connected, the `objectEncoding` property is read-only.

These rules turn Flash Media Server 3 into an AMF0 client when it connects to a remote Flash Media Server version 2 or earlier (which only support AMF0).

Note: *The server always serializes data in AMF0 while executing Flash Remoting functions.*

Availability

Flash Media Interactive Server 3 and Flash Media Development Server 3

NetConnection.onStatus()

```
nc.onStatus = function(infoObject) {}
```

Invoked every time the status of the NetConnection object changes. For example, if the connection with the server is lost in an RTMP connection, the `NetConnection.isConnected` property is set to `false`, and `NetConnection.onStatus()` is invoked with a status message of `NetConnection.Connect.Closed`. For AMF connections, `NetConnection.onStatus()` is used only to indicate a failed connection. Use this event handler to check for connectivity.

Availability

Flash Communication Server 1

Parameters

`infoObject` An Object with properties that provide information about the status of a NetConnection information object. This parameter is optional, but it is usually used. The NetConnection information object contains the following properties:

Property	Meaning
`code`	A string identifying the event that occurred.
`description`	A string containing detailed information about the code. Not every information object includes this property.
`level`	A string indicating the severity of the event.

The following table contains the `code` and `level` property values and their meanings:

Code	Level	Meaning
NetConnection.Call.Failed	error	The NetConnection.call() method was not able to invoke the server-side method or command.
NetConnection.Connect.AppShutdown	error	The application has been shut down (for example, if the application is out of memory resources and must shut down to prevent the server from crashing) or the server has shut down.
NetConnection.Connect.Closed	status	The connection was closed successfully.
NetConnection.Connect.Failed	error	The connection attempt failed.
NetConnection.Connect.Rejected	error	The client does not have permission to connect to the application, or the application name specified during the connection attempt was not found on the server. This information object also has an application property that contains the value returned by application.rejectConnection().
NetConnection.Connect.Success	status	The connection attempt succeeded.

Example

The following example defines a function for the onStatus() handler that outputs messages to indicate whether the connection was successful:

```
nc = new NetConnection();
nc.onStatus = function(info){
    if (info.code == "NetConnection.Connect.Success") {
        _root.gotoAndStop(2);
    }   else {
        if (! nc.isConnected){
            _root.gotoAndStop(1);
        }
    }
};
```

NetConnection.uri

nc.uri

Read-only; a string indicating the URI parameter of the NetConnection.connect() method. This property is set to null before a call to NetConnection.connect() or after a call to NetConnection.close().

Availability

Flash Communication Server 1

NetStream class

Opens a one-way streaming connection between Flash Media Interactive Server and a remote Flash Media Interactive Server through a NetConnection object. A NetStream object is a channel inside a NetConnection object; call NetStream.publish() to publish data over this channel. Unlike a client-side NetStream object, a server-side NetStream object can only publish data; it cannot subscribe to a publishing stream or play a recorded stream.

Use the NetStream class to scale live broadcasting applications to support more clients. Flash Media Interactive Server can support only a certain number of subscribing clients. To increase that number, you can use the NetStream class to move traffic to remote servers while still maintaining only one client-to-server connection.

The following steps describe the workflow for publishing a stream to a remote Flash Media Interactive Server:

1 Call the NetConnection constructor, `nc = new NetConnection`, to create a NetConnection object.

2 Call `nc.connect("rtmp://serverName/appName/appInstanceName")` to connect to an application on a remote Flash Media Interactive Server.

Note: *You cannot use RTMPT, RTMPE, or RTMPTE when connecting to a remote server.*

3 Call the NetStream constructor, `ns = new NetStream(nc)`, to create a data stream over the connection.

4 Call `ns.publish("myStream")` to give the stream a unique name and send data over the stream to the remote server. You can also record the data as you publish it, so that users can play it back later.

5 Clients that subscribe to this stream connect to the same application on the remote server (in a client-side script), `NetConnection.connect("rtmp://serverName/appName/appInstanceName")`, and then call `NetStream.play("myStream")` with the same stream name.

Availability

Flash Media Interactive Server 3 and Flash Media Development Server 3

Property summary

Property	Description
`NetStream.bufferTime`	Read-only; indicates the number of seconds assigned to the buffer by the `NetStream.setBufferTime()` method.
`NetStream.time`	Read-only; indicates the number of seconds the stream has been publishing.

Method summary

Method	Description
`NetStream.attach()`	Attaches a data source to the NetStream object.
`NetStream.publish()`	Publishes a stream to a remote server.
`NetStream.send()`	Broadcasts a data message over a stream.
`NetStream.setBufferTime()`	Sets the size of the outgoing buffer in seconds.

Event handler summary

Event handler	Description
`NetStream.onStatus()`	Invoked every time a status change or error occurs in a NetStream Object.

NetStream class constructor

```
ns = new NetStream(connection)
```

Creates a stream that can be used for publishing (sending) data through the specified NetConnection object. However, you can create multiple streams that run simultaneously over the same connection.

Availability

Flash Media Interactive Server 3 and Flash Media Development Server 3

Parameters

`connection` A NetConnection object.

Returns

A NetStream object if successful; otherwise, `null`.

Example

```
nc = new NetConnection();
nc.connect("rtmp://xyz.com/myApp");
ns = new NetStream(nc);
```

NetStream.attach()

```
ns.attach(stream)
```

Attaches a data source to the NetStream object.

Availability

Flash Media Interactive Server 3 and Flash Media Development Server 3

Parameters

`stream` A Stream object. If you pass `false`, the attached Stream object detaches from the NetStream object.

Returns

A boolean value. If the attached object is a valid data source, `true`; otherwise, `false`.

Example

```
myStream = Stream.get("foo");
ns = new NetStream(nc);
ns.attach(myStream);
```

NetStream.bufferTime

```
ns.bufferTime
```

Read-only; indicates the number of seconds assigned to the buffer by the `NetStream.setBufferTime()` method.

Availability

Flash Media Interactive Server 3 and Flash Media Development Server 3

NetStream.onStatus()

```
ns.onStatus = function(infoObject){})
```

Invoked every time a status change or error occurs in a NetStream object. The remote server can accept or reject a call to `NetStream.publish()`.

Availability

Flash Media Interactive Server 3 and Flash Media Development Server 3

Parameters

`infoObject` An Object with `code` and `level` properties that provide information about the status of a NetStream call. Both properties are strings.

Code property	Level property	Description
`NetStream.Publish.Start`	`status`	An attempt to publish was successful.
`NetStream.Publish.BadName`	`error`	An attempt was made to publish to a stream that is already being published by someone else.
`NetStream.Unpublish.Success`	`status`	An attempt to stop publishing a stream was successful.
`NetStream.Record.Start`	`status`	Recording was started.
`NetStream.Record.Stop`	`status`	Recording was stopped.
`NetStream.Record.NoAccess`	`status`	An attempt was made to record a read-only stream.
`NetStream.Record.Failed`	`error`	An attempt to record a stream failed.

Example

```
ns = new NetStream(nc);
ns.onStatus = function(info){
    if (info.code == "NetStream.Publish.Start"){
        trace("It is now publishing");
    }
ns.publish("foo", "live");
}
```

NetStream.publish()

```
ns.publish(name, howToPublish)
```

Publishes a stream to a remote server. If the stream has been published by another client, the `publish()` call can fail when it reaches the remote server. Check the status in the `NetStream.onStatus()` handler to make sure that the publisher has been accepted.

Availability

Flash Media Interactive Server 3 and Flash Media Development Server 3

Parameters

`name` A string identifying the stream to publish. If you pass `false`, the stream stops publishing.

`howToPublish` A string specifying how to publish the stream. Valid values are `"record"`, `"append"`, and `"live"`. The default value is `"live"`. This parameter is optional.

If you pass `"record"`, the live data is recorded to a file called *name*.flv. The file is stored on the remote server associated with the NetConnection object. If the file already exists, it is overwritten.

If you pass `"append"`, the live data is appended to a file called *name*.flv. The file is stored on the remote server associated with the NetConnection object. If a file called *name*.flv is not found, it is created.

If you omit this parameter or pass `"live"`, live data is published but not recorded. If a file called *name*.flv exists on the remote server, it is deleted.

Note: *If* name.*flv is read-only, live data is published and* name.*flv is not deleted.*

Example

```
application.onPublish = function(client, myStream){
    nc = new NetConnection();
    nc.connect("rtmp://example.com/myApp");
    ns = new NetStream(nc);
```

```
    ns.attach(myStream);
    ns.publish(myStream.name, "live");
};
```

NetStream.send()

```
ns.send(handlerName, [p1, ..., pN])
```

Broadcasts a data message over a stream.

Availability

Flash Media Interactive Server 3 and Flash Media Development Server 3

Parameters

`handlerName` A string that identifies the name of the handler to receive the message.

`p1, ..., pN` Optional parameters of any type. They are serialized and sent over the connection. The receiving handler receives them in the same order.

Returns

A boolean value; `true` if the data message is dispatched; otherwise, `false`.

Example

The following client-side code broadcasts the message "`Hello world`" to the `foo` handler function on each client that is connected to `myApp`:

```
nc = new NetConnection();
nc.connect("rtmp://xyz.com/myApp");
ns = new NetStream(nc);
ns.send("foo", "Hello world");
```

NetStream.setBufferTime()

```
ns.setBufferTime(bufferTime)
```

Sets the size of the outgoing buffer in seconds. If publishing, it controls the buffer in the local server.

Availability

Flash Media Interactive Server 3 and Flash Media Development Server 3

Parameters

`bufferTime` A number indicating the size of the outgoing buffer in seconds.

Example

```
nc = new NetConnection();
nc.connect("rtmp://xyz.com/myApp");
ns = new NetStream(nc);
ns.setBufferTime(2);
```

NetStream.time

```
ns.time
```

Read-only; indicates the number of seconds the stream has been publishing. This is a good indication of whether data is flowing from the source that has been set in a call to the `NetStream.attach()` method.

Availability
Flash Media Interactive Server 3 and Flash Media Development Server 3

SharedObject class

The SharedObject class lets you store data on the server and share data between multiple client applications in real time. Shared objects can be temporary, or they can persist on the server after an application has closed; you can consider shared objects as real-time data transfer devices.

Note: *This entry explains the server-side SharedObject class. You can also create shared objects with the client-side SharedObject class.*

The following list describes common ways to use shared objects in Server-Side ActionScript:

1 Storing and sharing data on a server. A shared object can store data on the server for other clients to retrieve. For example, you can open a remote shared object, such as a phone list, that is persistent on the server. Whenever a client makes a change to the shared object, the revised data is available to all clients that are currently connected to the object or that connect to it later. If the object is also persistent locally and a client changes the data while not connected to the server, the changes are copied to the remote shared object the next time the client connects to the object.

2 Sharing data in real time. A shared object can share data among multiple clients in real time. For example, you can open a remote shared object that stores real-time data that is visible to all clients connected to the object, such as a list of users connected to a chat room. When a user enters or leaves the chat room, the object is updated and all clients that are connected to the object see the revised list of chat-room users.

It is important to understand the following information about using shared objects in Server-Side ActionScript:

- The Server-Side ActionScript method `SharedObject.get()` creates remote shared objects; there is no server-side method that creates local shared objects. Local shared objects are stored in memory, unless they're persistent, in which case they are stored in .sol files.

- Remote shared objects that are stored on the server have the file extension .fso and are stored in a subdirectory of the application that created them. Remote shared objects on the client have the file extension .sor and are also stored in a subdirectory of the application that created them.

- Server-side shared objects can be *nonpersistent* (that is, they exist for the duration of an application instance) or *persistent* (that is, they are stored on the server after an application closes).

- To create a persistent shared object, set the `persistence` parameter of the `SharedObject.get()` method to `true`. Persistent shared objects let you maintain an application's state.

3 Every remote shared object is identified by a unique name and contains a list of name-value pairs, called *properties,* like any other ActionScript object. A name must be a unique string and a value can be any ActionScript data type.

Note: *Unlike client-side shared objects, server-side shared objects do not have a* `data` *property.*

- To get the value of a server-side shared object property, call `SharedObject.getProperty()`. To set the value of a server-side shared object property, call `SharedObject.setProperty()`.

- To clear a shared object, call the `SharedObject.clear()` method; to delete multiple shared objects, call the `application.clearSharedObjects()` method.

- Server-side shared objects can be owned by the current application instance or by another application instance. The other application instance can be on the same server or on a different server. References to shared objects that are owned by a different application instance are called *proxied shared objects*.

If you write a server-side script that modifies multiple properties, you can prevent other clients from modifying the object during the update by calling the `SharedObject.lock()` method before updating the object. Then you can call `SharedObject.unlock()` to commit the changes and allow other changes to be made. Call `SharedObject.mark()` to deliver `change` events in groups within the `lock()` and `unlock()` methods.

When you get a reference to a proxied shared object, any changes made to the object are sent to the instance that owns the object. The success or failure of any changes is sent by using the `SharedObject.onSync()` event handler, if it is defined.

The `SharedObject.lock()` and `SharedObject.unlock()` methods cannot lock or unlock proxied shared objects.

Availability

Flash Communication Server 1

Property summary

Property	Description
`SharedObject.autoCommit`	A boolean value indicating whether the server periodically stores all persistent shared objects (`true`) or not (`false`).
`SharedObject.isDirty`	Read-only; a boolean value indicating whether the persistent shared object has been modified since the last time it was stored (`true`) or not (`false`).
`SharedObject.name`	Read-only; the name of a shared object.
`SharedObject.resyncDepth`	An integer that indicates when the deleted values of a shared object should be permanently deleted.
`SharedObject.version`	Read-only; the current version number of a shared object.

Method summary

Method	Description
`SharedObject.clear()`	Deletes all the properties of a single shared object and sends a clear event to all clients that subscribe to a persistent shared object.
`SharedObject.close()`	Detaches a reference from a shared object.
`SharedObject.commit()`	Static; stores either a specific persistent shared object instance or all persistent shared object instances with an `isDirty` property whose value is `true`.
`SharedObject.flush()`	Saves the current state of a persistent shared object.
`SharedObject.get()`	Static; creates a shared object or returns a reference to an existing shared object.
`SharedObject.getProperty()`	Retrieves the value of a named property in a shared object.
`SharedObject.getPropertyNames()`	Enumerates all the property names for a given shared object.
`SharedObject.lock()`	Locks a shared object.
`SharedObject.mark()`	Delivers all `change` events to a subscribing client as a single message.
`SharedObject.purge()`	Causes the server to remove all deleted properties that are older than the specified version.
`SharedObject.send()`	Executes a method in a client-side script.

Method	Description
SharedObject.setProperty()	Updates the value of a property in a shared object.
SharedObject.size()	Returns the total number of valid properties in a shared object.
SharedObject.unlock()	Allows other clients to update the shared object.

Event handler summary

Event handler	Description
SharedObject.handlerName()	An event handler invoked when a shared object receives a message with the same name from the client-side SharedObject.send() method.
SharedObject.onStatus()	Invoked when errors, warnings, and status messages associated with either a local instance of a shared object or a persistent shared object occur.
SharedObject.onSync()	Invoked when a shared object changes.

SharedObject.autoCommit

so.autoCommit

A boolean value indicating whether the server periodically stores all persistent shared objects (true) or not (false). If autoCommit is false, the application must call SharedObject.commit() to save the shared object; otherwise, the data is lost.

This property is true by default. To override the default, specify the initial state by using the following configuration key in the Application.xml file, as shown in the following example:

```
<SharedObjManager>
    <AutoCommit>false</AutoCommit>
</SharedObjManager>
```

Availability

Flash Media Server 2

SharedObject.clear()

so.clear()

Deletes all the properties of a single shared object and sends a clear event to all clients that subscribe to a persistent shared object. The persistent data object is also removed from a persistent shared object.

Availability

Flash Communication Server 1

Returns

Returns true if successful; otherwise, false.

See also

application.clearSharedObjects()

SharedObject.close()

so.close()

Detaches a reference from a shared object. A call to the `SharedObject.get()` method returns a reference to a shared object instance. The reference is valid until the variable that holds the reference is no longer in use and the script is garbage collected. To destroy a reference immediately, you can call `SharedObject.close()`. You can use `SharedObject.close()` when you no longer want to proxy a shared object.

Availability

Flash Communication Server 1

Example

In the following example, `so` is attached as a reference to shared object `foo`. When you call `so.close()`, you detach the reference `so` from the shared object `foo`.

```
so = SharedObject.get("foo");
    // Insert code here.
so.close();
```

See also

`SharedObject.get()`

SharedObject.commit()

`so.commit([name])`

Static; stores either a specific persistent shared object instance or all persistent shared object instances with an `isDirty` property whose value is `true`. Use this method if the `SharedObject.autoCommit` property is `false` and you need to manage when a shared object is stored locally.

Availability

Flash Media Server 2

Parameters

`name` A string indicating the name of the persistent shared object instance to store. If no name is specified, or if an empty string is passed, all persistent shared objects are stored. This parameter is optional.

Returns

A boolean value indicating success (`true`) or failure (`false`).

Example

The following code commits all dirty shared objects to local storage when the application stops:

```
application.onAppStop = function (info){
    // Insert code here.
    SharedObject.commit();
}
```

SharedObject.flush()

`so.flush()`

Saves the current state of a persistent shared object. Invokes the `SharedObject.onStatus()` handler and passes it an object that contains information about the success or failure of the call.

Availability

Flash Communication Server 1

Returns

A boolean value of `true` if successful; otherwise, `false`.

Example

The following example places a reference to the shared object `foo` in the variable `so`. It then locks the shared object instance so that no one can make any changes to it and saves the shared object by calling `so.flush()`. After the shared object is saved, it is unlocked so that further changes can be made.

```
var so = SharedObject.get("foo", true);
so.lock();
// Insert code here that operates on the shared object.
so.flush();
so.unlock();
```

SharedObject.get()

```
SharedObject.get(name, persistence [, netConnection])
```

Static; creates a shared object or returns a reference to an existing shared object. To perform any operation on a shared object, the server-side script must get a reference to the shared object by using the `SharedObject.get()` method. If the requested object is not found, a new instance is created.

Availability

Flash Communication Server 1

Parameters

`name` Name of the shared object instance to return.

`persistence` A boolean value: `true` for a persistent shared object; `false` for a nonpersistent shared object. If no value is specified, the default value is `false`.

`netConnection` A NetConnection object that represents a connection to an application instance. You can pass this parameter to get a reference to a shared object on another server or a shared object that is owned by another application instance. All update notifications for the shared object specified by the `name` parameter are proxied to this instance, and the remote instance notifies the local instance when a persistent shared object changes. The NetConnection object that is used as the `netConnection` parameter does not need to be connected when you call `SharedObject.get()`. The server connects to the remote shared object when the NetConnection state changes to connected. This parameter is optional.

Returns

A reference to an instance of the SharedObject class.

Details

There are two types of shared objects, persistent and nonpersistent, and they have separate namespaces. This means that a persistent and a nonpersistent shared object can have the same name and exist as two distinct shared objects. Shared objects are scoped to the namespace of the application instance and are identified by a string. The shared object names should conform to the URI specification.

You can also call `SharedObject.get()` to get a reference to a shared object that is in a namespace of another application instance. This instance can be on the same server or on a different server and is called a *proxied shared object*. To get a reference to a shared object from another instance, create a NetConnection object and use the `NetConnection.connect()` method to connect to the application instance that owns the shared object. Pass the NetConnection object as the `netConnection` parameter of the `SharedObject.get()` method. The server-side script must get a reference to a proxied shared object before there is a request for the shared object from any client. To do this, call `SharedObject.get()` in the `application.onAppStart()` handler.

If you call `SharedObject.get()` with a `netConnection` parameter and the local application instance already has a shared object with the same name, the shared object is converted to a proxied shared object. All shared object messages for clients that are connected to a proxied shared object are sent to the master instance.

If the connection state of the NetConnection object that was used as the `netConnection` parameter changes state from connected to disconnected, the proxied shared object is set to idle and any messages received from subscribers are discarded. The `NetConnection.onStatus()` handler is called when a connection is lost. You can then reestablish a connection to the remote instance and call `SharedObject.get()`, which changes the state of the proxied shared object from idle to connected.

If you call `SharedObject.get()` with a new NetConnection object on a proxied shared object that is already connected, and if the URI of the new NetConnection object doesn't match the current NetConnection object, the proxied shared object unsubscribes from the previous shared object, sends a `clear` event to all subscribers, and subscribes to the new shared object instance. When a subscribe operation to a proxied shared object is successful, all subscribers are reinitialized to the new state. This process lets you migrate a shared object from one application instance to another without disconnecting the clients.

Updates received by proxied shared objects from subscribers are checked to see if the update can be rejected based on the current state of the proxied shared object version and the version of the subscriber. If the change can be rejected, the proxied shared object doesn't forward the message to the remote instance; the reject message is sent to the subscriber.

The corresponding client-side ActionScript method is `SharedObject.getRemote()`.

Example

The following example creates a shared object named `foo` in the function `onProcessCmd()`. The function is passed a parameter, `cmd`, that is assigned to a property in the shared object.

```
function onProcessCmd(cmd){
    // Insert code here.
    var shObj = SharedObject.get("foo", true);
    propName = cmd.name;
    shObj.getProperty (propName, cmd.newAddress);
}
```

The following example uses a proxied shared object. A proxied shared object resides on a server or in an application instance (called *master*) that is different from the server or application instance that the client connects to (called *proxy*). When the client connects to the proxy and gets a remote shared object, the proxy connects to the master and gives the client a reference to this shared object. The following code is in the main.asc file:

```
application.appStart = function() {
 nc = new NetConnection();
 nc.connect("rtmp://" + master_server + "/" + master_instance);
 proxySO = SharedObject.get("myProxy",true,nc);
        // Now, whenever the client asks for a persistent
        // shared object called myProxy, it receives themyProxy
        // shared object from master_server/master_instance.
};
```

SharedObject.getProperty()

```
so.getProperty(name)
```

Retrieves the value of a named property in a shared object. The returned value is a copy associated with the property, and any changes made to the returned value do not update the shared object. To update a property, use the `SharedObject.setProperty()` method.

Availability

Flash Communication Server 1

Parameters

`name` A string indicating the name of a property in a shared object.

Returns

The value of a SharedObject property. If the property doesn't exist, returns `null`.

Example

The following example gets the value of the `name` property on the `user` shared object and assigns it to the `firstName` variable:

```
firstName = user.getProperty("name");
```

See also

`SharedObject.setProperty()`

SharedObject.getPropertyNames()

```
so.getPropertyNames()
```

Enumerates all the property names for a given shared object.

Availability

Flash Communication Server 1

Returns

An array of strings that contain all the property names of a shared object.

Example

The following example calls `getPropertyNames()` on the `myInfo` shared object and places the names in the `names` variable. It then enumerates those property names in a `for` loop.

```
myInfo = SharedObject.get("foo");
var addr = myInfo.getProperty("address");
myInfo.setProperty("city", San Francisco");
var names = myInfo.getPropertyNames();
for (x in names){
    var propVal = myInfo.getProperty(names[x]);
    trace("Value of property " + names[x] + " = " + propVal);
}
```

SharedObject.handlerName()

```
so.handlerName = function([p1,..., pN]){}
```

An event handler invoked when a shared object receives a message with the same name from the client-side `SharedObject.send()` method. You must define a Function object and assign it to the event handler.

The `this` keyword used in the body of the function is set to the shared object instance returned by `SharedObject.get()`.

If you don't want the server to receive a particular message, do not define this handler.

Availability

Flash Communication Server 1

Parameters

`p1, ..., pN` Optional parameters passed to the handler method if the message contains user-defined parameters. These parameters are the user-defined objects that are passed to the `SharedObject.send()` method.

Returns

Any return value is ignored by the server.

Example

The following example defines an event handler called `traceArgs`:

```
var so = SharedObject.get("userList", false);
so.traceArgs = function(msg1, msg2){
    trace(msg1 + " : " + msg2);
};
```

SharedObject.isDirty

`so.isDirty`

Read-only; a boolean value indicating whether a persistent shared object has been modified since the last time it was stored (`true`) or not (`false`). The `SharedObject.commit()` method stores shared objects with an `isDirty` property that is `true`.

This property is always `false` for nonpersistent shared objects.

Availability

Flash Media Server 2

Example

The following example saves the `so` shared object if it has been changed:

```
var so = SharedObject.get("foo", true);
if (so.isDirty){
    SharedObject.commit(so.name);
}
```

SharedObject.lock()

`so.lock()`

Locks a shared object. This method gives the server-side script exclusive access to the shared object; when the `SharedObject.unlock()` method is called, all changes are batched and one update message is sent through the `SharedObject.onSync()` handler to all the clients that subscribe to this shared object. If you nest the `SharedObject.lock()` and `SharedObject.unlock()` methods, make sure that there is an `unlock()` method for every `lock()` method; otherwise, clients are blocked from accessing the shared object.

You cannot use the `SharedObject.lock()` method on proxied shared objects.

Availability

Flash Communication Server 1

Returns

An integer indicating the lock count: 0 or greater indicates success; -1 indicates failure. For proxied shared objects, always returns -1.

Example

The following example locks the `so` shared object, executes the code that is to be inserted, and then unlocks the object:

```
var so = SharedObject.get("foo");
so.lock();
// Insert code here that operates on the shared object.
so.unlock();
```

SharedObject.mark()

```
so.mark(handlerName, p1, ..., pN)
```

Delivers all `change` events to a subscribing client as a single message.

In a server-side script, you can call the `SharedObject.setProperty()` method to update multiple shared object properties between a call to the `lock()` and `unlock()` methods. All subscribing clients receive a `change` event notification through the `SharedObject.onSync()` handler. However, because the server may collapse multiple messages to optimize bandwidth, the `change` event notifications may not be sent in the same order as they were in the code.

Use the `mark()` method to execute code after all the properties in a set have been updated. You can call the `handlerName` parameter passed to the `mark()` method, knowing that all property changes before the `mark()` call have been updated.

Availability

Flash Media Server 2

Parameters

`handlerName` Calls the specified handler on the client-side SharedObject instance. For example, if the `handlerName` parameter is `onChange`, the client invokes the `SharedObject.onChange()` handler with all the `p1, ...,pN` parameters.

Note: *Do not use a built-in method name for a handler name. For example, if the handler name is `close`, the subscribing stream will be closed.*

`p1, ..., pN` Parameters of any ActionScript type, including references to other ActionScript objects. Parameters are passed to `handlerName` when it is executed on the client.

Returns

A boolean value. Returns `true` if the message can be dispatched to the client; otherwise, `false`.

Example

The following example calls the `mark()` method twice to group two sets of shared object property updates for clients:

```
var myShared = SharedObject.get("foo", true);
```

```
myShared.lock();
myShared.setProperty("name", "Stephen");
myShared.setProperty("address", "Xyz lane");
myShared.setProperty("city", "SF");
myShared.mark("onAdrChange", "name");
myShared.setProperty("account", 12345);
myShared.mark("onActChange");
myShared.unlock();
```

The following example shows the receiving client-side script:

```
connection = new NetConnection();
connection.connect("rtmp://flashmediaserver/someApp");
var x = SharedObject.get( "foo", connection.uri, true);
x.connect(connection);
x.onAdrChange = function(str) {
    // Shared object has been updated,
    // can look at the "name", "address" and "city" now.
}

x.onActChange = function(str) {
    // Shared object has been updated,
    // can look at the "account" property now,
}
```

SharedObject.name

`so.name`

Read-only; the name of a shared object.

Availability

Flash Communication Server 1

SharedObject.onStatus()

`so.onStatus = function(info) {}`

Invoked when errors, warnings, and status messages associated with either a local instance of a shared object or a persistent shared object occur.

Availability

Flash Communication Server 1

Parameters

`info` An information object.

Example

The following client-side code defines an anonymous function that just traces the `level` and `code` properties of the specified shared object:

```
so = SharedObject.get("foo", true);
so.onStatus = function(infoObj){
    //Handle status messages passed in infoObj.
    trace(infoObj.level + "; " + infoObj.code);
};
```

SharedObject.onSync()

```
so.onSync = function(list){}
```

Invoked when a shared object changes. Use the `onSync()` handler to define a function that handles changes made to a shared object by subscribers.

For proxied shared objects, defines the function to get the status of changes made by the server and other subscribers.

Note: *You cannot define the `onSync()` handler on the `prototype` property of the SharedObject class in Server-Side ActionScript.*

Availability

Flash Communication Server 1

Parameters

`list` An array of objects that contain information about the properties of a shared object that have changed since the last time the `onSync()` handler was called. The notifications for proxied shared objects are different from the notifications for shared objects that are owned by the local application instance. The following table describes the codes for local shared objects:

Local code	Meaning
`change`	A property was changed by a subscriber.
`delete`	A property was deleted by a subscriber.
`name`	The name of a property that has changed or been deleted.
`oldValue`	The old value of a property. This is true for both `change` and `delete` messages; on the client, `oldValue` is not set for `delete`.

Note: *Changing or deleting a property on the server side by using the `SharedObject.setProperty()` method always succeeds, so there is no notification of these changes.*

The following table describes the codes for local shared objects:

Proxied code	Meaning
`success`	A server change of the shared object was accepted.
`reject`	A server change of the shared object was rejected. The value on the remote instance was not changed.
`change`	A property was changed by another subscriber.
`delete`	A property was deleted. This notification can occur when a server deletes a shared object or if another subscriber deletes a property.
`clear`	All the properties of a shared object are deleted. This can happen when the server's shared object is out of sync with the master shared object or when the persistent shared object migrates from one instance to another. This event is typically followed by a `change` message to restore all of the server's shared object properties.
`name`	The name of a property that has changed or been deleted.
`oldValue`	The old value of the property. This is valid only for the reject, change, and delete codes.

Note: *The `SharedObject.onSync()` handler is invoked when a shared object has been successfully synchronized with the server. If there is no change in the shared object, the list object may be empty.*

Example

The following example creates a function that is invoked whenever a property of the shared object `so` changes:

```
// Create a new NetConnection object.
nc = new NetConnection();
nc.connect("rtmp://server1.xyx.com/myApp");
// Create the shared object.
so = SharedObject.get("MasterUserList", true, nc);
// The list parameter is an array of objects containing information
// about successfully or unsuccessfully changed properties
// from the last time onSync() was called.
so.onSync = function(list) {
    for (var i = 0; i < list.length; i++) {
        switch (list[i].code ) {
            case "success":
                trace ("success");
                break;
            case "change":
                trace ("change");
                break;
            case "reject":
                trace ("reject");
                break;
            case "delete":
                trace ("delete");
                break;
            case "clear":
                trace ("clear");
                break;
        }
    }
};
```

SharedObject.purge()

`so.purge(version)`

Causes the server to remove all deleted properties that are older than the specified version. Although you can also accomplish this task by setting the `SharedObject.resyncDepth` property, the `purge()` method gives the script more control over which properties to delete.

Availability

Flash Communication Server 1

Parameters

`version` A number indicating the version. All deleted data that is older than this version is removed.

Returns

A boolean value.

Example

The following example deletes all the properties of the `so` shared object that are older than the value of `so.version - 3`:

```
var so = SharedObject.get("foo", true);
so.lock();
so.purge(so.version - 3);
so.unlock();
```

SharedObject.resyncDepth

```
so.resyncDepth
```

An integer that indicates when the deleted properties of a shared object should be permanently deleted. You can use this property in a server-side script to resynchronize shared objects and to control when shared objects are deleted. The default value is infinity.

If the current revision number of the shared object minus the revision number of the deleted property is greater than the value of `SharedObject.resyncDepth`, the property is deleted. Also, if a client connecting to this shared object has a client revision that, when added to the value of `SharedObject.resyncDepth`, is less than the value of the current revision on the server, all the current elements of the client shared object are deleted, the valid properties are sent to the client, and the client receives a "clear" message.

This method is useful when you add and delete many properties and you don't want to send too many messages to the client. Suppose that a client is connected to a shared object that has 12 properties and then disconnects. After that client disconnects, other clients that are connected to the shared object delete 20 properties and add 10 properties. When the client reconnects, it could, for example, receive a delete message for the 10 properties it previously had and then a change message on two properties. You can use `SharedObject.resyncDepth` property to send a "clear" message, followed by a change message for two properties, which saves the client from receiving 10 delete messages.

Availability

Flash Communication Server 1

Example

The following example resynchronizes the shared object `so` if the revision number difference is greater than 10:

```
so = SharedObject.get("foo");
so.resyncDepth = 10;
```

SharedObject.send()

```
so.send(methodName, [p1, ..., pN])
```

Executes a method in a client-side script. You can use `SharedObject.send()` to asynchronously execute a method on all the Flash clients subscribing to a shared object. The server does not receive any notification from the client on the success, failure, or return value in response to this message.

Availability

Flash Communication Server 1

Parameters

`methodName` A string indicating the name of a method on a client-side shared object. For example, if you specify `"doSomething"`, the client must invoke the `SharedObject.doSomething()` method, with all the `p1, ..., pN` parameters.

`p1, ..., pN` Parameters of any type, including references to other objects. These parameters are passed to the specified `methodName` on the client.

Returns

A boolean value of `true` if the message was sent to the client; otherwise, `false`.

Example

The following example calls the `SharedObject.send()` method to invoke the `doSomething()` method on the client and passes the string `"This is a test"`:

```
var so = SharedObject.get("foo", true);
so.send("doSomething", "This is a test");
```

The following example is the client-side ActionScript code that defines the `doSomething()` method:

```
nc = new NetConnection();
nc.connect("rtmp://www.adobe.com/someApp");
var so = SharedObject.getRemote("foo", nc.uri, true);
so.connect(nc);
so.doSomething = function(str) {
// Process the str object.
};
```

SharedObject.setProperty()

```
so.setProperty(name, value)
```

Updates the value of a property in a shared object.

The `name` parameter on the server side is the same as an attribute of the `data` property on the client side. For example, the following two lines of code are equivalent; the first line is Server-Side ActionScript and the second is client-side ActionScript:

```
so.setProperty(nameVal, "foo");
clientSO.data[nameVal] = "foo";
```

A shared object property can be modified by a client between successive calls to `SharedObject.getProperty()` and `SharedObject.setProperty()`. If you want to preserve transactional integrity, call the `SharedObject.lock()` method before modifying the shared object; be sure to call `SharedObject.unlock()` when you finish making modifications. If you call `SharedObject.setProperty()` without first calling `SharedObject.lock()`, the change is made to the shared object, and all object subscribers are notified before `SharedObject.setProperty()` returns. If you call `SharedObject.lock()` before you call `SharedObject.setProperty()`, all changes are batched and sent when the `SharedObject.unlock()` method is called. The `SharedObject.onSync()` handler on the client side is invoked when the local copy of the shared object is updated.

Note: *If only one source (whether client or server) is updating a shared object in a server-side script, you don't need to use the `lock()` or `unlock()` method or the `onSync()` handler.*

Availability

Flash Communication Server 1

Parameters

`name` The name of the property in the shared object.

`value` An ActionScript object associated with the property, or `null` to delete the property.

Example

The following example uses the `SharedObject.setProperty()` method to create the `city` property with the value `San Francisco`. It then enumerates all the property values in a `for` loop and calls `trace()` to display the values.

```
myInfo = SharedObject.get("foo");
var addr = myInfo.getProperty("address");
```

```
myInfo.setProperty("city", "San Francisco");
var names = sharedInfo.getPropertyNames();
for (x in names){
    var propVal = sharedInfo.getProperty(names[x]);
    trace("Value of property " + names[x] + " = " + propVal);
}
```

See also

`SharedObject.getProperty()`

SharedObject.size()

`so.size()`

Returns the total number of valid properties in a shared object.

Availability

Flash Communication Server 1

Returns

An integer indicating the number of properties.

Example

The following example gets the number of properties of a shared object and assigns that number to the variable `len`:

```
var so = SharedObject.get("foo", true);
var soLength = so.size();
```

SharedObject.unlock()

`so.unlock()`

Allows other clients to update the shared object. A call to this method also causes the server to commit all changes made after the `SharedObject.lock()` method is called and sends an update message to all clients.

You cannot call the `SharedObject.unlock()` method on proxied shared objects.

Availability

Flash Communication Server 1

Returns

An integer indicating the lock count: 0 or greater if successful; -1 otherwise. For proxied shared objects, this method always returns -1.

Example

The following example unlocks a shared object:

```
var so = SharedObject.get("foo", true);
so.lock();
// Insert code to manipulate the shared object.
so.unlock();
```

See also

`SharedObject.lock()`

SharedObject.version

`so.version`

Read-only; the current version number of the shared object. Calls to the `SharedObject.setProperty()` method on either the client or the server increment the value of the `version` property.

Availability

Flash Communication Server 1

SOAPCall class

Availability

Flash Media Server 2

The SOAPCall class is the object type that is returned from all web service calls. These objects are typically constructed automatically when a Web Service Definition Language (WSDL) is parsed and a stub is generated.

Property summary

Property	Description
`SOAPCall.request`	An XML object that represents the current SOAP (Simple Object Access Protocol) request.
`SOAPCall.response`	An XML object that represents the most recent SOAP response.

Event handler summary

Event handler	Description
`SOAPCall.onFault()`	Invoked when a method has failed and returned an error.
`SOAPCall.onResult()`	Invoked when a method has successfully invoked and returned.

SOAPCall.onFault()

`SOAPCall.onFault(fault)`

Invoked when a method has failed and returned an error.

Availability

Flash Media Server 2

Parameters

`fault` The `fault` parameter is an object version of an XML SOAP Fault (see SOAPCall class).

SOAPCall.onResult()

`mySOAPCall.onResult(result){}`

Invoked when a method has been successfully invoked and returned.

Availability

Flash Media Server 2

Parameters

`result` The decoded ActionScript object returned by the operation (if any). To get the raw XML returned instead of the decoded result, access the `SOAPCall.response` property.

SOAPCall.request

`mySOAPCall.request`

An XML object that represents the current Simple Object Access Protocol (SOAP) request.

Availability

Flash Media Server 2

SOAPCall.response

`mySOAPCall.response`

An XML object that represents the most recent SOAP response.

Availability

Flash Media Server 2

SOAPFault class

The SOAPFault class is the object type of the error object returned to the `WebService.onFault()` and `SOAPCall.onFault()` functions. This object is returned as the result of a failure and is an ActionScript mapping of the SOAP Fault XML type.

Availability

Flash Media Server 2

Property summary

Property	Description
`SOAPFault.detail`	A string indicating the application-specific information associated with the error, such as a stack trace or other information returned by the web service engine.
`SOAPFault.faultactor`	A string indicating the source of the fault.
`SOAPFault.faultcode`	A string indicating the short, standard qualified name describing the error.
`SOAPFault.faultstring`	A string indicating the human-readable description of the error.

SOAPFault.detail

`mySOAPFault.detail`

A string indicating the application-specific information associated with the error, such as a stack trace or other information returned by the web service engine.

Availability

Flash Media Server 2

SOAPFault.faultactor

```
mySOAPFault.faultactor
```

A string indicating the source of the fault. This property is optional if an intermediary is not involved.

Availability

Flash Media Server 2

SOAPFault.faultcode

```
mySOAPFault.faultcode
```

A string indicating the short, standard qualified name describing the error.

Availability

Flash Media Server 2

SOAPFault.faultstring

```
mySOAPFault.faultstring
```

A string indicating the human-readable description of the error.

Availability

Flash Media Server 2

Example

The following example shows the fault code in a text field if the WSDL fails to load:

```
// Load the WebServices class:
load("webservices/WebServices.asc");

// Prepare the WSDL location:
var wsdlURI = "http://www.flash-db.com/services/ws/companyInfo.wsdl";

// Instantiate the web service object by using the WSDL location:
stockService = new WebService(wsdlURI);

// Handle the WSDL parsing and web service instantiation event:
stockService.onLoad = function(wsdl){
    wsdlField.text = wsdl;
}

// If the wsdl fails to load, the onFault event is fired:
stockService.onFault = function(fault){
    wsdlField.text = fault.faultstring;
}
```

Stream class

The Stream class lets you manage or republish streams in a Flash Media Server application. You can't attach audio or video sources to a Stream object; you can only play and manage existing streams. Use the Stream class to shuffle existing streams in a playlist, pull streams from other servers, control access to streams, and record data streams such as log files.

A stream is a one-way connection between a client running Flash Player and a server running Flash Media Server, or between two servers running Flash Media Server. You can create a stream in Server-Side ActionScript by calling `Stream.get()`. A client can access multiple streams at the same time, and there can be hundreds or thousands of Stream objects active at the same time.

Streams can contain ActionScript data. Call the `Stream.send()` method to add data to a stream. You can extract this data without waiting for a stream to play in real time, such as when you're creating a log file. You can also use it to add metadata to a stream.

Availability

Flash Communication Server 1

Property summary

Property (read-only)	Description
`Stream.bufferTime`	Read-only; indicates how long to buffer messages before a stream is played, in seconds.
`Stream.name`	Read-only; contains a unique string associated with a live stream.
`Stream.syncWrite`	A boolean value that controls when a stream writes the contents of the buffer to a file when the stream is recording.

Method summary

Method	Description
`Stream.clear()`	Deletes a recorded FLV file from the server.
`Stream.flush()`	Flushes a stream.
`Stream.get()`	Static; returns a reference to a Stream object.
`Stream.getOnMetaData()`	Returns an object containing the metadata for the named stream or video file.
`Stream.length()`	Static; returns the length of a recorded stream in seconds.
`Stream.play()`	Controls the data source of a stream with an optional start time, duration, and reset flag to flush any previously playing stream.
`Stream.record()`	Records all the data passing through a Stream object and creates an FLV file of the recorded stream.
`Stream.send()`	Invokes a remote method on a client-side NetStream object subscribing to the stream and passes it parameters of any ActionScript data type.
`Stream.setBufferTime()`	Sets the length of the message queue.
`Stream.setVirtualPath()`	Sets the virtual directory path for video stream playback.
`Stream.size()`	Static; returns the size of a recorded stream in bytes.

Event handler summary

Event handler	Description
`Stream.onStatus()`	Invoked every time the status of a Stream object changes.

Stream.bufferTime

```
myStream.bufferTime
```

Read-only; indicates how long to buffer messages before a stream plays, in seconds. This property applies only when playing a stream from a remote server or when playing a recorded stream locally. Call `Stream.setBufferTime()` to set the `bufferTime` property.

A message is data that is sent back and forth between Flash Media Server and Flash Player. The data is divided into small packets (messages), and each message has a type (audio, video, or data).

Availability

Flash Communication Server 1

Stream.clear()

```
myStream.clear()
```

Deletes a recorded FLV file from the server.

Availability

Flash Communication Server 1

Returns

A boolean value of `true` if the call succeeds; otherwise, `false`.

Example

The following example deletes a recorded stream called playlist.flv. Before the stream is deleted, the example defines an `onStatus()` handler that uses two information object error codes, `NetStream.Clear.Success` and `NetStream.Clear.Failed`, to send status messages to the application log file and the Live Log panel in the Administration Console.

```
s = Stream.get("playlist");
if (s){
    s.onStatus = function(info){
        if(info.code == "NetStream.Clear.Success"){
            trace("Stream cleared successfully.");
        }
        if(info.code == "NetStream.Clear.Failed"){
            trace("Failed to clear stream.");
        }
    };
    s.clear();
}
```

Stream.flush()

```
myStream.flush()
```

Flushes a stream. If the stream is used for recording, the `flush()` method writes the contents of the buffer associated with the stream to the recorded file.

It is highly recommended that you call `flush()` on a stream that contains only data. Synchronization problems can occur if you call the `flush()` method on a stream that contains data and either audio, video, or both.

Important: *H.264 data and AAC data is not copied to a recorded video file.*

Availability

Flash Media Server 2

Returns

A boolean value of `true` if the buffer was successfully flushed; otherwise, `false`.

Example

The following example flushes the `myStream` stream:

```
// Set up the server stream.
application.videoStream = Stream.get("aVideo");

if (application.videoStream){
    application.videoStream.record();
    application.videoStream.send("test", "hello world");
    application.videoStream.flush();
}
```

Stream.get()

```
Stream.get(name)
```

Static; returns a reference to a Stream object. If the requested object is not found, a new instance is created.

You can publish streams only in FLV format; `mp3:`, `mp4:`, and `id3:` are not supported in the stream name for the `Stream.get()` method.

Availability

Flash Communication Server 1

Parameters

`name` A string indicating the name of the stream instance to return.

Returns

A Stream object if the call is successful; otherwise, `null`.

Examples

The following example gets the stream `myVideo` and assigns it to the variable `playStream`. It then calls the `Stream.play()` method from `playStream`.

```
var playStream = Stream.get("videos");
playStream.play("file1", 0, -1);
```

In the following example, the value of `playStream` is `null` because this method doesn't support MP3 files:

```
var playStream = Stream.get("mp3:foo");
```

Stream.getOnMetaData()

```
Stream.getOnMetaData(name)
```

Returns an object containing the metadata for the named stream or video file. The object contains one property for each metadata item. The Flash Video Exporter utility (version 1.1 or later) embeds a video's duration, creation date, data rates, and other information into the video file. This method is currently supported only with FLV files.

Availability

Flash Media Server 2

Parameter

`name` A string indicating the name of a recorded stream, such as "myVideo". The name can be passed in either of the following forms: "myVideo" or "flv:myVideo".

Returns

An Object containing the metadata as properties.

Example

The following example lists the properties and values for the metadata for the recorded stream `myVideo.flv`:

```
var infoObject = Stream.getOnMetaData("myVideo");

trace("Metadata for myVideo.flv:");

for( i in infoObject ){
    trace( i + " = " + infoObject[i] );
}
```

Stream.length()

```
Stream.length(name[, virtualKey])
```

Static; returns the length of a recorded file in seconds. If the requested file is not found, the return value is 0.

Availability

Flash Communication Server 1

Parameters

`name` A string indicating the name of a recorded stream. To get the length of an MP3 file, precede the name of the file with `mp3:` (for example, `"mp3:beethoven"`).

`virtualKey` A string indicating a key value. Starting with Flash Media Server 2, stream names are not always unique; you can create multiple streams with the same name, place them in different physical directories, and use the `VirtualDirectory` section and `VirtualKeys` section of the Vhost.xml file to direct clients to the appropriate stream. Because the `Stream.length()` method is not associated with a client, but connects to a stream on the server, you may need to specify a virtual key to identify the correct stream. For more information about keys, see `Client.virtualKey`. This parameter is optional.

Returns

A number.

Example

The following example gets the length of the recorded stream file `myVideo` and assigns it to the variable `streamLen`:

```
function onProcessCmd(cmd){
    var streamLen = Stream.length("myVideo");
    trace("Length: " + streamLen + "\n");
}
```

The following example gets the length of the MP3 file `beethoven.mp3` and assigns it to the variable `streamLen`:

```
function onProcessCmd(cmd){
    var streamLen = Stream.length("mp3:beethoven");
    trace("Length: " + streamLen + "\n");
}
```

The following example gets the length of the MP4 file `beethoven.mp4` and assigns it to the variable `streamLen`:

```
function onProcessCmd(cmd){
    var streamLen = Stream.length("mp4:beethoven");
    trace("Length: " + streamLen + "\n");
}
```

Stream.name

```
myStream.name
```

Read-only; contains a unique string associated with a live stream. You can use this property as an index to find a stream within an application.

Availability

Flash Communication Server 1

Example

The following function takes a Stream object as a parameter and returns the name of the stream:

```
function getStreamName(myStream){
    return myStream.name;
}
```

Stream.onStatus()

```
myStream.onStatus = function([infoObject]) {}
```

Invoked every time the status of a Stream object changes. For example, if you play a file in a stream, `Stream.onStatus()` is invoked. Use `Stream.onStatus()` to check when play starts and ends, when recording starts, and so on.

Availability

Flash Communication Server 1

Parameters

`infoObject` An Object with `code` and `level` properties that contain information about a stream. This parameter is optional, but it is usually used. The Stream information object contains the following properties:

Property	Meaning
`clientid`	A unique number identifying each client.
`code`	A string identifying the event that occurred.
`description`	Detailed information about the code. Not every information object includes this property.
`details`	The stream name.
`level`	A string indicating the severity of the event.

The following table describes the `code` and `level` property values:

Code property	Level property	Description
NetStream.Clear.Failed	error	A call to application.clearStreams() failed to delete a stream.
NetStream.Clear.Success	status	A call to application.clearStreams() successfully deleted a stream.
NetStream.Failed	error	An attempt to use a Stream method failed.
NetStream.Play.Failed	error	An call to Stream.play() failed.
NetStream.Play.InsufficientBW	warning	Data is playing behind the normal speed.
NetStream.Play.Start	status	Play was started.
NetStream.Play.StreamNotFound	error	An attempt was made to play a stream that does not exist.
NetStream.Play.Stop	status	Play was stopped.
NetStream.Play.Reset	status	A playlist was reset.
NetStream.Play.PublishNotify	status	The initial publish operation to a stream was successful. This message is sent to all subscribers.
NetStream.Play.UnpublishNotify	status	An unpublish operation from a stream was successful. This message is sent to all subscribers.
NetStream.Publish.BadName	error	An attempt was made to publish a stream that is already being published by someone else.
NetStream.Publish.Start	status	Publishing was started.
NetStream.Record.Failed	error	An attempt to record a stream failed.
NetStream.Record.NoAccess	error	An attempt was made to record a read-only stream.
NetStream.Record.Start	status	Recording was started.
NetStream.Record.Stop	status	Recording was stopped.
NetStream.Unpublish.Success	status	A stream has stopped publishing.

Example

The following server-side code attempts to delete a given stream and traces the resulting return code:

```
Client.prototype.delStream = function(streamName){
    trace("*** deleting stream: " + streamName);
    s = Stream.get(streamName);
    if (s) {
        s.onStatus = function(info){
            if (info.code == "NetStream.Clear.Success"){
                trace("*** Stream " + streamName + "deleted.");
            }
            if (info.code == "NetStream.Clear.Failure"){
                trace("*** Failure to delete stream " + streamName);
            }
        };
        s.clear();
    }
}
```

Stream.play()

```
myStream.play(streamName, [startTime, length, reset, remoteConnection, virtualKey])
```

Controls the data source of a stream with an optional start time, duration, and reset flag to flush any previously playing stream. Call `play()` to do the following:

- Chain streams between servers.
- Create a hub to switch between live streams and recorded streams.
- Combine steams into a recorded stream.

You can combine multiple streams to create a playlist for clients. The `Stream.play()` method behaves a bit differently from the `NetStream.play()` method on the client side. A server-side call to `Stream.play()` is similar to a client-side call to `NetStream.publish()`; it controls the source of data coming into a stream. When you call `Stream.play()` on the server, the server becomes the publisher. Because the server has higher priority than the client, the client is forced to unpublish from the stream if the server calls a `play()` method on the same stream.

If any recorded streams are included in a server playlist, you cannot play the server playlist stream as a live stream.

Note: *A stream that plays from a remote server by means of the NetConnection object is considered a live stream.*

To delete a Stream object, use the `delete` operator to mark the stream for deletion. The script engine deletes the object during its garbage collection routine.

```
// Initialize the Stream object.
s = stream.get("foo");
// Play the stream.
s.play("name", p1, ... pN);
// Stop the stream.
s.play(false);
// Mark the Stream object for deletion during server garbage routine.
delete s;
```

Availability

Flash Communication Server 1

Parameters

`streamName` A string indicating the name of any published live stream, recorded stream, MP3 file, or MP4 file.

To play video files, specify the name of the stream without a file extension (for example, `"bolero"`). To play back MP3 or ID3 tags, you must precede the stream name with `mp3:` or `id3:` (for example, `"mp3:bolero"` or `"id3:bolero"`). To play H.264/AAC files, you must precede the stream name with `mp4:`. For example, to play the file file1.m4v, specify `"mp4:file1.m4v"`.

Note: *For H.264 media files, specify the full file name, including the file extension.*

`startTime` A number indicating the start time of the stream playback, in seconds. If no value is specified, it is set to -2. If `startTime` is -2, the server tries to play a live stream with the name specified in `streamName`. If no live stream is available, the server tries to play a recorded stream with the name specified in `streamName`. If no recorded stream is found, the server creates a live stream with the name specified in `streamName` and waits for someone to publish to that stream. If `startTime` is -1, the server attempts to play a live stream with the name specified in `streamName` and waits for a publisher if no specified live stream is available. If `startTime` is greater than or equal to 0, the server plays the recorded stream with the name specified in `streamName`, starting from the time given. If no recorded stream is found, the `play()` method is ignored. If a negative value other than -1 is specified, the server interprets it as -2. This parameter is optional.

`length` A number indicating the length of play, in seconds. For a live stream, a value of -1 plays the stream as long as the stream exists. Any positive value plays the stream for the corresponding number of seconds. For a recorded stream, a value of -1 plays the entire file, and a value of 0 returns the first video frame. Any positive number plays the stream for the corresponding number of seconds. By default, the value is -1. This parameter is optional.

`reset` A boolean value, or number, that flushes the playing stream. If `reset` is `false` (0), the server maintains a playlist, and each call to `Stream.play()` is appended to the end of the playlist so that the next play does not start until the previous play finishes. You can use this technique to create a dynamic playlist. If `reset` is `true` (1), any playing stream stops, and the playlist is reset. By default, the value is `true`.

You can also specify a number value of 2 or 3 for the `reset` parameter, which is useful when playing recorded stream files that contain message data. These values are analogous to `false` (0) and `true` (1), respectively: a value of 2 maintains a playlist, and a value of 3 resets the playlist. However, the difference is that specifying either 2 or 3 for `reset` returns all messages in the specified recorded stream at once, rather than at the intervals at which the messages were originally recorded (the default behavior).

`remoteConnection` A NetConnection object that is used to connect to a remote server. If this parameter is provided, the requested stream plays from the remote server. This is an optional parameter.

`virtualKey` A string indicating a key value. Starting with Flash Media Server 2, stream names are not always unique; you can create multiple streams with the same name, place them in different physical directories, and use the `VirtualDirectory` section and `VirtualKeys` section of the Vhost.xml file to direct clients to the appropriate stream. Because the `Stream.length()` method is not associated with a client, but connects to a stream on the server, you may need to specify a virtual key to identify the correct stream. For more information about keys, see `Client.virtualKey`. This is an optional parameter.

Returns

A boolean value: `true` if the call is accepted by the server; otherwise, `false`. If the server fails to find the stream, or if an error occurs, the `Stream.play()` method can fail. To get information about the `Stream.play()` method, define a `Stream.onStatus()` handler.

If the `streamName` parameter is `false`, the stream stops playing. A boolean value of `true` is returned if the stop succeeds; otherwise, `false`.

Example

The following example shows how streams can be chained between servers:

```
application.myRemoteConn = new NetConnection();
application.myRemoteConn.onStatus = function(info){
    trace("Connection to remote server status " + info.code + "\n");
    // Tell all the clients.
    for (var i = 0; i < application.clients.length; i++){
        application.clients[i].call("onServerStatus", null,
        info.code, info.description);
    }
};
// Use the NetConnection object to connect to a remote server.
application.myRemoteConn.connect(rtmp://movie.com/movieApp);
// Set up the server stream.
application.myStream = Stream.get("foo");
if (application.myStream){
    application.myStream.play("Movie1", 0, -1, true, application.myRemoteConn);
}
```

The following example shows how to use `Stream.play()` as a hub to switch between live streams and recorded streams:

```
// Set up the server stream.
application.myStream = Stream.get("foo");
if (application.myStream){
    // This server stream plays "Live1",
    // "Record1", and "Live2" for 5 seconds each.
    application.myStream.play("Live1", -1, 5);
    application.myStream.play("Record1", 0, 5, false);
    application.myStream.play("Live2", -1, 5, false);
}
```

The following example combines different streams into a recorded stream:

```
// Set up the server stream.
application.myStream = Stream.get("foo");
if (application.myStream){
    // Like the previous example, this server stream
    // plays "Live1", "Record1", and "Live2"
    // for 5 seconds each. But this time,
    // all the data will be recorded to a recorded stream "foo".
    application.myStream.record();
    application.myStream.play("Live1", -1, 5);
    application.myStream.play("Record1", 0, 5, false);
    application.myStream.play("Live2", -1, 5, false);
}
```

The following example calls `Stream.play()` to stop playing the stream `foo`:

```
application.myStream.play(false);
```

The following example creates a playlist of three MP3 files (`beethoven.mp3`, `mozart.mp3`, and `chopin.mp3`) and plays each file in turn over the live stream `foo`:

```
application.myStream = Stream.get("foo");
if(application.myStream) {
    application.myStream.play("mp3:beethoven", 0);
    application.myStream.play("mp3:mozart", 0, false);
    application.myStream.play("mp3:chopin.mp3", 0, false);
    application.myStream.play("mp4:file1.mp4", -1, 5, false);
}
```

The following example plays an MP4 file:

```
application.myStream = Stream.get("foo");
if(application.myStream) {
    application.myStream.play("mp4:beethoven", 0);
    application.myStream.play("mp4:mozart", 0, false);
}
```

In the following example, data messages in the recorded stream file `log.flv` are returned at the intervals at which they were originally recorded:

```
application.myStream = Stream.get("data");
if (application.myStream) {
    application.myStream.play("log", 0, -1);
}
```

In the following example, data messages in the recorded stream file `log.flv` are returned all at once, rather than at the intervals at which they were originally recorded:

```
application.myStream = Stream.get("data");
if (application.myStream) {
    application.myStream.play("log", 0, -1, 2);
}
```

A server-side stream cannot subscribe to itself. For example, the following code is invalid:

```
// Client-side code
var ns = new NetStream
ns.publish("TestStream");

// Server-side code
st = Stream.get("TestStream");
st.play("TestStream");
```

Stream.record()

```
myStream.record(flag)
```

Records all the data passing through a Stream object and creates an FLV file of the recorded stream.

Note: *The `Stream.record()` method saves all streams as FLV files, even if the stream contains FLV, MP3, and MP4 content. H.264 data and AAC data is not copied to a recorded FLV file.*

When you record a stream, the server creates an FLV file and stores it in the streams subdirectory of the application folder. The server creates the streams directory and subdirectories for each application instance name. If a stream isn't associated with an application instance, it is stored in a subdirectory called _definst_ (default instance).

For example, a stream from the default lecture application instance would be stored here: applications\lectures\streams_definst_. A stream from the monday lectures application instance would be stored here: applications\lectures\streams\monday.

Note: *The server creates these directories automatically; you don't have to create one for each instance name.*

Availability

Flash Communication Server 1

Parameters

`flag` One of the these values: "`record`", "`append`", or `false`. If the value is "`record`", the data file is overwritten if it exists. If the value is "`append`", the incoming data is appended to the end of the existing file. If the value is `false`, any previous recording stops. By default, the value is "`record`".

Returns

A boolean value of `true` if the recording succeeds; otherwise, `false`.

Example

The following example opens a stream `s` and, when it is open, plays `smith` and records it. Because no value is passed to the `record()` method, the default value, `record`, is passed.

```
// Start recording.
s = Stream.get("SurfVideos");
if (s){
    s.play("smith");
    s.record();
}
// Stop recording.
s = Stream.get("SurfVideos");
if (s){
    s.record(false);
}
```

Stream.send()

```
myStream.send(handlerName, [p1, ..., pN])
```

Invokes a remote method on a client-side NetStream object subscribing to the stream and passes it parameters of any ActionScript data type. The server does not receive a response object, and any values returned by the client-side method are discarded.

You can call `Stream.send()` to send data over to clients subscribing to a stream. The data is passed in the `p1, . . . , pN` parameters to the `handlerName` method, which is defined on the subscribing stream. Publishing streams do not receive remote method calls, even if they define a method called `handlerName()`.

You can call `Stream.send()` to send metadata to clients subscribing to a live stream in a data keyframe. When a client subscribes to a live stream after it starts playing, the client may not receive the stream's metadata. This metadata can contain any information about the stream that you want the client to know, such as the length, the name of the speaker, and the location of the broadcast.

A data keyframe is a special data message that can be added to a live stream and stored in the memory of the server. The data keyframe is retrieved when a client subscribes to the stream. There are two reserved values that tell the server to set or clear a data keyframe: `@setDataFrame` and `@clearDataFrame`. Like other data messages, a data keyframe contains a handler name and a list of parameters. Use the following syntax to set or clear a data keyframe:

```
Stream.send("@setDataFrame", handlerName [, p1, p2, ..., pN ]);
```

You can send multiple data keyframes for each live stream. However, the handler name of the data keyframe must be unique. Only the stream's publisher and the server are allowed to set and clear data keyframes. You can call the client-side ActionScript `NetStream.send()` method or the Server-Side ActionScript `Stream.send()` method to set a data keyframe in a stream. Setting data keyframes is supported in Flash Media Interactive Server 3 and Flash Media Development Server 3 and later.

Note: *The server does not need to take ownership of a stream from the client in order to send a message. After `send()` is called, the client still owns the stream as a publisher. This is different from how the `Stream.play()` method behaves.*

Availability

Flash Communication Server 1

Parameters

`handlerName` A string indicating the remote method to call on the client. The `handlerName` value is the name of a method relative to the subscribing Stream object. For example, if `handlerName` is `doSomething`, the `doSomething` method at the stream level is invoked with all the `p1, . . . , pN` parameters. Unlike the method names in `Client.call()` and `NetConnection.call()`, the handler name can be only one level deep (that is, it cannot have the form object/method).

Note: *Do not use a built-in method name for a handler name. For example, if the handler name is `close`, the subscribing stream will close.*

`p1, . . . , pN` Parameters of any ActionScript type, including references to other ActionScript objects. These parameters are passed to the specified handler when it is executed on the Flash client.

Returns

A boolean value of `true` if the message was sent to the client; otherwise, `false`.

Example

The following example calls the `onMsg()` method on the client-side NetStream object and sends it the string `"Hello World"`:

```
s = Stream.get("testStream");
s.send("onMsg", "Hello World");
```

The following client-side ActionScript defines the method that handles the data passed on the `testStream` stream:

```
ns = new NetStream(nc);
ns.onMsg = function(str) {
    trace(str); //"Hello World" is output
}
ns.play("testStream", -2, -1, 3);
```

The following example adds metadata to a live stream:

```
s = new Stream(nc);
s.onStatus = function(info){
    if (info.code == "NetStream.Publish.Start"){
        metaData = new Object();
        metaData.title = "myStream";
        metaData.width = 400;
        metaData.height = 200;
        this.send("@setDataFrame", "onMetaData", metaData);
    }
};
s.publish("myStream");
```

Stream.setBufferTime()

```
myStream.setBufferTime()
```

Sets the length of the message queue. When you play a stream from a remote server, the `Stream.setBufferTime()` method sends a message to the remote server that adjusts the length of the message queue. The default length of the message queue is 0 seconds. You should set the buffer time higher when playing a high-quality recorded stream over a low-bandwidth network.

When a user clicks a seek button in an application, buffered packets are sent to the server. The buffered seeking in a Flash Media Server application occurs on the server; Flash Media Server doesn't support client-side buffering. The seek time can be smaller or larger than the buffer size, and it has no direct relationship to the buffer size. Every time the server receives a seek request from Flash Player, it clears the message queue on the server. The server tries to seek to the desired position and starts filling the queue again. At the same time, Flash Player also clears its own buffer after a seek request, and the buffer is eventually filled after the server starts sending the new messages.

Availability

Flash Communication Server 1

Stream.setVirtualPath()

```
myStream.setVirtualPath(virtualPath, directory, virtualKey)
```

Sets the virtual directory path for video stream playback. Maps a virtual directory path to a physical directory and assigns that mapping to a virtual key. The virtual key designates a range of Flash Player versions. These mappings let you use the same URL to serve different versions of streams to clients based on the Flash Player version.

First, create a mapping between Flash Player versions and virtual keys in the `VirtualKeys` section of the Vhost.xml file. When Flash Player requests a stream from Flash Media Interactive Server, the Flash Player version is mapped to a virtual key based on the values that you set in the Vhost.xml file, as in this example:

```
<VirtualKeys>
    <!-- Create your own ranges and key values.-->
    <!-- You can create as many Key elements as you need.-->
```

```
    <Key from="WIN 8,0,0,0" to="WIN 9,0,59,0">A</Key>
    <Key from="WIN 6,0,0,0" to="WIN 7,0,55,0">B</Key>
</VirtualKeys>
```

Next, in the `VirtualDirectory` section of the Vhost.xml file, map the virtual keys to a virtual path and a physical directory, which are separated by a semicolon (for example, `foo;c:\streams`). To set up several virtual directories for different Flash Player versions, use the same virtual path with different physical directories for each `Streams` tag, as shown in this example:

```
<VirtualDirectory>
    <Streams key="A">foo;c:\streams\on2</Streams>
    <Streams key="B">foo;c:\streams\sorenson</Streams>
</VirtualDirectory>
```

Flash Media Interactive Server serves the client a stream from whichever virtual directory the virtual key is mapped to. For example, if the client is Flash Player 8 and the call is `myNetStream.play("foo/familyVideo")`, the `Streams` element with key A would be used and the client would be served the higher-quality stream c:\streams\on2\familyVideo.flv. If the client is Flash Player 7, the same URL maps to the sorenson stream directory and the c:\streams\sorenson\familyVideo.flv file plays.

It is most common to change the values of the `VirtualKeys` and `VirtualDirectory` elements in the Vhost.xml file. However, you can call `Stream.setVirtualPath()` to create `Streams` elements and you can use `Client.virtualKey` to set a client's `Key` value.

For more information about the Vhost.xml file, see *Adobe Flash Media Server Configuration and Administration Guide*.

Availability

Flash Media Server 2

Parameters

`virtualPath` A string indicating the virtual directory path of a stream. If the stream is not located in the virtual path, the default virtual directory path is searched.

`directory` A string indicating the physical directory in which to store streams.

`virtualKey` A string that sets or removes the key value for each Streams entry.

Note: *To indicate a slash in the `virtualPath` and `directory` parameters, you must use a forward slash (/) or a double backslash (\\). In strings, single backslashes are used to escape characters. A double backslash is the escape sequence for a backslash character.*

Example

The following code sets the virtual key to `B`, the virtual path to `foo`, and the physical directory to c:\streams\on2:

```
Stream.setVirtualPath("foo", "c:/streams/on2", "B");
```

Stream.size()

```
Stream.size(name[, virtualKey])
```

Static; returns the size of a recorded stream in bytes.

Availability

Flash Media Server 2

Parameters

`name` A string indicating the name of a stream. You can use the format tag in the `name` parameter to specify the type.

`virtualKey` A string indicating a key value. Starting with Flash Media Server 2, stream names are not always unique; you can create multiple streams with the same name, place them in different physical directories, and use the `VirtualDirectory` section and `VirtualKeys` section of the Vhost.xml file to direct clients to the appropriate stream. Because the `Stream.size()` method is not associated with a client, but connects to a stream on the server, you may need to specify a virtual key to identify the correct stream. For more information about keys, see `Client.virtualKey`. This parameter is optional.

Returns

A number; if the requested stream is not found, returns 0.

Example

The following examples display the size of a stream and an MP3 stream, respectively:

```
function onProcessCmd(cmd){
    // Insert code here...
    var streamSize = Stream.size("foo");
    trace("Size: " + streamSize + "\n");
}

//For mp3

function onProcessCmd(cmd){
    // Insert code here...
    var streamSize = Stream.size("mp3:foo" );
    trace("Size: " + streamSize + "\n");
}

//For mp4

function onProcessCmd(cmd){
    // Insert code here...
    var streamSize = Stream.size("mp4:foo" );
    trace("Size: " + streamSize + "\n");
}
```

Stream.syncWrite

`myStream.syncWrite`

A boolean value that controls when a stream writes the contents of the buffer to a file as the stream is recording. When `syncWrite` is `true`, all the messages that pass through the stream are flushed to the file immediately. It is highly recommended that you set `syncWrite` to `true` only in a stream that contains only data. Synchronization problems may occur if `syncWrite` is set to `true` in a stream that contains data and audio, video, or some combination.

Availability

Flash Media Server 2

Example

The following example flushes data immediately to the file:

```
// Assume foo is a data-only stream.
application.myStream = Stream.get("foo");
```

```
if (application.myStream){
    application.myStream.syncWrite = true;
    application.myStream.record();
    application.myStream.send("test", "hello world");
}
```

WebService class

Availability

Flash Media Server 2

Description

You can use the WebService class to create and access a WSDL/SOAP web service. Several classes comprise the Flash Media Interactive Server web services feature: WebService class, SOAPFault class, SOAPCall class, and Log class.

Note: *The WebService class is not able to retrieve complex data or an array returned by a web service. Also, the WebService class does not support security features.*

The following steps outline the process of creating and accessing a web service.

Create and access a web service:

1 Load the WebServices class:

```
load("webservices/WebServices.asc");
```

2 Prepare the WSDL location:

```
var wsdlURI = "http://www.flash-db.com/services/ws/companyInfo.wsdl";
```

3 Instantiate the web service object by using the WSDL location:

```
stockService = new WebService(wsdlURI);
```

4 (Optional) Handle the WSDL parsing and web service instantiation event through the `WebService.onLoad()` handler:

```
// Handle the WSDL loading event.
stockService.onLoad = function(wsdl){
    wsdlField.text = wsdl;
}
```

5 (Optional) If the WSDL doesn't load, handle the fault:

```
// If the WSDL fails to load, the onFault event is fired.
stockService.onFault = function(fault){
    wsdlField.text = fault.faultstring;
}
```

6 (Optional) Set the SOAP headers:

```
// If headers are required, they are added as follows:
var myHeader = new XML(headerSource);
stockService.addHeader(myHeader);
```

7 Invoke a web service operation:

```
// Method invocations return an asynchronous callback.
callback = stockService.doCompanyInfo("anyuser", "anypassword", "ADBE");
// NOTE: callback is undefined if the service itself is not created
```

```
// (and service.onFault is also invoked).
```

8 Handle either the output or the error fault returned from the invocation:

```
// Handle a successful result.
callback.onResult = function(result){
    // Receive the SOAP output, which in this case
    // is deserialized as a struct (ActionScript object).
    for (var i in result){
        trace(i +" : " +result[i]);
    }
}
// Handle an error result.
callback.onFault = function(fault){
    // Catch the SOAP fault and handle it
    // according to this application's requirements.
    for (var i in fault){
        trace(i +" : " +fault[i]);
    }
}
```

Event handler summary

Event handler	Description
WebService.onFault()	Invoked when an error occurs during WSDL parsing.
WebService.onLoad()	Invoked when the web service has successfully loaded and parsed its WSDL file.

WebService constructor

```
new WebService(wsdlURI)
```

Creates a new WebService object. You must use the constructor to create a WebService object before you call any of the WebService class methods.

Availability

Flash Media Server 2

Parameters

`wsdlURI` A string specifying the URI of a WSDL.

Returns

A WebService object.

Example

The following example prepares the WSDL location and passes it to the WebService constructor to create a new WebService object, `stockService`:

```
load("webservices/WebServices.asc");
var wsdlURI = "http://www.flash-db.com/services/ws/companyInfo.wsdl";
stockService = new WebService(wsdlURI);
```

WebService.onFault()

```
myWS.onFault(fault){}
```

Invoked when an error occurs during WSDL parsing. The web services features convert parsing and network problems into SOAP faults for simple handling.

Availability

Flash Media Server 2

Parameters

`fault` An object version of an XML SOAP fault (see SOAPFault class).

Example

The following example displays the fault code in a text field if the WSDL fails to load and the `onFault()` event fires:

```
// Load the WebServices class:
load("webservices/WebServices.asc");

// Prepare the WSDL location:
var wsdlURI = "http://www.flash-db.com/services/ws/companyInfo.wsdl";

// Instantiate the web service object by using the WSDL location:
stockService = new WebService(wsdlURI);

// Handle the WSDL parsing and web service instantiation event:
stockService.onLoad = function(wsdl){
    wsdlField.text = wsdl;
}

// If the WSDL fails to load, the onFault event is fired:
stockService.onFault = function(fault){
    wsdlField.text = fault.faultstring;
}
```

WebService.onLoad()

`myWS.onLoad(wsdldocument)`

Invoked when the web service has successfully loaded and parsed its WSDL file. Operations can be invoked in an application before this event occurs; when this happens, they are queued internally and are not actually transmitted until the WSDL has loaded.

Availability

Flash Media Server 2

Parameters

`wsdldocument` A WSDL XML document.

Example

In the following example, the `onLoad` event is used to handle the WSDL parsing:

```
// Load the WebServices class:
load("webservices/WebServices.asc");

// Prepare the WSDL location:
var wsdlURI = "http://www.flash-db.com/services/ws/companyInfo.wsdl";

// Instantiate the web service object by using the WSDL location:
stockService = new WebService(wsdlURI);

// Handle the WSDL parsing and web service instantiation event:
stockService.onLoad = function(wsdl){
    wsdlField.text = wsdl;
```

```
}
```

XML class

The XML class lets you load, parse, send, build, and manipulate XML document trees.

Note: *You can load XML files only over HTTP, not over RTMP.*

You must use the `new XML()` constructor to create an XML object before calling any method of the XML class.

An XML document is represented by the XML class. Each element of the document is represented by an XMLNode object.

The XML and XMLNode objects are modeled after the W3C DOM Level 1 Recommendation. That recommendation specifies a Node interface and a Document interface. The Document interface inherits from the Node interface, and adds methods such as `createElement()` and `createTextNode()`. In ActionScript, the XML and XMLNode objects are designed to divide functionality along similar lines.

Note: *Many code examples for the XML class include `trace()` statements. Server-side `trace()` statements are output to the application log file and to the Live Log panel in the Administration Console.*

Availability

Flash Media Server 2

Property summary

Property	Description
`XML.attributes`	An object that contains all the attributes of the specified XML object.
`XML.childNodes`	Read-only; an array of the specified XML object's children.
`XML.contentType`	The MIME content type that is sent to the server when you call the `XML.send()` or `XML.sendAndLoad()` method.
`XML.docTypeDecl`	Specifies information about the XML document's DOCTYPE declaration.
`XML.firstChild`	Read-only; evaluates the specified XML object and references the first child in the parent node's child list.
`XML.ignoreWhite`	When set to `true`, discards, during the parsing process, text nodes that contain only white space.
`XML.lastChild`	Read-only; an XMLNode value that references the last child in the node's child list.
`XML.loaded`	A boolean value; `true` if the document-loading process initiated by the `XML.load()` call completed successfully; otherwise, `false`.
`XML.localName`	Read-only; the local name portion of the XML node's name.
`XML.namespaceURI`	Read-only; if the XML node has a prefix, the value of the `xmlns` declaration for that prefix (the URI), which is typically called the namespace URI.
`XML.nextSibling`	Read-only; an XMLNode value that references the next sibling in the parent node's child list.
`XML.nodeName`	A string representing the node name of the XML object.
`XML.nodeType`	Read-only; a `nodeType` value, either 1 for an XML element or 3 for a text node.
`XML.nodeValue`	The node value of the XML object.

Property	Description
XML.parentNode	Read-only; an XMLNode value that references the parent node of the specified XML object or returns null if the node has no parent.
XML.prefix	Read-only; the prefix portion of the XML node name.
XML.previousSibling	Read-only; an XMLNode value that references the previous sibling in the parent node's child list.
XML.status	A number indicating whether an XML document was successfully parsed into an XML object.
XML.xmlDecl	Specifies information about a document's XML declaration.

Method summary

Method	Description
XML.addRequestHeader()()	Adds or changes HTTP request headers (such as Content-Type or SOAPAction) that are sent with POST actions.
XML.appendChild()	Appends the specified node to the XML object's child list.
XML.cloneNode()	Constructs and returns a new XMLNode object of the same type, name, Administration Console value, and attributes as the specified XML object.
XML.createElement()	Creates a new XML element with the name specified in the `name` parameter.
XML.createTextNode()	Creates a new XML text node with the specified text.
XML.getBytesLoaded()	Returns the number of bytes loaded (streamed) for the specified XML document.
XML.getBytesTotal()	Returns the size of the XML document, in bytes.
XML.getNamespaceForPrefix()	Returns the namespace URI that is associated with the specified prefix for the node.
XML.getPrefixForNamespace()	Returns the prefix that is associated with the specified namespace URI for the node.
XML.hasChildNodes()	Returns `true` if the specified XML object has child nodes; otherwise, `false`.
XML.insertBefore()	Inserts a new child node into the XML object's child list, before the specified node.
XML.load()	Loads an XML document from a File object or from a URL over HTTP, and replaces the contents of the specified XML object with the XML data.
XML.parseXML()	Parses the XML text specified in the `source` parameter and populates the specified XML object with the resulting XML tree.
XML.removeNode()	Removes the specified XML object from its parent and deletes all descendants of the node.
XML.send()	Encodes the specified XML object into an XML document and sends it to the specified URL by using the POST method in a browser.
XML.sendAndLoad()	Encodes the specified XML object into an XML document, sends it to the specified URL by using the HTTP POST method, downloads the server's response, and loads it into the specified object.
XML.toString()	Evaluates the specified XML object, constructs a textual representation of the XML structure, including the node, children, and attributes, and returns the result as a string.

Event handler summary

Event handler	Description
XML.onData()	Invoked when XML text has been completely downloaded from the server or when an error occurs in downloading XML text from a server.
XML.onHTTPStatus()	Invoked when Flash Media Interactive Server receives an HTTP status code from the server.
XML.onLoad()	Invoked when an XML document is received from the server.

XML constructor

```
new XML([source])
```

Creates a new XML object. You must use the constructor to create an XML object before you call any of the XML class methods.

Note: *Use the* `createElement()` *and* `createTextNode()` *methods to add elements and text nodes to an XML document tree.*

Availability

Flash Media Server 2

Parameters

`source` A string; the XML text to parse to create the new XML object.

Returns

A reference to an XML object.

Example

The following example creates a new, empty XML object:

```
var my_xml = new XML();
```

The following example creates an XML object by parsing the XML text specified in the `source` parameter and populates the newly created XML object with the resulting XML document tree:

```
var other_xml = new XML("<state name=\"California\"><city>San Francisco</city></state>");
```

See also

XML.createElement(), XML.createTextNode()

XML.addRequestHeader()

```
my_xml.addRequestHeader(headerName, headerValue)
my_xml.addRequestHeader([headerName_1, headerValue_1 ... headerName_n, headerValue_n])
```

Adds or changes HTTP request headers (such as `Content-Type` or `SOAPAction`) that are sent with `POST` actions. In the first usage, you pass two strings, `headerName` and `headerValue`, to the method. In the second usage, you pass an array of strings, alternating header names and header values.

If multiple calls are made to set the same header name, each successive value replaces the value set in the previous call.

You cannot add or change the following standard HTTP headers by using this method: Accept-Ranges, Age, Allow, Allowed, Connection, Content-Length, Content-Location, Content-Range, ETag, Host, Last-Modified, Locations, Max-Forwards, Proxy-Authenticate, Proxy-Authorization, Public, Range, Retry-After, Server, TE, Trailer, Transfer-Encoding, Upgrade, URI, Vary, Via, Warning, and WWW-Authenticate.

Note: *A call to* `XML.addRequestHeader()` *that sets a value for the Content-Type header overrides any value set in the* `XML.contentType` *property.*

Availability

Flash Media Server 2

Parameters

`headerName` A string representing an HTTP request header name.

`headerValue` A string representing the value associated with `headerName`.

Example

The following example adds a custom HTTP header named `SOAPAction` with a value of `Foo` to an XML object named `my_xml`:

```
my_xml.addRequestHeader("SOAPAction", "'Foo'");
```

The following example creates an array named `headers` that contains two alternating HTTP headers and their associated values. The array is passed as a parameter to the `addRequestHeader()` method.

```
var headers = new Array("Content-Type", "text/plain", "X-ClientAppVersion", "2.0");
my_xml.addRequestHeader(headers);
```

XML.appendChild()

```
my_xml.appendChild(childNode)
```

Appends the specified node to the XML object's child list. This method operates directly on the node referenced by the `childNode` parameter; it does not append a copy of the node. If the node to be appended already exists in another tree structure, appending the node to the new location removes it from its current location. If the `childNode` parameter refers to a node that already exists in another XML tree structure, the appended child node is placed in the new tree structure after it is removed from its existing parent node.

Availability

Flash Media Server 2

Parameters

`childNode` An XMLNode object that represents the node to be moved from its current location to the child list of the `my_xml` object.

Returns

A boolean value; `true` if successful; otherwise, `false`.

Example

The following example performs these actions:

1 Creates two empty XML documents, `doc1` and `doc2`.

2 Creates a new node, by using the `createElement()` method, and appends it, by using the `appendChild()` method, to the XML document named `doc1`.

3 Shows how to move a node by using the `appendChild()` method, by moving the root node from `doc1` to `doc2`.

4 Clones the root node from `doc2` and appends it to `doc1`.

5 Creates a new node and appends it to the root node of the XML document `doc1`.

```
var doc1 = new XML();
var doc2 = new XML();

// Create a root node and add it to doc1.
var rootnode = doc1.createElement("root");
doc1.appendChild(rootnode);
trace ("doc1: " + doc1); // output: doc1: <root />
trace ("doc2: " + doc2); // output: doc2:

// Move the root node to doc2.
doc2.appendChild(rootnode);
trace ("doc1: " + doc1); // output: doc1:
trace ("doc2: " + doc2); // output: doc2: <root />

// Clone the root node and append it to doc1.
var clone = doc2.firstChild.cloneNode(true);
doc1.appendChild(clone);
trace ("doc1: " + doc1); // output: doc1: <root />
trace ("doc2: " + doc2); // output: doc2: <root />

// Create a new node to append to root node (named clone) of doc1.
var newNode = doc1.createElement("newbie");
clone.appendChild(newNode);
trace ("doc1: " + doc1); // output: doc1: <root><newbie /></root>
```

XML.attributes

`my_xml.attributes`

An object that contains all the attributes of the specified XML object. Associative arrays use keys as indexes, not ordinal integer indexes that are used by regular arrays. In the `XML.attributes` associative array, the key index is a string representing the name of the attribute. The value associated with that key index is the string value associated with that attribute. For example, if you have an attribute named `color`, you would retrieve that attribute's value by using the color as the key index, as shown in the following code:

```
var myColor = doc.firstChild.attributes.color
```

Availability

Flash Media Server 2

Example

The following example shows the XML attribute names:

```
// Create a tag called 'mytag' with
// an attribute called 'name' with value 'Val'.
var doc = new XML("<mytag name=\"Val\"> item </mytag>");

// Assign the value of the 'name' attribute to variable y.
var y = doc.firstChild.attributes.name;
trace (y);// output: Val

// Create a new attribute named 'order' with value 'first'.
doc.firstChild.attributes.order = "first";
```

```
// Assign the value of the 'order' attribute to variable z.
var z = doc.firstChild.attributes.order
trace(z);// output: first
```

XML.childNodes

```
my_xml.childNodes
```

Read-only; an array of the specified XML object's children. Each element in the array is a reference to an XML object that represents a child node. This read-only property cannot be used to manipulate child nodes. Use the `XML.appendChild()`, `XML.insertBefore()`, and `XML.removeNode()` methods to manipulate child nodes.

This property is undefined for text nodes (`nodeType == 3`).

Availability

Flash Media Server 2

Example

The following example shows how to use the `XML.childNodes` property to return an array of child nodes:

```
// Create a new XML document.
var doc = new XML();

// Create a root node.
var rootNode = doc.createElement("rootNode");

// Create three child nodes.
var oldest = doc.createElement("oldest");
var middle = doc.createElement("middle");
var youngest = doc.createElement("youngest");

// Add the rootNode as the root of the XML document tree.
doc.appendChild(rootNode);

// Add each of the child nodes as children of rootNode.
rootNode.appendChild(oldest);
rootNode.appendChild(middle);
rootNode.appendChild(youngest);

// Create an array and use rootNode to populate it.
var firstArray:Array = doc.childNodes;
trace (firstArray);
// Output: <rootNode><oldest /><middle /><youngest /></rootNode>

// Create another array and use the child nodes to populate it.
var secondArray = rootNode.childNodes;
trace(secondArray);
// Output: <oldest />,<middle />,<youngest />
```

See also

`XML.nodeType`

XML.cloneNode()

```
my_xml.cloneNode(deep)
```

Constructs and returns a new XMLNode object of the same type, name, Administration Console value, and attributes as the specified XML object. If `deep` is set to `true`, all child nodes are recursively cloned, resulting in an exact copy of the original object's document tree.

The clone of the node that is returned is no longer associated with the tree of the cloned item. Consequently, `nextSibling`, `parentNode`, and `previousSibling` have a value of `null`. If the `deep` parameter is set to `false`, or if `my_xml` has no child nodes, `firstChild` and `lastChild` are also `null`.

Availability

Flash Media Server 2

Parameters

`deep` A boolean value; if set to `true`, the children of the specified XML object will be recursively cloned; otherwise, `false`.

Returns

An XMLNode object.

Example

The following example shows how to use the `XML.cloneNode()` method to create a copy of a node:

```
// Create a new XML document.
var doc = new XML();

// Create a root node.
var rootNode = doc.createElement("rootNode");

// Create three child nodes.
var oldest = doc.createElement("oldest");
var middle = doc.createElement("middle");
var youngest = doc.createElement("youngest");

// Add the rootNode as the root of the XML document tree.
doc.appendChild(rootNode);

// Add each of the child nodes as children of rootNode.
rootNode.appendChild(oldest);
rootNode.appendChild(middle);
rootNode.appendChild(youngest);

// Create a copy of the middle node by using cloneNode().
var middle2 = middle.cloneNode(false);

// Insert the clone node into rootNode between
// the middle and youngest nodes.
rootNode.insertBefore(middle2, youngest);
trace(rootNode);
// Output (with line breaks added):
// <rootNode>
//<oldest />
//<middle />
//<middle />
//<youngest />
// </rootNode>

// Create a copy of rootNode by using cloneNode() to demonstrate a deep copy.
var rootClone = rootNode.cloneNode(true);
```

```
// Insert the clone, which contains all child nodes, to rootNode.
rootNode.appendChild(rootClone);
trace(rootNode);
// Output (with line breaks added):
// <rootNode>
// <oldest/>
// <middle/>
// <middle/>
// <youngest/>
// <rootNode>
//<oldest/>
//<middle/>
//<middle/>
//<youngest/>
// </rootNode>
// </rootNode>
```

XML.contentType

```
my_xml.contentType
```

The MIME content type that is sent to the server when you call the `XML.send()` or `XML.sendAndLoad()` method. The default is *application/x-www-form-urlencoded*, which is the standard MIME content type used for most HTML forms.

Availability

Flash Media Server 2

Example

The following example creates a new XML document and checks its default content type:

```
// Create a new XML document.
var doc = new XML();

// Trace the default content type.
trace(doc.contentType);

// output: application/x-www-form-urlencoded
```

XML.createElement()

```
my_xml.createElement(name)
```

Creates a new XML element with the name specified in the `name` parameter. The new element initially has no parent, children, or siblings. The method returns a reference to the newly created XML object that represents the element. This method and the `XML.createTextNode()` method are the constructor methods for creating nodes for an XML object.

Availability

Flash Media Server 2

Parameters

`name` A string indicating the tag name of the XML element being created.

Returns

An XML node; an XML element.

Example

The following example creates three XMLNode objects by using the `createElement()` method:

```
// Create an XML document.
var doc = new XML();

// Create three XML nodes by using createElement().
var element1 = doc.createElement("element1");
var element2 = doc.createElement("element2");
var element3 = doc.createElement("element3");

// Place the new nodes into the XML tree.
doc.appendChild(element1);
element1.appendChild(element2);
element1.appendChild(element3);

trace(doc);
// Output: <element1><element2 /><element3 /></element1>
```

See also

`XML.createTextNode()`

XML.createTextNode()

```
my_xml.createTextNode(text)
```

Creates a new XML text node with the specified text. The new node initially has no parent, and text nodes cannot have children or siblings. This method returns a reference to the XML object that represents the new text node. This method and the `XML.createElement()` method are the constructor methods for creating nodes for an XML object.

Availability

Flash Media Server 2

Parameters

`text` A string; the text used to create the new text node.

Returns

An XML node.

Example

The following example creates two XML text nodes by using the `createTextNode()` method and places them into existing XML nodes:

```
// Create an XML document.
var doc = new XML();

// Create three XML nodes by using createElement().
var element1 = doc.createElement("element1");
var element2 = doc.createElement("element2");
var element3 = doc.createElement("element3");

// Place the new nodes into the XML tree.
doc.appendChild(element1);
element1.appendChild(element2);
element1.appendChild(element3);

// Create two XML text nodes by using createTextNode().
```

```
var textNode1 = doc.createTextNode("textNode1");
var textNode2 = doc.createTextNode("textNode2");

// Place the new nodes into the XML tree.
element2.appendChild(textNode1);
element3.appendChild(textNode2);

trace(doc);
// Output (with line breaks added between tags):
// <element1>
//<element2>textNode1</element2>
//<element3>textNode2</element3>
// </element1>
```

See also

XML.createElement()

XML.docTypeDecl

my_xml.docTypeDecl

Specifies information about the XML document's DOCTYPE declaration. After the XML text has been parsed into an XML object, the XML.docTypeDecl property of the XML object is set to the text of the XML document's DOCTYPE declaration (for example, <!DOCTYPE greeting SYSTEM "hello.dtd">). This property is set by using a string representation of the DOCTYPE declaration, not an XMLNode object.

The ActionScript XML parser is not a validating parser. The DOCTYPE declaration is read by the parser and stored in the XML.docTypeDecl property, but no DTD validation is performed.

If no DOCTYPE declaration occurs during a parse operation, the XML.docTypeDecl property is set to undefined. The XML.toString() method outputs the contents of XML.docTypeDecl immediately after the XML declaration stored in XML.xmlDecl and before any other text in the XML object. If XML.docTypeDecl is undefined, there is no DOCTYPE declaration.

Availability

Flash Media Server 2

Example

The following example uses the XML.docTypeDecl property to set the DOCTYPE declaration for an XML object:

```
my_xml.docTypeDecl = "<!DOCTYPE greeting SYSTEM \"hello.dtd\">";
```

XML.firstChild

my_xml.firstChild

Read-only; evaluates the specified XML object and references the first child in the parent node's child list. If the node does not have children, this property is null. If the node is a text node, this property is null. You cannot use this property to manipulate child nodes; use the appendChild(), insertBefore(), and removeNode() methods instead.

Availability

Flash Media Server 2

Example

The following example shows how to use XML.firstChild to loop through a node's child nodes:

```
// Create a new XML document.
var doc = new XML();

// Create a root node.
var rootNode = doc.createElement("rootNode");

// Create three child nodes.
var oldest = doc.createElement("oldest");
var middle = doc.createElement("middle");
var youngest = doc.createElement("youngest");

// Add the rootNode as the root of the XML document tree.
doc.appendChild(rootNode);

// Add each of the child nodes as children of rootNode.
rootNode.appendChild(oldest);
rootNode.appendChild(middle);
rootNode.appendChild(youngest);

// Use firstChild to iterate through the child nodes of rootNode.
for (var aNode = rootNode.firstChild; aNode != null; aNode = aNode.nextSibling) {
    trace(aNode);
}

// Output:
// <oldest />
// <middle />
// <youngest />
```

XML.getBytesLoaded()

`my_xml.getBytesLoaded()`

Returns the number of bytes loaded (streamed) for the XML document. You can compare the value of `getBytesLoaded()` with the value of `getBytesTotal()` to determine what percentage of an XML document has loaded.

Availability

Flash Media Server 2

Returns

A number.

See also

`XML.getBytesTotal()`

XML.getBytesTotal()

`my_xml.getBytesTotal()`

Returns the size of the XML document, in bytes.

Availability

Flash Media Server 2

Returns

A number.

See also

XML.getBytesTotal()

XML.getNamespaceForPrefix()

my_xml.getNamespaceForPrefix(prefix)

Returns the namespace URI that is associated with the specified prefix for the node. To determine the URI, getPrefixForNamespace() searches up the XML hierarchy from the node, as necessary, and returns the namespace URI of the first xmlns declaration for the given prefix.

If no namespace is defined for the specified prefix, the method returns null.

If you specify an empty string ("") as the prefix and a default namespace is defined for the node (as in xmlns="http://www.example.com/"), the method returns that default namespace URI.

Availability

Flash Media Server 2

Parameters

prefix A string; the prefix for which the method returns the associated namespace.

Returns

A string.

Example

The following example creates a very simple XML object and outputs the result of a call to getNamespaceForPrefix():

```
function createXML() {
    var str = "<Outer xmlns:exu=\"http://www.example.com/util\">" + "<exu:Child id='1' />"
+ "<exu:Child id='2' />" + "<exu:Child id='3' />" + "</Outer>";
    return new XML(str).firstChild;
}

var xml = createXML();
trace(xml.getNamespaceForPrefix("exu")); // output: http://www.example.com/util
trace(xml.getNamespaceForPrefix("")); // output: null
```

See also

XML.getPrefixForNamespace()

XML.getPrefixForNamespace()

my_xml.getPrefixForNamespace(nsURI)

Returns the prefix that is associated with the specified namespace URI for the node. To determine the prefix, getPrefixForNamespace() searches up the XML hierarchy from the node, as necessary, and returns the prefix of the first xmlns declaration with a namespace URI that matches nsURI.

If there is no xmlns assignment for the given URI, the method returns null. If there is an xmlns assignment for the given URI but no prefix is associated with the assignment, the method returns an empty string ("").

Availability

Flash Media Server 2

Parameters

`nsURI` A string; the namespace URI for which the method returns the associated prefix.

Returns

A string.

Example

The following example creates a very simple XML object and outputs the result of a call to the `getPrefixForNamespace()` method. The Outer XML node, which is represented by the `xmlDoc` variable, defines a namespace URI and assigns it to the `exu` prefix. Calling the `getPrefixForNamespace()` method with the defined namespace URI ("http://www.example.com/util") returns the prefix `exu`, but calling this method with an undefined URI ("http://www.example.com/other") returns `null`. The first `exu:Child` node, which is represented by the `child1` variable, also defines a namespace URI ("http://www.example.com/child"), but does not assign it to a prefix. Calling this method on the defined, but unassigned, namespace URI returns an empty string.

```
function createXML() {
    var str = "<Outer xmlns:exu=\"http://www.example.com/util\">"
+ "<exu:Child id='1' xmlns=\"http://www.example.com/child\"/>"
+ "<exu:Child id='2' />"
+ "<exu:Child id='3' />"
+ "</Outer>";
    return new XML(str).firstChild;
}

var xmlDoc = createXML();
trace(xmlDoc.getPrefixForNamespace("http://www.example.com/util")); // output: exu
trace(xmlDoc.getPrefixForNamespace("http://www.example.com/other")); // output: null

var child1 = xmlDoc.firstChild;
trace(child1.getPrefixForNamespace("http://www.example.com/child")); // output: [empty
string]
trace(child1.getPrefixForNamespace("http://www.example.com/other")); // output: null
```

See also

`XML.getNamespaceForPrefix()`

XML.hasChildNodes()

`my_xml.hasChildNodes()`

Returns `true` if the specified XML object has child nodes; otherwise, `false`.

Availability

Flash Media Server 2

Returns

A boolean value.

Example

The following example creates a new XML packet. If the root node has child nodes, the code loops over each child node to display the name and value of the node.

```
var my_xml = new
XML("<login><username>hank</username><password>rudolph</password></login>");
if (my_xml.firstChild.hasChildNodes()) {
    // Use firstChild to iterate through the child nodes of rootNode.
```

```
    for (var aNode = my_xml.firstChild.firstChild; aNode != null; aNode=aNode.nextSibling) {
        if (aNode.nodeType == 1) {
            trace(aNode.nodeName+":\t"+aNode.firstChild.nodeValue);
        }
    }
}
```

The following output appears:

```
username:hank
password:rudolph
```

XML.ignoreWhite

```
my_xml.ignoreWhite
XML.prototype.ignoreWhite
```

When set to `true`, discards, during the parsing process, text nodes that contain only white space. The default setting is `false`. Text nodes with leading or trailing white spaces are unaffected.

Usage 1: You can set the `ignoreWhite` property for individual XML objects, as shown in the following code:

```
my_xml.ignoreWhite = true;
```

Usage 2: You can set the default `ignoreWhite` property for XML objects, as shown in the following code:

```
XML.prototype.ignoreWhite = true;
```

Availability

Flash Media Server 2

Example

The following example loads an XML file with a text node that contains only white space; the `foyer` tag contains 14 space characters. To run this example, create a text file named flooring.xml and copy the following tags into it:

```
<house>
 <kitchen> ceramic tile </kitchen>
 <bathroom> linoleum </bathroom>
 <foyer></foyer>
</house>
```

The following is the server-side code:

```
// Create a new XML object.
var flooring = new XML();

// Set the ignoreWhite property to true (the default value is false).
flooring.ignoreWhite = true;

// After loading is complete, trace the XML object.
flooring.onLoad = function(success) {
    trace(flooring);
}

// Load the XML into the flooring object.
flooring.load("flooring.xml");

/* output (line breaks added for clarity):
<house>
<kitchen>ceramic tile</kitchen>
<bathroom>linoleum</bathroom>
</foyer>
```

```
</house>
*/
```

If you change the setting of `flooring.ignoreWhite` to `false`, or simply remove that line of code entirely, the 14 space characters in the `foyer` tag are preserved:

```
...
// Set the ignoreWhite property to false (the default value).
flooring.ignoreWhite = false;
...
/* output (line breaks added for clarity):
<house>
    <kitchen> ceramic tile </kitchen>
    <bathroom>linoleum</bathroom>
    <foyer></foyer>
</house>
*/
```

XML.insertBefore()

```
my_xml.insertBefore(childNode, beforeNode)
```

Inserts a new child node into the XML object's child list, before the `beforeNode` node. If `beforeNode` is not a child of `my_xml`, the insertion fails.

Availability

Flash Media Server 2

Parameters

`childNode` The XMLNode object to be inserted.

`beforeNode` The XMLNode object before the insertion point for the `childNode` node.

Returns

A boolean value; `true` if successful; otherwise, `false`.

Example

The following example is an excerpt from the `XML.cloneNode()` example:

```
// Create a copy of the middle node by using cloneNode().
var middle2 = middle.cloneNode(false);

// Insert the clone node into rootNode
// between the middle and youngest nodes.
rootNode.insertBefore(middle2, youngest);
```

XML.lastChild

```
my_xml.lastChild
```

Read-only; an XMLNode value that references the last child in the node's child list. If the node does not have children, the `XML.lastChild` property is `null`. You cannot use this property to manipulate child nodes; use the `appendChild()`, `insertBefore()`, and `removeNode()` methods instead.

Availability

Flash Media Server 2

Example

The following example uses the `XML.lastChild` property to iterate through the child nodes of an XMLNode object, starting with the last item in the node's child list and ending with the first child of the node's child list:

```
// Create a new XML document.
var doc = new XML();

// Create a root node.
var rootNode = doc.createElement("rootNode");

// Create three child nodes.
var oldest = doc.createElement("oldest");
var middle = doc.createElement("middle");
var youngest = doc.createElement("youngest");

// Add the rootNode as the root of the XML document tree.
doc.appendChild(rootNode);

// Add each of the child nodes as children of rootNode.
rootNode.appendChild(oldest);
rootNode.appendChild(middle);
rootNode.appendChild(youngest);

// Use lastChild to iterate through the child nodes of rootNode.
for (var aNode = rootNode.lastChild; aNode != null; aNode = aNode.previousSibling) {
    trace(aNode);
}

/*
output:
<youngest />
<middle />
<oldest />
*/
```

The following example creates a new XML packet and uses the `XML.lastChild` property to iterate through the child nodes of the root node:

```
// Create a new XML document.
var doc = new XML("<rootNode><oldest /><middle /><youngest /></rootNode>");

var rootNode = doc.firstChild;

// Use lastChild to iterate through the child nodes of rootNode.
for (var aNode = rootNode.lastChild; aNode != null; aNode=aNode.previousSibling) {
    trace(aNode);
}
/*
output:
<youngest />
<middle />
<oldest />
*/
```

XML.load()

```
my_xml.load(url)
```

Loads an XML document from a File object or from a URL over HTTP, and replaces the contents of the specified XML object with the XML data. The load process is asynchronous; it does not finish immediately after the `load()` method is executed.

When the `load()` method is executed, the XML object property `loaded` is set to `false`. When the XML data finishes downloading, the `loaded` property is set to `true` and the `onLoad()` event handler is invoked. The XML data is not parsed until it is completely downloaded. If the XML object previously contained any XML trees, they are discarded.

You can define a custom function that is executed when the `onLoad()` event handler of the XML object is invoked.

Availability

Flash Media Server 2

Parameters

`url` A File object or a URL where the XML document to be loaded is located. If the SWF file that issues this call is running in a web browser, `url` must be in the same domain as the SWF file. You cannot use a file path for this parameter.

Returns

A boolean value; `true` if successful; otherwise, `false`.

Example

The following simple example uses the `XML.load()` method:

```
// Create a new XML object.
var flooring = new XML();

// Set the ignoreWhite property to true (the default value is false).
flooring.ignoreWhite = true;

// After loading is complete, trace the XML object.
flooring.onLoad = function(success) {
    trace(flooring);
};

// Load the XML into the flooring object.
flooring.load("http://somehttpserver/flooring.xml");
```

For the contents of the flooring.xml file, and the output that this example produces, see the example for `XML.ignoreWhite`.

XML.loaded

`my_xml.loaded`

A boolean value; `true` if the document-loading process initiated by the `XML.load()` call completed successfully; otherwise, `false`.

Availability

Flash Media Server 2

Example

The following example uses the `XML.loaded` property in a simple script:

```
var my_xml = new XML();
my_xml.ignoreWhite = true;
```

```
my_xml.onLoad = function(success) {
    trace("success: "+success);
    trace("loaded:"+my_xml.loaded);
    trace("status:"+my_xml.status);
};
my_xml.load("http://www.flash-mx.com/mm/problems/products.xml");
```

Information is written to the log file when the `onLoad()` handler is invoked. If the call completes successfully, the `loaded` status `true` is written to the log file, as shown in the following example:

```
success: true
loaded:true
status:0
```

XML.localName

`my_xml.localName`

Read-only; the local name portion of the XML node's name. This is the element name without the namespace prefix. For example, the node `<contact:mailbox/>bob@example.com</contact:mailbox>` has the local name `mailbox` and the prefix `contact`, which comprise the full element name `contact.mailbox`.

You can access the namespace prefix by using the `XML.prefix` property of the XML node object. The `XML.nodeName` property returns the full name, including the prefix and the local name.

Availability

Flash Media Server 2

Example

This example uses a SWF file and an XML file located in the same directory. The XML file, named SoapSample.xml, contains the following code:

```
<?xml version="1.0"?>
    <soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope">
        <soap:Body xmlns:w="http://www.example.com/weather">
            <w:GetTemperature>
                <w:City>San Francisco</w:City>
            </w:GetTemperature>
        </soap:Body>
    </soap:Envelope>
```

The source for the SWF file contains the following script (note the comments for the Output strings):

```
var xmlDoc = new XML()
xmlDoc.ignoreWhite = true;
xmlDoc.load("http://www.example.com/SoapSample.xml")
xmlDoc.onLoad = function(success) {
    var tempNode = xmlDoc.childNodes[0].childNodes[0].childNodes[0];
    trace("w:GetTemperature localname: " + tempNode.localName);
    // Output: ... GetTemperature
    var soapEnvNode = xmlDoc.childNodes[0];
    trace("soap:Envelope localname: " + soapEnvNode.localName);
    // Output: ... Envelope
};
```

See also

`XML.nodeName`, `XML.prefix`

XML.namespaceURI

my_xml.namespaceURI

Read-only; if the XML node has a prefix, the value of the xmlns declaration for that prefix (the URI), which is typically called the namespace URI. The xmlns declaration is in the current node or in a node higher in the XML hierarchy.

If the XML node does not have a prefix, the value of the namespaceURI property depends on whether a default namespace is defined (as in xmlns="http://www.example.com/"). If there is a default namespace, the value of the namespaceURI property is the value of the default namespace. If there is no default namespace, the namespaceURI property for that node is an empty string ("").

You can use the XML.getNamespaceForPrefix() method to identify the namespace associated with a specific prefix. The namespaceURI property returns the prefix associated with the node name.

Availability

Flash Media Server 2

Example

The following example shows how the namespaceURI property is affected by the use of prefixes. The XML file used in the example is named SoapSample.xml and contains the following tags:

```
<?xml version="1.0"?>
    <soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope">
        <soap:Body xmlns:w="http://www.example.com/weather">
            <w:GetTemperature>
                <w:City>San Francisco</w:City>
            </w:GetTemperature>
        </soap:Body>
    </soap:Envelope>
```

The source for the Server-Side ActionScript Communication File (ASC file) contains the following script (note the comments for the Output strings). For tempNode, which represents the w:GetTemperature node, the value of namespaceURI is defined in the soap:Body tag. For soapBodyNode, which represents the soap:Body node, the value of namespaceURI is determined by the definition of the soap prefix in the node above it, rather than the definition of the w prefix that the soap:Body node contains.

```
var xmlDoc = new XML();
xmlDoc.load("http://www.example.com/SoapSample.xml");
xmlDoc.ignoreWhite = true;
xmlDoc.onLoad = function(success:Boolean) {
    var tempNode:XMLNode = xmlDoc.childNodes[0].childNodes[0].childNodes[0];
    trace("w:GetTemperature namespaceURI: " + tempNode.namespaceURI);
    // Output: ... http://www.example.com/weather

    trace("w:GetTemperature soap namespace: " + tempNode.getNamespaceForPrefix("soap"));
    // Output: ... http://www.w3.org/2001/12/soap-envelope

    var soapBodyNode = xmlDoc.childNodes[0].childNodes[0];
    trace("soap:Envelope namespaceURI: " + soapBodyNode.namespaceURI);
    // Output: ... http://www.w3.org/2001/12/soap-envelope
};
```

The following example uses XML tags without prefixes. It uses a SWF file and an XML file located in the same directory. The XML file, named NoPrefix.xml, contains the following tags:

```
<?xml version="1.0"?>
<rootnode>
```

```
    <simplenode xmlns="http://www.w3.org/2001/12/soap-envelope">
        <innernode/>
    </simplenode>
</rootnode>
```

The source for the server-side script file contains the following code (note the comments for the Output strings). The `rootNode` node does not have a default namespace, so its `namespaceURI` value is an empty string. The `simpleNode` node defines a default namespace, so its `namespaceURI` value is the default namespace. The `innerNode` node does not define a default namespace, but uses the default namespace defined by `simpleNode`, so its `namespaceURI` value is the same as that of `simpleNode`.

```
var xmlDoc = new XML()
xmlDoc.load("http://www.example.com/NoPrefix.xml");
xmlDoc.ignoreWhite = true;
xmlDoc.onLoad = function(success) {
    var rootNode = xmlDoc.childNodes[0];
    trace("rootNode Node namespaceURI: " + rootNode.namespaceURI);
    // Output: [empty string]

    var simpleNode = xmlDoc.childNodes[0].childNodes[0];
    trace("simpleNode Node namespaceURI: " + simpleNode.namespaceURI);
    // Output: ... http://www.w3.org/2001/12/soap-envelope

    var innerNode = xmlDoc.childNodes[0].childNodes[0].childNodes[0];
    trace("innerNode Node namespaceURI: " + innerNode.namespaceURI);
    // Output: ... http://www.w3.org/2001/12/soap-envelope
};
```

XML.nextSibling

`my_xml.nextSibling`

Read-only; an XMLNode value that references the next sibling in the parent node's child list. If the node does not have a next sibling node, this property is `null`. This property cannot be used to manipulate child nodes; use the `appendChild()`, `insertBefore()`, and `removeNode()` methods to manipulate child nodes.

Availability

Flash Media Server 2

Example

The following example is an excerpt from the example for the `XML.firstChild` property. It shows how you can use the `XML.nextSibling` property to loop through an XMLNode object's child nodes.

```
for (var aNode = rootNode.firstChild; aNode != null; aNode = aNode.nextSibling) {
    trace(aNode);
}
```

XML.nodeName

`my_xml.nodeName`

A string representing the node name of the XML object. If the XML object is an XML element (`nodeType==1`), `nodeName` is the name of the tag that represents the node in the XML file. For example, `TITLE` is the node name of an HTML `TITLE` tag. If the XML object is a text node (`nodeType==3`), `nodeName` is `null`.

Availability

Flash Media Server 2

Example

The following example creates an element node and a text node, and checks the node name of each:

```
// Create an XML document.
var doc = new XML();

// Create an XML node by using createElement().
var myNode = doc.createElement("rootNode");

// Place the new node into the XML tree.
doc.appendChild(myNode);

// Create an XML text node by using createTextNode().
var myTextNode = doc.createTextNode("textNode");

// Place the new node into the XML tree.
myNode.appendChild(myTextNode);

trace(myNode.nodeName);
trace(myTextNode.nodeName);

/*
output:
rootNode
null
*/
```

The following example creates a new XML packet. If the root node has child nodes, the code loops over each child node to display the name and value of the node. Add the following ActionScript to your ASC file:

```
var my_xml = new
XML("<login><username>hank</username><password>rudolph</password></login>");
if (my_xml.firstChild.hasChildNodes()) {
    // Use firstChild to iterate through the child nodes of rootNode.
    for (var aNode = my_xml.firstChild.firstChild; aNode != null; aNode=aNode.nextSibling) {
        if (aNode.nodeType == 1) {
            trace(aNode.nodeName+":\t"+aNode.firstChild.nodeValue);
        }
    }
}
```

The following node names appear:

```
username:hank
password:rudolph
```

XML.nodeType

`my_xml.nodeType`

Read-only; a `nodeType` value, either 1 for an XML element or 3 for a text node.

The `nodeType` property is a numeric value from the NodeType enumeration in the W3C DOM Level 1 Recommendation. The following table lists the values:

Integer value	Defined constant
1	ELEMENT_NODE
2	ATTRIBUTE_NODE
3	TEXT_NODE
4	CDATA_SECTION_NODE
5	ENTITY_REFERENCE_NODE
6	ENTITY_NODE
7	PROCESSING_INSTRUCTION_NODE
8	COMMENT_NODE
9	DOCUMENT_NODE
10	DOCUMENT_TYPE_NODE
11	DOCUMENT_FRAGMENT_NODE
12	NOTATION_NODE

In Flash Player, the built-in XML class supports only 1 (ELEMENT_NODE) and 3 (TEXT_NODE).

Availability

Flash Media Server 2

Example

The following example creates an element node and a text node and checks the node type of each:

```
// Create an XML document.
var doc = new XML();

// Create an XML node by using createElement().
var myNode = doc.createElement("rootNode");

// Place the new node into the XML tree.
doc.appendChild(myNode);

// Create an XML text node by using createTextNode().
var myTextNode = doc.createTextNode("textNode");

// Place the new node into the XML tree.
myNode.appendChild(myTextNode);

trace(myNode.nodeType);
trace(myTextNode.nodeType);

/*
output:
1
3
*/
```

XML.nodeValue

```
my_xml.nodeValue
```

The node value of the XML object. If the XML object is a text node, the `nodeType` is 3, and the `nodeValue` is the text of the node. If the XML object is an XML element (`nodeType` is 1), `nodeValue` is `null` and read-only.

Availability

Flash Media Server 2

Example

The following example creates an element node and a text node and checks the node value of each:

```
// Create an XML document.
var doc = new XML();

// Create an XML node by using createElement().
var myNode = doc.createElement("rootNode");

// Place the new node into the XML tree.
doc.appendChild(myNode);

// Create an XML text node by using createTextNode().
var myTextNode = doc.createTextNode("myTextNode");

// Place the new node into the XML tree.
myNode.appendChild(myTextNode);

trace(myNode.nodeValue);
trace(myTextNode.nodeValue);

/*
output:
null
myTextNode
*/
```

The following example creates and parses an XML packet. The code loops through each child node and displays the node value by using the `firstChild` property and `firstChild.nodeValue`.

```
var my_xml = new
XML("<login><username>morton</username><password>good&amp;evil</password></login>");
trace("using firstChild:");
for (var i = 0; i<my_xml.firstChild.childNodes.length; i++) {
    trace("\t"+my_xml.firstChild.childNodes[i].firstChild);
}
trace("");
trace("using firstChild.nodeValue:");
for (var i = 0; i<my_xml.firstChild.childNodes.length; i++) {
    trace("\t"+my_xml.firstChild.childNodes[i].firstChild.nodeValue);
}
```

The following information is written to the log file:

```
using firstChild:
    morton
    good&evil

using firstChild.nodeValue:
    morton
    good&evil
```

XML.onData()

```
my_xml.onData = function(src) {}
```

Invoked when XML text has been completely downloaded from the server or when an error occurs in downloading XML text from a server. This handler is invoked before the XML is parsed, and you can use it to call a custom parsing routine instead of using the Flash XML parser. The `src` parameter is a string that contains XML text downloaded from the server, unless an error occurs during the download. In this situation, the `src` parameter is `undefined`.

By default, the `XML.onData()` event handler invokes `XML.onLoad()`. You can override the `XML.onData()` event handler with custom behavior, but `XML.onLoad()` is not called unless you call it in your `XML.onData()` implementation.

Availability

Flash Media Server 2

Parameters

`src` A string or `undefined`; the raw data, usually in XML format, that is sent by the server.

Example

The following example shows what the `XML.onData()` event handler looks like by default:

```
XML.prototype.onData = function (src) {
    if (src == undefined) {
        this.onLoad(false);
    } else {
        this.parseXML(src);
        this.loaded = true;
        this.onLoad(true);
    }
};
```

You can override the `XML.onData()` event handler to intercept the XML text without parsing it.

XML.onHTTPStatus()

```
myXML.onHTTPStatus(httpStatus){}
```

Invoked when Flash Media Interactive Server receives an HTTP status code from the server. This handler lets you capture and act on HTTP status codes.

The `onHTTPStatus()` handler is invoked before `onData()`, which triggers calls to `onLoad()` with a value of `undefined` if the load fails. After `onHTTPStatus()` is triggered, `onData()` is always triggered, whether or not you override `onHTTPStatus()`. To best use the `onHTTPStatus()` handler, you should write a function to catch the result of the `onHTTPStatus()` call; you can then use the result in your `onData()` and `onLoad()` handlers. If `onHTTPStatus()` is not invoked, this indicates that Flash Media Server did not try to make the URL request.

If Flash Media Interactive Server cannot get a status code from the server, or if it cannot communicate with the server, the default value of 0 is passed to your ActionScript code.

Availability

Flash Media Server 2

Parameters

`httpStatus` A number; the HTTP status code returned by the server. For example, a value of 404 indicates that the server has not found a match for the requested URI. HTTP status codes can be found in sections 10.4 and 10.5 of the HTTP Specification at ftp://ftp.isi.edu/in-notes/rfc2616.txt.

Example

The following example shows how to use `onHTTPStatus()` to help with debugging. The example collects HTTP status codes and assigns their value and type to an instance of the XML object. (This example creates the instance members `this.httpStatus` and `this.httpStatusType` at runtime.) The `onData()` handler uses these instance members to trace information about the HTTP response that can be useful in debugging.

```
var myXml = new XML();

myXml.onHTTPStatus = function(httpStatus) {
    this.httpStatus = httpStatus;
    if(httpStatus < 100) {
        this.httpStatusType = "flashError";
    }
    else if(httpStatus < 200) {
        this.httpStatusType = "informational";
    }
    else if(httpStatus < 300) {
        this.httpStatusType = "successful";
    }
    else if(httpStatus < 400) {
        this.httpStatusType = "redirection";
    }
    else if(httpStatus < 500) {
        this.httpStatusType = "clientError";
    }
    else if(httpStatus < 600) {
        this.httpStatusType = "serverError";
    }
};

myXml.onData = function(src) {
    trace(">> " + this.httpStatusType + ": " + this.httpStatus);
    if(src != undefined) {
        this.parseXML(src);
        this.loaded = true;
        this.onLoad(true);
    } else {
        this.onLoad(false);
    }
};

myXml.onLoad = function(success) {
    // Insert code here.
}

myXml.load("http://weblogs.macromedia.com/mxna/xml/rss.cfm?query=byMostRecent&languages=1"
);
```

See also

`LoadVars.onHTTPStatus()`, `XML.send()`, `XML.sendAndLoad()`

XML.onLoad()

```
my_xml.onLoad = function (success) {}
```

Invoked when an XML document is received from the server. If the XML document is received successfully, the `success` parameter is `true`. If the document was not received, or if an error occurred in receiving the response from the server, the `success` parameter is `false`. The default implementation of this method is not active. To override the default implementation, you must assign a function that contains custom actions.

Availability

Flash Media Server 2

Parameters

`success` A boolean value; `true` if the XML object successfully loads with an `XML.load()` or `XML.sendAndLoad()` operation; otherwise, `false`.

Example

The following example includes ActionScript for a simple e-commerce storefront application. The `sendAndLoad()` method transmits an XML element that contains the user's name and password and uses an `XML.onLoad()` handler to process the reply from the server.

```
var login_str = "<login username=\""+username_txt.text+"\"
password=\""+password_txt.text+"\" />";
var my_xml = new XML(login_str);
var myLoginReply_xml = new XML();
myLoginReply_xml.ignoreWhite = true;
myLoginReply_xml.onLoad = function(success){
    if (success) {
        if ((myLoginReply_xml.firstChild.nodeName == "packet") &&
             (myLoginReply_xml.firstChild.attributes.success == "true")) {
            gotoAndStop("loggedIn");
        } else {
            gotoAndStop("loginFailed");
        }
    } else {
        gotoAndStop("connectionFailed");
    }
};
my_xml.sendAndLoad("http://www.flash-mx.com/mm/login_xml.cfm", myLoginReply_xml);
```

See also

`XML.load()`, `XML.sendAndLoad()`

XML.parentNode

```
my_xml.parentNode
```

Read-only; an XMLNode value that references the parent node of the specified XML object or returns `null` if the node has no parent. This property cannot be used to manipulate child nodes; use the `appendChild()`, `insertBefore()`, and `removeNode()` methods instead.

Availability

Flash Media Server 2

Example

The following example creates an XML packet and writes the parent node of the `username` node to the log file:

```
var my_xml = new
XML("<login><username>morton</username><password>good&amp;evil</password></login>");

// The first child is the <login /> node.
var rootNode = my_xml.firstChild;

// The first child of the root is the <username /> node.
var targetNode = rootNode.firstChild;
trace("the parent node of '"+targetNode.nodeName+"' is: "+targetNode.parentNode.nodeName);
trace("contents of the parent node are:\n"+targetNode.parentNode);

/* output (line breaks added for clarity):

the parent node of 'username' is: login
contents of the parent node are:
<login>
    <username>morton</username>
    <password>good&amp;evil</password>
</login>

*/
```

XML.parseXML()

```
my_xml.parseXML(source)
```

Parses the XML text specified in the `source` parameter and populates the specified XML object with the resulting XML tree. Any existing trees in the XML object are discarded.

Availability

Flash Media Server 2

Parameters

`source` A string; the XML text to be parsed and passed to the specified XML object.

Returns

A boolean value; `true` if successful, otherwise, `false`.

Example

The following example creates and parses an XML packet:

```
var xml_str = "<state name=\"California\"><city>San Francisco</city></state>"

// Defining the XML source within the XML constructor:
var my1_xml = new XML(xml_str);
trace(my1_xml.firstChild.attributes.name); // output: California

// Defining the XML source by using the XML.parseXML method:
var my2_xml = new XML();
my2_xml.parseXML(xml_str);
trace(my2_xml.firstChild.attributes.name);
// output: California
```

XML.prefix

```
my_xml.prefix
```

Read-only; the prefix portion of the XML node name. For example, in the node `<contact:mailbox/>bob@example.com</contact:mailbox>`, the prefix `contact` and the local name `mailbox` comprise the full element name `contact.mailbox`.

Availability

Flash Media Server 2

Example

A directory contains a server-side script file and an XML file. The XML file, named SoapSample.xml, contains the following code:

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope">
<soap:Body xmlns:w="http://www.example.com/weather">
<w:GetTemperature>
<w:City>San Francisco</w:City>
</w:GetTemperature>
</soap:Body>
</soap:Envelope>
```

The source for the server-side script file contains the following code (note the comments for the Output strings):

```
var xmlDoc = new XML();
xmlDoc.ignoreWhite = true;
xmlDoc.load("http://www.example.com/SoapSample.xml");
xmlDoc.onLoad = function(success) {
    var tempNode = xmlDoc.childNodes[0].childNodes[0].childNodes[0];
    trace("w:GetTemperature prefix: " + tempNode.prefix); // Output: ... w
    var soapEnvNode = xmlDoc.childNodes[0];
    trace("soap:Envelope prefix: " + soapEnvNode.prefix); // Output: ... soap
};
```

XML.previousSibling

`my_xml.previousSibling`

Read-only; an XMLNode value that references the previous sibling in the parent node's child list. If the node does not have a previous sibling node, the property has a value of `null`. This property cannot be used to manipulate child nodes; use the `XML.appendChild()`, `XML.insertBefore()`, and `XML.removeNode()` methods instead.

Availability

Flash Media Server 2

Example

The following example is an excerpt from the example for the `XML.lastChild` property. It shows how you can use the `XML.previousSibling` property to loop through an XMLNode object's child nodes:

```
for (var aNode = rootNode.lastChild; aNode != null; aNode = aNode.previousSibling) {
    trace(aNode);
}
```

XML.removeNode()

`my_xml.removeNode()`

Removes the specified XML object from its parent and deletes all descendants of the node.

Availability

Flash Media Server 2

Example

The following example creates an XML packet and then deletes the specified XML object and its descendant nodes:

```
var xml_str = "<state name=\"California\"><city>San Francisco</city></state>";

var my_xml = new XML(xml_str);
var cityNode = my_xml.firstChild.firstChild;
trace("before XML.removeNode():\n"+my_xml);
cityNode.removeNode();
trace("after XML.removeNode():\n"+my_xml);

/* output (line breaks added for clarity):
before XML.removeNode():
<state name="California">
<city>San Francisco</city>
</state>

after XML.removeNode():
<state name="California" />
*/
```

XML.send()

```
my_xml.send(url, [fileObj])
```

Encodes the specified XML object into an XML document and sends it to the specified URL by using the `POST` method in a browser. The Flash test environment uses only the `GET` method.

Availability

Flash Media Server 2

Returns

A boolean value; `true` if successful, otherwise, `false`.

Parameters

`url` A string; the destination URL for the specified XML object.

`fileObj` A File object, that is not read-only, to which the response is written. If the File object is not open, Flash Media Interactive Server opens the file, writes to it, and closes it. If the File object is open, Flash Media Interactive Server writes to the file and leaves it open. This parameter is optional.

Example

The following example defines an XML packet and sets the content type for the XML object. The data is then sent to a server and the result is written in a File object.

```
var my_xml = new XML("<highscore><name>Ernie</name><score>13045</score></highscore>");
my_xml.contentType = "text/xml";
my_xml.send("http://www.flash-mx.com/mm/highscore.cfm", myFile);
```

XML.sendAndLoad()

```
my_xml.sendAndLoad(url, targetXMLobject)
```

Encodes the specified XML object into an XML document, sends it to the specified URL using the HTTP `POST` method, downloads the server's response, and loads it into the `targetXMLobject` object. The server response loads the same as the response to the `XML.load()` method.

When `sendAndLoad()` is executed, the `loaded` property is set to `false`. When the XML data finishes loading successfully, and the `onLoad()` event handler is invoked. The XML data is not parsed until it is completely downloaded. If the XML object previously contained any XML trees, they are discarded.

Availability

Flash Media Server 2

Parameters

`url` A string; the destination URL for the specified XML object. If the SWF file issuing this call is running in a web browser, `url` must be in the same domain as the SWF file.

`targetXMLobject` An XML object created with the XML constructor method that will receive the return information from the server.

Returns

A boolean value; `true` if successful, otherwise, `false`.

XML.status

```
my_xml.status
```

A number indicating whether an XML document was successfully parsed into an XML object. The following table contains the numeric status codes and their descriptions:

Status code	Description
`0`	No error; parse was completed successfully.
`-2`	A CDATA section was not properly terminated.
`-3`	The XML declaration was not properly terminated.
`-4`	The `DOCTYPE` declaration was not properly terminated.
`-5`	A comment was not properly terminated.
`-6`	An XML element was malformed.
`-7`	Out of memory.
`-8`	An attribute value was not properly terminated.
`-9`	A start tag was not matched with an end tag.
`-10`	An end tag was encountered without a matching start tag.

Availability

Flash Media Server 2

Example

The following example loads an XML packet into a SWF file. A status message indicates whether the XML is loaded and parsed successfully.

```
var my_xml = new XML();
```

```
my_xml.onLoad = function(success) {
    if (success) {
        if (my_xml.status == 0) {
            trace("XML was loaded and parsed successfully");
        } else {
            trace("XML was loaded successfully, but was unable to be parsed.");
        }
        var errorMessage;
        switch (my_xml.status) {
        case 0 :
            errorMessage = "No error; parse was completed successfully.";
            break;
        case -2 :
            errorMessage = "A CDATA section was not properly terminated.";
            break;
        case -3 :
            errorMessage = "The XML declaration was not properly terminated.";
            break;
        case -4 :
            errorMessage = "The DOCTYPE declaration was not properly terminated.";
            break;
        case -5 :
            errorMessage = "A comment was not properly terminated.";
            break;
        case -6 :
            errorMessage = "An XML element was malformed.";
            break;
        case -7 :
            errorMessage = "Out of memory.";
            break;
        case -8 :
            errorMessage = "An attribute value was not properly terminated.";
            break;
        case -9 :
            errorMessage = "A start tag was not matched with an end tag.";
            break;
        case -10 :
            errorMessage = "An end tag was encountered without a matching
                start tag.";
            break;
        default :
            errorMessage = "An unknown error has occurred.";
            break;
        }
        trace("status: "+my_xml.status+" ("+errorMessage+")");
    } else {
        trace("Unable to load/parse XML. (status: "+my_xml.status+")");
    }
};
my_xml.load("http://www.flash-mx.com/mm/badxml.xml");
```

XML.toString()

```
my_xml.toString()
```

Evaluates the specified XML object, constructs a textual representation of the XML structure, including the node, children, and attributes, and returns the result as a string.

For top-level XML objects (those created with the constructor), the `XML.toString()` method outputs the document's XML declaration (stored in the `XML.xmlDecl` property), followed by the document's `DOCTYPE` declaration (stored in the `XML.docTypeDecl` property), followed by the text representation of all XML nodes in the object. If the `XML.xmlDecl` property is `undefined`, the XML declaration is not output. If the `XML.docTypeDecl` property is `undefined`, the `DOCTYPE` declaration is not output.

Availability

Flash Media Server 2

Returns

A string.

Example

The following example of the `XML.toString()` method sends `<h1>test</h1>` to the log file:

```
var node = new XML("<h1>test</h1>");
trace(node.toString());
```

XML.xmlDecl

`my_xml.xmlDecl`

Specifies information about a document's XML declaration. After the XML document is parsed into an XML object, this property is set to the text of the document's XML declaration. This property is set by using a string representation of the XML declaration, not an XMLNode object. If no XML declaration is encountered during a parse operation, the property is set to `undefined`.`XML`. The `XML.toString()` method outputs the contents of the `XML.xmlDecl` property before any other text in the XML object. If the `XML.xmlDecl` property contains the `undefined` type, no XML declaration is output.

Availability

Flash Media Server 2

Example

The following example loads an XML file and outputs information about the file:

```
var my_xml = new XML();
my_xml.ignoreWhite = true;
my_xml.onLoad = function(success){
    if (success){
        trace("xmlDecl: " + my_xml.xmlDecl);
        trace("contentType: " + my_xml.contentType);
        trace("docTypeDecl: " + my_xml.docTypeDecl);
        trace("packet: " + my_xml.toString());
    }
    else {
        trace("Unable to load remote XML.");
    }
};
my_xml.load("http://foo.com/crossdomain.xml");
```

See also

`XML.docTypeDecl`, `XML.toString()`

XMLSocket class

The XMLSocket class implements client sockets that let Flash Media Interactive Server communicate with a server identified by an IP address or domain name. The XMLSocket class is useful for client-server applications that require low latency, such as real-time chat systems. A traditional HTTP-based chat solution polls the server frequently and downloads new messages by using an HTTP request. In contrast, an XMLSocket chat solution maintains an open connection to the server, which lets the server send incoming messages immediately, without a request from the client.

Note: *You can also use the XMLSocket class to create an XMLStreams object. See XMLSocket constructor and XMLStreams class.*

To use the XMLSocket class, the server computer must run a daemon that understands the protocol used by this class. The protocol has the following characteristics:

- XML messages are sent over a full-duplex TCP/IP stream socket connection.
- Each XML message is a complete XML document, terminated by a zero (0) byte.
- An unlimited number of XML messages can be sent and received over a single XMLSocket connection.

The following restriction applies to how and where an XMLSocket object can connect to the server:

- The `XMLSocket.connect()` method can connect only to TCP port numbers greater than or equal to 1024. One consequence of this restriction is that the server daemons that communicate with the XMLSocket object must also be assigned to port numbers greater than or equal to 1024. Port numbers less than 1024 are often used by system services such as FTP, Telnet, and HTTP, which prohibits XMLSocket objects from these ports for security reasons. The port number restriction limits the possibility that these resources can be inappropriately accessed and abused.

To use the methods of the XMLSocket class, you must first use the constructor, `new XMLSocket()`, to create an XMLSocket object.

Availability

Flash Media Server 2

Property summary

Property	Description
`XMLSocket.maxUnprocessedChars`	The number of characters the connection can receive from the XML server without receiving an end tag or the XMLSocket connection closes.

Method summary

Method	Description
`XMLSocket.close()`	Closes the connection specified by the XMLSocket object.
`XMLSocket.connect()`	Establishes a connection to the specified Internet host by using the specified TCP port (must be 1024 or higher), and returns `true` or `false`, depending on whether a connection is successfully established.
`XMLSocket.send()`	Converts the XML object or data specified in the `object` parameter to a string and transmits it to the server, followed by a zero (0) byte.

Event handler summary

Event handler	Description
`XMLSocket.onClose()`	Invoked when an open connection is closed by the server.
`XMLSocket.onConnect()`	Invoked by Flash Media Interactive Server when a connection request initiated through `XMLSocket.connect()` succeeds or fails.
`XMLSocket.onData()`	Invoked when a message has been downloaded from the server, terminated by a zero (0) byte.
`XMLSocket.onXML()`	Invoked when the specified XML object containing an XML document arrives through an open XMLSocket connection.

XMLSocket constructor

```
new XMLSocket(streamOrFlash)
```

Creates a new XMLSocket object (`"flash"`) or a new XMLStreams object (`"stream"`). The XMLSocket and XMLStreams objects are not initially connected to any server. You must call `XMLSocket.connect()` to connect the object to a server.

For more information about the XMLStreams class, see XMLStreams class.

Availability

Flash Communication Server 1.5

Parameters

`streamOrFlash` A string indicating whether this object is an XMLSocket object or an XMLStreams object. This parameter can have one of the following two values: `"flash"` or `"stream"`.

Returns

A reference to an XMLSocket object or an XMLStreams object.

Example

The following example creates an XMLSocket object:

```
var socket = new XMLSocket("flash");
```

The following example creates an XMLStreams object:

```
var stream = new XMLSocket("stream");
```

XMLSocket.close()

```
myXMLSocket.close()
```

Closes the connection specified by the XMLSocket object.

Availability

Flash Media Server 2

Example

The following simple example creates an XMLSocket object, attempts to connect to the server, and then closes the connection:

```
var socket = new XMLSocket();
socket.connect(null, 2000);
```

```
socket.close();
```

XMLSocket.connect()

```
myXMLSocket.connect(host, port)
```

Establishes a connection to the specified Internet host by using the specified TCP port (must be 1024 or higher), and returns `true` or `false`, depending on whether a connection is successfully established. If you don't know the port number of the Internet host computer, contact your network administrator.

If you specify `null` for the `host` parameter, the local host is contacted.

The Server-Side ActionScript `XMLSocket.connect()` method can connect to computers that are not in the same domain as the SWF file.

If `XMLSocket.connect()` returns a value of `true`, the initial stage of the connection process is successful. Later, the `XMLSocket.onConnect()` handler is invoked to determine whether the final connection succeeded or failed. If `XMLSocket.connect()` returns `false`, a connection could not be established.

Availability

Flash Media Server 2

Parameters

`host` A string; a fully qualified DNS domain name or an IP address. Specify `null` to connect to the local host. Do not enclose IPv6 addresses in square brackets.

`port` A number; the TCP port number on the host used to establish a connection. The port number must be 1024 or higher.

Returns

A boolean value; `true` if successful, otherwise, `false`.

Example

The following example uses `XMLSocket.connect()` to connect to the local host:

```
var socket = new XMLSocket()
socket.onConnect = function (success) {
    if (success) {
        trace ("Connection succeeded!")
    } else {
        trace ("Connection failed!")
    }
};
if (!socket.connect(null, 2000)) {
    trace ("Connection failed!")
}
```

Note: *Server-side `trace()` statements are output to the application log file and to the Live Log panel in the Administration Console.*

XMLSocket.maxUnprocessedChars

```
myXMLSocket.maxUnprocessedChars
```

The number of characters the connection can receive from the XML server without receiving an end tag or the XMLSocket connection closes. The value of `XMLSocket.maxUnprocessedChars` can be -1 or any value greater than 0. The value -1 means that an unlimited amount of data can be processed. However, the value of `maxUnprocessedChars` cannot exceed the value specified in the Application.xml file. The default value in the Application.xml file is 4096 bytes.

Setting this property in a server-side script overrides the value of the `MaxUnprocessedChars` element in the Application.xml file for each XMLSocket object. If the property is not set in a server-side script, the server uses the value set in the `MaxUnprocessedChars` element of the Application.xml file.

Availability

Flash Media Interactive Server

XMLSocket.onClose()

```
myXMLSocket.onClose = function() {}
```

Invoked when an open connection is closed by the server. The default implementation of this method performs no actions. To override the default implementation, you must assign a function containing custom actions.

Availability

Flash Media Server 2

Example

The following example executes a `trace()` statement if an open connection is closed by the server:

```
var socket = new XMLSocket();
socket.connect(null, 2000);
socket.onClose = function () {
    trace("Connection to server lost.");
}
```

Note: *Server-side `trace()` statements are output to the application log file and to the Live Log panel in the Administration Console.*

XMLSocket.onConnect()

```
myXMLSocket.onConnect = function(success) {}
```

Invoked by Flash Media Interactive Server when a connection request initiated through `XMLSocket.connect()` succeeds or fails. If the connection succeeded, the `success` parameter is `true`; otherwise, `false`.

The default implementation of this method performs no actions. To override the default implementation, you must assign a function containing custom actions.

Availability

Flash Media Server 2

Parameters

`success` A boolean value indicating whether a socket connection is successfully established (`true` or `false`).

Example

The following example defines a function for the `onConnect()` handler:

```
socket = new XMLSocket();
```

```
socket.onConnect = myOnConnect;

socket.connect(null,2000);

function myOnConnect(success) {
    if (success) {
        trace("Connection success")
    } else {
        trace("Connection failed")
    }
}
```

XMLSocket.onData()

```
myXMLSocket.onData = function(src) {}
```

Invoked when a message has been downloaded from the server, terminated by a zero (0) byte. You can override `XMLSocket.onData()` to intercept the data sent by the server without parsing it as XML. This is useful if you're transmitting arbitrarily formatted data packets and you would prefer to manipulate the data directly when it arrives, rather than have Flash Media Interactive Server parse the data as XML.

By default, the `XMLSocket.onData()` method invokes the `XMLSocket.onXML()` method. If you override `XMLSocket.onData()` with custom behavior, `XMLSocket.onXML()` is not called unless you call it in your implementation of `XMLSocket.onData()`.

Availability

Flash Media Server 2

Parameters

`src` A string containing the data sent by the server.

Example

In the following example, the `src` parameter is a string containing XML text downloaded from the server. The zero-byte (0) terminator is not included in the string.

```
XMLSocket.prototype.onData = function (src) {
    this.onXML(new XML(src));
}
```

XMLSocket.onXML()

```
myXMLSocket.onXML = function(object) {}
```

Invoked when the specified XML object containing an XML document arrives through an open XMLSocket connection. An XMLSocket connection can be used to transfer an unlimited number of XML documents between the client and the server. Each document is terminated with a zero (0) byte. When Flash Media Interactive Server receives the zero byte, it parses all of the XML received since the previous zero byte or, if this is the first message received, since the connection was established. Each batch of parsed XML is treated as a single XML document and passed to the `onXML()` handler.

The default implementation of this method performs no actions. To override the default implementation, you must assign a function containing actions that you define.

Availability

Flash Media Server 2

Parameters

`object` An XML object that contains a parsed XML document received from a server.

Example

The following function overrides the default implementation of the `onXML()` method in a simple chat application. The `myOnXML()` function instructs the chat application to recognize a single XML element, `MESSAGE`, in the following format:

```
<MESSAGE USER="John" TEXT="Hello, my name is John!" />.
var socket = new XMLSocket();
```

The following `displayMessage()` function is assumed to be a user-defined function that shows the message that the user receives:

```
socket.onXML = function (doc) {
    var e = doc.firstChild;
    if (e != null && e.nodeName == "MESSAGE") {
        displayMessage(e.attributes.user, e.attributes.text);
    }
};
```

XMLSocket.send()

```
myXMLSocket.send(object)
```

Converts the XML object or data specified in the `object` parameter to a string and transmits it to the server, followed by a zero (0) byte. If `object` is an XML object, the string is the XML textual representation of the XML object.

If the `myXMLSocket` object is not connected to the server (by using `XMLSocket.connect()`), the `XMLSocket.send()` operation fails.

Availability

Flash Media Server 2

Parameters

`object` An XML object or other data to transmit to the server.

Returns

A boolean value; `true` if the server is able to get the socket and the socket state is connected; otherwise, `false`. A `true` value does not mean that the data has been transmitted successfully. The `send()` method is asynchronous; it returns a value immediately, but the data may be transmitted later.

Example

The following example shows how you can specify a user name and password to send the XML object `my_xml` to the server:

```
var myXMLSocket = new XMLSocket();
var my_xml = new XML();
var myLogin = my_xml.createElement("login");
myLogin.attributes.username = usernameTextField;
myLogin.attributes.password = passwordTextField;
my_xml.appendChild(myLogin);
myXMLSocket.send(my_xml);
```

See also

`XMLSocket.connect()`

XMLStreams class

The XMLStreams class is a variation of the XMLSocket class—it has all the same methods, properties, and events, but it transmits and receives data in fragments. To create an XMLStreams object, use the XMLSocket constructor and pass `"stream"` as the parameter. See XMLSocket constructor.

Flash Media Interactive Server can transmit XML data in stream format (for example, as needed by a Jabber server or IM applications). Streaming XML data passes over a normal XMLSocket connection, but it begins with a `stream:stream` tag, contains fragments of XML content, and concludes with a `/stream:stream` closing tag.

The `onData()` handler is invoked and returns complete XML tags whenever it receives them. The `/stream:stream` tag closes the stream. There is an asynchronous call to `onData()` whenever a complete tag has been received by the stream.

Note: *As a security precaution, if 4096 bytes of data arrive before a closing XML tag, the socket connection closes. This value is configurable in the* `XMLSocket.maxUnprocessedChars` *property or in the* `MaxUnprocessedChars` *element in the Application.xml file.*

Availability

Flash Media Server 2

Example

If you want your Flash Media Server application to communicate with a Jabber server, which uses XML streaming, create an XMLStreams object. The XMLStreams object connects to a remote XML streaming server, and the `onData()` handler is called as complete sections of XML occur in the stream.

```
myXMLStreams = new XMLSocket("stream");
```