Writing a Custom Access Control Module

This document contains information for Java programmers who want to write a custom Java access control module (JAM) for Burstware. Before reading this document, you should read and understand the chapter, Burstware Security, in the *Burstware User Guide*.

Burstware provides a pluggable access control interface that allows Burstware administrators to implement a wide range of access control policies. This allows a Burstware administrator, in cooperation with application developers, to control who can access which content, and under what circumstances.

You have several options for deploying access control in your Burstware environment, depending on your needs.

- Burstware default access control

  The default access control functionality that ships with Burstware is simple—it fulfills all play requests.

- Simple Access Control Module (SAM)

  For deployments with relatively modest access control needs, Burst.Com offers as a professional service its Simple Access Control Module (SAM). SAM provides an easy-to-use text-based format, similar to that used by firewalls, for defining access rules. SAM can address a certain class of access control scenarios, but it is not meant to be a complete, fully scalable, or totally flexible access control solution. For more information about SAM, contact your sales representative or email sales@burst.com.

- Custom Java access control module (JAM)

  For deployments that require more sophisticated, scalable, or robust access control than SAM provides, you can write a custom Java access control module (JAM). This document gives an overview of the steps

you need to take to write, test, deploy, and monitor a custom Burstware JAM.

Why Write a Custom JAM?

You may develop a custom JAM because you want to:

- Integrate Burstware with a third-party access control or authentication system
- Interface with a relational database management system that contains access lists or other information used for access control
- Perform access control that uses more than just the limited set of parameters SAM uses (such as IP address and playerId) to determine whether to fulfill a play request
- Perform more sophisticated access control than the regular expression matching provided by SAM
- Provide a more scalable solution than SAM supports

  For example, you may need to define individualized access control policies for many thousands of users, and SAM is not designed to scale to support this.

Access Control Architecture

There are three components to the Burstware access control system. Every play request involves all three components. Each component and the role it plays in implementing access control is described below:

- Burst-enabled player

  The burst-enabled player gathers data about each play request, including identifying information about the user making the request. The burst-enabled player then sends this information to the Burstware Conductor.
- The access control module

  The access control module is a plug-in component for the Burstware Conductor that can be customized or replaced. The access control

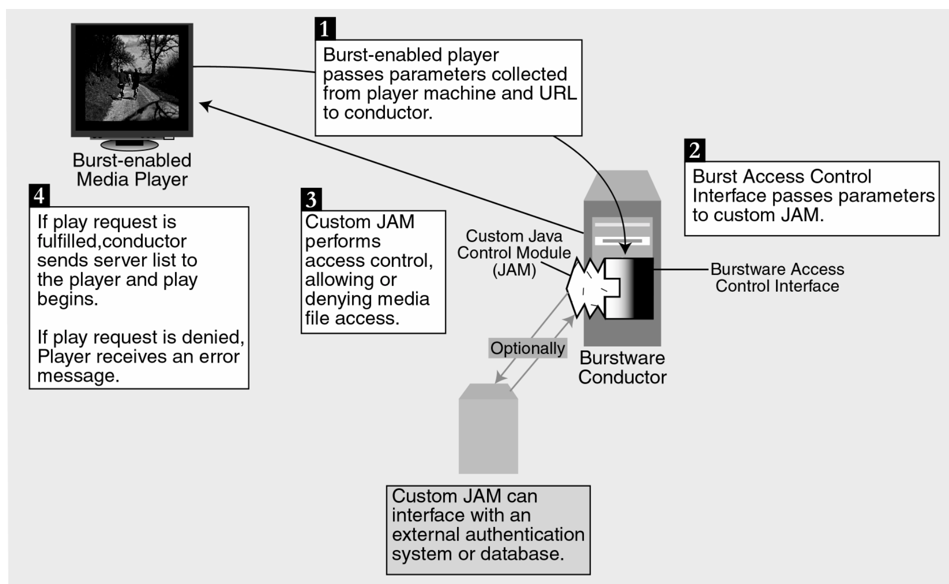
module applies access control policies for each play request and determines whether to fulfill or deny the play request. The default access control module that ships with Burstware applies a simple policy—it allows *all* play requests to be fulfilled.

- The access control interface

  The access control interface is built into Burstware Conductor. It passes the data gathered by the burst-enabled player to the conductor's access control module. The access control module makes the decision to fulfill or deny the play request; the access control interface enforces this request by sending the player machine the server list to fulfill the play request, or an error message when the play request has been denied.

Access Control Sequence

The access control module manages access control each time a play request is made. The following diagram shows the Burstware access control sequence using a custom Java access control module.



Here are the steps in the access control sequence:

1. The client passes parameters collected from the player machine and the play request URL to the Burstware access control interface on the conductor. These parameters include information such as the client IP address and the media file name. The **AccessControlString** parameter, an arbitrary string passed as part of the play request URL, can be used to implement authentication. See the section, Access Control String, on page 4.
2. The access control interface passes parameters to the custom JAM.
3. The custom JAM performs access control, fulfilling or denying the play request.

   There are several options for access control, depending on how you implement your JAM. Common strategies include interfacing with a third-party authentication server or implementing a rules-based system.
4. If the play request is fulfilled, the conductor sends the client a server list, the client contacts the server, and play begins.
5. If the play request is denied, an error message is sent to the client.

Access Control String

In addition to the information Burstware obtains from clients, you can pass any other information from the client machine to the JAM via the **AccessControlString** parameter. This parameter is optionally specified in the URL of the play request. Typically, the web page accessing the content will generate the access control string dynamically.

You can use any scheme to generate the access control string, such as an algorithm implemented in JavaScript. The access control string is passed unchanged to the JAM.

A Burstware application can use an access control string in a number of ways:

- As an encrypted representation of a user name and password.

  For example, when the end user requests a multimedia file, the application might query the end user for a user name and password,

and encrypt this information prior to requesting the media file from Burstware. The encrypted name and password are then passed to Burstware as the **AccessControlString** parameter, so that the JAM can perform user authentication and decide whether or not the request should be fulfilled.

- To store any information that may be useful in logging, since the access control string is written verbatim to the conductor log.

When using the **AccessControlString** parameter, the application generating the play request URL (for example, a web page containing a burst-enabled player) cooperates with the JAM. Typically, the application generating the play request URL uses the access control string as a “key”, and the JAM decides whether the key “fits” the play request. This allows authentication to be performed on each request— the JAM uses the access control string to authenticate the user.

For more information on the **AccessControlString** parameter, see the chapter, Optional Configuration of Burst-Enabled Players, in *Writing A Custom Access Control Module*.

Writing a Custom JAM

To write a custom JAM, you must be a Java programmer with a strong working knowledge of writing scalable, robust, and re-entrant software in multithreaded environments. If you want to interface with third-party applications to help perform access control, you must also understand, and have access to, the APIs for the third-party applications you want to use.

There are two ways to create a custom JAM:

- Use the SAM source code as a starting point, and customize it
- Write a custom JAM from scratch

Customizing SAM Code

The SAM code is included in the **BurstJAMDevKit.exe** self-extracting executable, in the file called **AccessControlImpl.java**.

Even if you plan to write a custom JAM from scratch, it may be helpful to look at the SAM code as a working example of how to write a custom JAM.

Overview of Writing a Custom JAM

The following steps provide an overview of how to write a custom JAM from scratch. These steps are discussed in detail in the remainder of the document.

1. Create a directory in which to develop and compile the JAM.
2. Define a Java class that implements the Burstware access control interface.
3. Write three methods in this class, which:
    - Initialize the JAM and check that the conductor version is compatible with the access control module
    - Decide whether to fulfill a play request
    - Allow the conductor to tell the access control module to clean up any existing connections or processes after a deliberate shutdown

    In writing these methods, you may find you want to define one or more helper classes.
4. Compile the class(es) into a Java archive (JAR).
5. In a test deployment of Burstware, install your JAR into a Burstware Conductor.
6. Test your custom JAM to make sure it is functionally correct and meets your scalability and reliability requirements.
    Debug the code, and fix problems by returning to step 2 as needed.
7. Once you are confident your custom JAM is ready to deploy, install your custom JAM into each Burstware Conductor in your production Burstware deployment(s).

After you introduce the JAM into your production environment, monitor it on a regular basis.

To enhance or modify your custom JAM, return to step 2.

Subsequent sections describe each step in more detail.

Step 1: Create a Development Directory

Your JAM will be part of the Java package **com.burst.accesscontrol**. Because of Java naming conventions, you may want to place your Java files in a directory ending with "com/burst/accesscontrol."

For example, if you put your development directory under /usr/local/burst/JAMs/test, your full directory path would be:

/usr/local/burst/JAMs/test/com/burst/accesscontrol

Put all Java classes you write for the JAM in this directory.

Step 2: Define the Access Control Class

Define a class that implements the access control interface. The class must be named **AccessControlImpl**, and it must implement the Java interface **com.burst.accesscontrol.AccessControlIfc**.

A simple example class that conforms to the **AccessControlIfc** interface is provided in the Examples/Conductor/AccessControl subdirectory of your Burstware installation. You can copy the class from this directory into your development directory and use it as a starting point.

Step 3: Implement the initialize Method

The **initialize** method of the **AccessControlImpl** class initializes the access control module and validates a conductor version string. The format of the version string is:

```
" Burstware Conductor - Release_number date"
```

For example:

```
" Burstware Conductor - Release 1.2.3 2/4/2000"
```

The first character in the version string is a space. The double quotes around the string are not part of the string syntax.

The **initialize** method is called only once, when the conductor loads the access control plug-in on startup. This is a good place to add code to establish connections to external databases or authentication systems.

If the conductor version string is valid, and no runtime errors occur in your code, the **initialize** method should return a String variable containing the version of the JAM plug-in. The conductor logs the JAM plug-in's version string in its log file so that users can ascertain what version of the plug-in is being used.

If the conductor version string is invalid, and no runtime errors occur in your code, the **initialize** method throws an exception of type **com.burst.accesscontrol.FatalException**.

Since 2.0 is the first version of Burstware Conductor that supports the access control interface, a 2.0-level JAM does not need to validate the conductor version string.

Step 4: Implement the verifyRequest Method

The **verifyRequest** method is the cornerstone of Burstware's access control functionality; it decides whether to fulfill or deny the play request.

This method takes as input parameters all the information available about the media file request, such as the media file name and the IP address of the client making the request. See the section, verifyRequest (method), on page 19 for the complete definition of this method.

In the **verifyRequest** method, you can implement simple or sophisticated logic to decide whether or not to fulfill a play request. You can interface with a file (as Burst.Com's SAM does), a relational database, or an external authentication system.

If your **verifyRequest** method is complex, you probably want to define one or more helper classes. Keep all of these classes in your development directory.

To accept a play request, simply issue a return from the method.

To deny a play request, throw an exception of type **com.burst.accesscontrol.AccessDeniedException**. You must create an instance of this class, and set the two strings which are attributes of this class. The first string in the constructor is the error message displayed to the user, and the second is only written to the conductor log. The second string can be used for debugging and/or auditing purposes. See the section, AccessDeniedException (class), on page 23 for the complete definition of this class.

In certain cases, your method may detect that something has happened that requires that the conductor exit immediately. For instance, SAM will do this if there is an error in the access rules file it uses. To cause the conductor to shut down at the earliest opportunity, the **verifyRequest** method should throw an exception of type **com.burst.accesscontrol.FatalException**. See the section, FatalException (class), on page 24, for the complete definition of this class.

verifyRequest Implementation Guidelines

Make **verifyRequest** as efficient as possible, as it is run for every play request. An inefficient **verifyRequest** can significantly reduce the amount of concurrent play requests that can be serviced.

It is critical that the **verifyRequest** method be re-entrant, since it may be called simultaneously by multiple conductor threads. Thus, any access to shared resources must be properly synchronized. If you do not have experience writing re-entrant code for multithreaded environments, you should familiarize yourself with this methodology before writing a JAM.

Step 5 - Implement the shutdown Method

The **shutdown** method is called during a deliberate conductor shutdown and allows the access control module to clean up any existing connections or processes, such as connections to external authentication systems or databases.

NOTE: If the conductor is killed unexpectedly, the shutdown method may not be called.

Step 6 - Compile your Custom JAM

Once you have written at least part of your JAM, you will want to compile it. To do this, you must first properly set your **CLASSPATH** environment variable to include the Burstware Conductor JAR.

The supporting classes for the JAM—the interface and the two Exception classes—reside in the Burstware Conductor JAR file. This JAR file is called **BurstConductor.jar** under Windows, and **burstconductor.jar** under UNIX. Therefore, your Burstware Conductor JAR file must be in the **CLASSPATH** environment variable when you compile your **AccessControlImpl.java** file.

In a Burstware installation in which the conductor is installed, the Burstware Conductor JAR file can be found in the **conductor** subdirectory under Windows, and in the **lib** subdirectory under UNIX.

After properly setting your **CLASSPATH**, use any standard Java Development Kit, version 1.1.5 to version 1.1.8, to compile the **.java** files into **.class** files.

When you are ready to test your JAM from within a Burstware Conductor, archive your **AccessControlImpl.class** file, and any other helper classes, into the Burstware access control JAR file. You can use the **jar** utility, for example, to create your archive file. Your archive file must be called **BurstAccessControl.jar** under Windows, and **burstaccesscontrol.jar** under UNIX. Note that your class files must reside in the **com/burst/accesscontrol** directory within the JAR file.

Step 7 - Installing and Testing a JAM

Before deploying your new JAM in a production environment, Burst.Com recommends extensive testing to ensure your custom access control module is:

- Functionally correct
- Scalable—it can handle the number of concurrent connections you anticipate the conductor needs to handle
- Re-entrant—it can handle many simultaneous requests without thread-related bugs

- Robust—for example, it does not contain any memory leaks, and it can handle unusual input parameters gracefully

Test your JAM by installing it into a test Burstware deployment that is not accessible to the general public.

Installing the JAM

To install a JAM, follow these steps:

1. Shut down the test conductor.
2. In the same subdirectory of your Burstware installation that contains **burstconductor.jar**, replace the old **BurstAccessControl.jar** with the new **BurstAccessControl.jar**. Burst.Com suggests saving the old JAR somewhere, just in case.
3. If your JAM requires third-party JAR files, put them in the same directory as the one containing **burstconductor.jar**.
4. If you are using third-party JAR files, specify the locations of these files so a conductor can find them.

   How you specify this depends on the platform you are running under and whether you are running the conductor as an NT Service:

Under UNIX

Under UNIX, edit the **burst_process.sh** file in the Burstware installation directory, setting the ADDITIONAL_CONDUCTOR_CLASSPATH variable to the location of the third-party JAR file(s), as in the following example:

```
ADDITIONAL_CONDUCTOR_CLASSPATH=”./lib/
StarwaveRegexp.jar”
```

Windows NT

Under Windows NT, if you are not running Burstware Conductor as an NT Service, add the **classpath** parameter to the Burstware Conductor command line, as in the following example:

```
.\BurstConductor.exe
-classpath .\StarwaveRegexp.jar -ini
burstconductor.ini
```

Windows NT running Burstware Conductor as NT Service

Under Windows NT, if you are running Burstware Conductor as an NT Service, you must run the command below from an MS-DOS prompt, in the same directory as the Burstware Conductor. This command reinstalls the conductor as an NT Service, and makes the conductor refer to the specified **classpath** parameter each time it starts up.

```
.\BurstConductor.exe -install -id=1
-classpath .\StarwaveRegexp.jar -ini
burstconductor.ini
```

5. Restart the test conductor.

Testing a JAM

Test a JAM by connecting players that should be denied and those that should be allowed. Verify that your JAM denies and fulfills requests as expected. If possible, automate this process, so that you can test how your JAM handles hundreds or thousands of clients—simulating as closely as possible the most stressful situation you think your production deployment will encounter.

You can monitor and debug a JAM in several ways, including analyzing the conductor log files. See the section, Monitoring and Debugging the JAM, on page 14.

Step 8 - Deploying a Custom JAM

Once your JAM is thoroughly tested, you are ready to deploy it in your production Burstware deployment.

Ensuring Consistent Access Control Policies

If you want to apply the same access control policies across all your conductors, you must give each of the running JAMs access to the same data that determines the policies. If your policies are defined in a simple file, this can be accomplished by doing a periodic FTP transfer of the file from its master location to all other conductors. You can also have all JAMs call out to a common access control server on the network to provide access control. Other data sources may require different synchronization strategies.

Deploying with Downtime

If your deployment can tolerate downtime, follow these steps to install your JAM:

1. Shut down the conductor(s) in your deployment.
2. In the subdirectory of your Burstware installation that contains the **BurstConductor.jar**, replace the old **BurstAccessControl.jar** with the new **BurstAccessControl.jar**. Burst.Com suggests saving the old JAR somewhere, just in case.
3. If your JAM requires third-party JAR files, put them in the same directory as the one containing **burstconductor.jar**.
4. If you are using third-party JAR files, specify the locations of these files so a conductor can find them. See the section, Installing the JAM, on page 11, for information on how to specify the locations.
5. Restart the conductor(s).
6. Have some players connect to verify the JAM is functioning properly.

Deploying with No Downtime

If your system doesn't allow downtime, you can use the following method to install your JAM. Since your system cannot tolerate downtime, we assume that each conductor in your deployment has been set up to have a backup conductor.

1. Stop the backup conductor.
2. Install the JAR (as described in step 2 in the section, Deploying with Downtime, on page 13) for the backup conductor.
3. Start the backup conductor.
4. Have test players connect to the backup conductor, to verify that your JAM is working properly.
5. Stop the primary conductor. (The backup conductor will temporarily take over.)
6. Install the JAR (as described in step 2 in the section, Deploying with Downtime, on page 13) for the primary conductor.
7. Restart the primary conductor. It will take over from the backup conductor.

8. Have test players connect to the primary conductor to ensure that the new access control is working properly.

NOTE: If you encounter problems, contact Burst.Com customer support or send an e-mail to support@burst.com.

Monitoring and Debugging the JAM

The Burstware conductor log file contains, among other things, a record of all play requests, as well as a record for each request denied by the access control module. The conductor log—called **burstconductor.log**, by default—is useful in monitoring and debugging your JAM. You can also build your own logging capability into your JAM, using a separate log file.

To analyze a conductor log, examine the log file directly or use the Burstware Log Toolkit—available as a professional service from Burst.Com—to help you analyze the logs.

The following is an excerpt from a conductor log file, containing these three lines:

- A SignOn message (phase one of a play request)
- An Open message for the same client (phase two of a play request)
- An access denied message for the same client, indicating the play request was denied by the access control module

```
1-May-00 8:22:10 PM TRACE: Client1: Command:
1|SignOn|1|0|1|0|115144021|100000|192.168.0.32|gfriedman|tru
e|WMPUser_{99522540-e7a9-11d2-a564-0050046852c0}_256|Greg
Friedman|none|1.9|none|BURST.COM|mplayer2.exe


11-May-00 8:22:10 PM TRACE: Client1: Command:
1|Open|1|0|sample1.asf|false|1500


11-May-00 8:22:10 PM ERROR: Client1: Access denied: Access
to asf files is currently disallowed.
Client1: Access denied details: Burstware SAM: access is
denied
Client IP address: 192.168.0.32
```

```
Media filename: "sample1.asf"
Player ID: "WMPUser_{99522540-e7a9-11d2-a564-
0050046852c0}_256"
Memory cache only: true
Access control string: "none"
Rule: "deny .* .*\.asf .* .* .* Access to asf files is
currently disallowed"
Line number: 64
Client1: Access denied messages: End

11-May-00 8:22:11 PM TRACE: Client1: Command: 1|Exit|1|0
```

An “access denied” entry log includes:

- The client session ID
- The error message displayed to the end user explaining the denial
- An internal error message giving details of the denial

  When using SAM, this error message includes the text of the rule responsible for the denial.

For more information on conductor logs, see “What’s in a Log File?” in the chapter “Conductor and Server Runtime Management” in *Writing A Custom Access Control Module*.

Interface and Class Descriptions

This section documents the Java interface and classes which comprise the Burstware access control interface.

AccessControlIfc (interface)

Usage

```
public interface AccessControlIfc
```

Description

This class defines the interface used to add access control capabilities to the Burstware Conductor. A user-provided implementation of this class is used to verify new requests for service received by a conductor.

The Burstware Conductor contains a JAR file named **./BurstAccessControl.jar** (or **./burstaccesscontrol.jar** under UNIX). The JAR file grants access to all play requests.

During startup, the conductor searches its installation directory for a file named **BurstAccessControl.jar**.

- If **BurstAccessControl.jar** does **not** exist, the conductor logs the message "No Access Control Plug-in" to its log, and will not perform any access control.
- If **BurstAccessControl.jar** exists, the conductor assumes the JAR file is included in the **CLASSPATH** environment variable and attempts to load the class **com.burst.accesscontrol.AccessControlImpl** via the Java **Class.forName** method. A new instance of this class is instantiated via the **Class.newInstance** method, and cast to the interface **AccessControlIfc**. All requests for service are authorized via the **verifyRequest** method described below. As part of the play request, a user-supplied access control string is sent to the conductor. The JAM can use this string to perform authentication.

Implementation Notes

Only a single implementation of this interface is possible within a Burstware Conductor installation. If the conductor is replicated, you should ensure that the same access control plug-in is installed on both conductors.

Interface Upgrades

Burstware upgrades of this interface are versioned by subclassing. For example, a new version of this interface may be implemented as a subclass named **AccessControlIfc1**, which is shipped in **BurstConductor.jar** in addition to **AccessControlIfc**.

If the user plug-in only implements **AccessControlIfc**, the conductor receives a **ClassCastException** when attempting to cast to **AccessControlIfc1** as described in the section, AccessControlIfc (interface), on page 15.

The conductor then attempts to cast to an older version of **AccessControlIfc** until it succeeds, or until all previous versions are exhausted. If the user's plug-in cannot be cast to any recognized version of **AccessControlIfc**, the conductor performs a System.exit(-18).

Methods

Methods defined in the **AccessControlIfcl** interface are described in the table below:

Table 1: AccessControlIfc Methods

Method	Description
initialize(String,String)	Initializes the access control module and allows the plug-in to verify the conductor version string.
verifyRequest(long, long, String, String, boolean, String, String, String, String, String, String, String, int, String, String)	Verifies the given access request is allowed. Throws an exception if the request is denied.
shutdown( )	Allows the access control module to clean up any existing connections or processes during a deliberate conductor shutdown.

Additional details on the methods defined in the **AccessControlIfcl** interface are proved in the sections that follow.

initialize (method)

The **initialize** method initializes the access control module and allows the plug-in to verify the conductor version string.

Usage

```
public abstract String initialize(String
conductorVersion, String iniFileName) throws
FatalException
```

Parameters

The parameters for the **initialize** method are described in the following table.

Table 2: initialize parameters

Parameter	Description
conductorVersion	String containing the conductor's version information.
iniFileName	String containing name of the conductor's ini file.

Returns

An identifier string for the JAM. This string is logged in the conductor’s log.

Throws

FatalException

If the conductor version string does not match the JAM’s expectation, the JAM should throw a FatalException which causes the conductor to shut down.

verifyRequest (method)

This method is used to verify that access to the given media file is allowed.

Usage

```
public abstract void verifyRequest(long bufferSize,
long networkSpeed,
String clientIPAddress,
String clientHostName,
boolean memCacheOnly,
String playerId,
String lastSuccessfulServer,
String burstwareBridgeVersion,
String accessControlString,
String networkDomain,
String hostExe,
String mediaFileName,
int mediaPlayRate,
```

```
String allSignOnParameters,
String allOpenParameters) throws
AccessDeniedException, FatalException
```

Parameters

The parameters for **verifyRequest** are shown in the following table:

Table 3: verifyRequest parameters

Parameter	Description
bufferSize	Buffer size, in bytes, used by requesting player.
networkSpeed	Maximum network speed, in Kbps, reported by requesting player (the value of the **UserNetworkBandwidth** parameter).
clientIPAddress	The IP address of requesting machine.
clientHostName	The host name of requesting machine.
memCacheOnly	Set to `true` if the requesting player is guaranteed to be using a memory-based buffer, not a disk-based buffer.

Parameter	Description

playerID	A string containing three pieces of information identifying the player making the request.
	• The type of player is being used (e.g.,Windows Media Player, QuickTime Player, JMF, etc.)
	• The unique ID of the player (such as the Windows Media Player's GUID)
	• The processID of the player process
	
	Windows Media Player example:
	`WMPuser_{def197d0-3bc3-11d3-8096-005004d38f7f}_323`
	
	Apple QuickTime Player example:
	`QuickTime Player_199`

lastSuccessfulServer	The name of the server last connected to, immediately before this reconnect request. Used only when a player has failed over from one server to another.
burstwareBridgeVersion	The version string of the Burstware Bridge making the play request.
accessControlString	An arbitrary string optionally passed in as part of the play request URL.
networkDomain	The network domain on which the requesting host resides.
hostExe	The name of the application making the play request.

Parameter	Description
mediaFileName	Name of requested media file. This can contain directories - e.g., "wrestling/WWF/ Andrethegiant.asf"
mediaPlayRate	The encoding rate, in Kbps, of the requested media file, as specified in the **MediaPlayRate** parameter.
allSignOnParameters	A list of all parameters sent as part of the SignOn message, from the burst-enabled player to the conductor, containing information about the Burstware Bridge. This is a "|" delimited String containing SignOn message parameters. The format is: sessionId | SignOn | SignOnMajorVersion | SignOnMinorVersion | clientMajorVersion | clientMinorVersion | bufferSize | networkSpeed | clientIPAddress | clientHostName | memCacheOnly | playerId | userName | lastSuccessfulServer | BurstwareBridgeVersion | AccessControlString | NetworkDomain | HostExe
allOpenParameters	A list of all the parameters sent as part of the Open message, from the burst-enabled player to the conductor. This is a "|" delimited String containing Open message parameters. The format is: sessionId | Open | OpenMajorVersion | OpenMinorVersion | fileName | autoStart | mediaPlayRate

Throws

AccessDeniedException - if the play request is denied.

FatalException - if the plugin experiences a fatal error condition that should cause the conductor to shutdown.

Shutdown (method)

This method is called during a deliberate conductor shutdown and allows the conductor to tell the access control module to “clean up” any existing connections or processes.

Usage

```
public void shutdown ()
```

AccessDeniedException (class)

Usage

```
public class AccessDeniedException (class)

extends BurstException
```

Description

This class is used by a user-supplied access control plug-in on the conductor to deny a play request.

NOTE: The constructor only allows Strings. If your plug-in makes a C callout to perform access control and the C callout only returns an integer code to indicate status, have the plug-in translate the integer code into readable text before throwing this exception.

Constructor Index

```
AccessDeniedException(String, String)

Constructor
```

Method Index

getErrDetails()

Retrieves the detailed error string for conductor logging.

Constructors

AccessDeniedException

```
public AccessDeniedException(String errText, String
errDetails)
```

```
Constructor
```

Parameters

The parameters for the **AccessDeniedException** class are described in the following table.

Table 4: AccessDeniedException Parameters

Parameter	Description
errText	A string containing error text to be displayed to end user.
errDetails	A string containing detailed information that is logged only in the conductor's log.

Methods

getErrDetails

Usage

```
public String getErrDetails()
```

Retrieves the detailed error string for conductor logging.

Returns

Detailed error string.

FatalException (class)

Usage

```
public class FatalException
```

```
extends BurstException
```

Description

A user-supplied access control plug-in uses this class to indicate fatal errors. This exception should only be thrown when the plug-in detects a condition that should cause the calling conductor to terminate execution.

The constructor only allows strings as arguments. If your plug-in makes a C callout to perform access control and the C callout only returns an integer code to indicate status, have the plug-in translate the integer code into readable text before throwing this exception.

Constructor Index

`FatalException(String)`

`Constructor`

Constructors

FatalException

`public FatalException(String errText)`

`Constructor`

Parameters

errText - String containing error text which is logged in conductor log and displayed to end user.