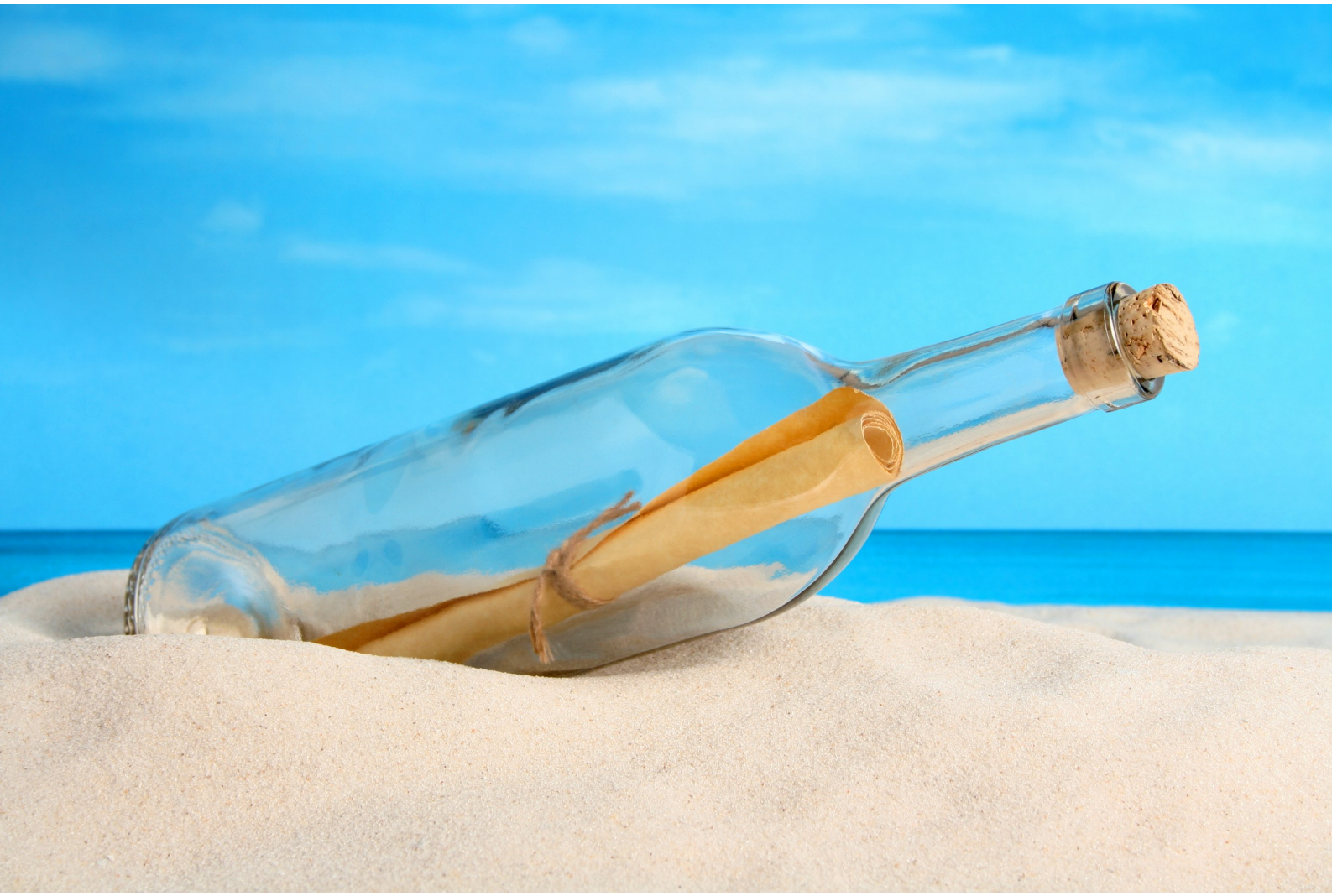

Cloud Alert
Next Generation Notifications

The Cloud Alert Messaging Gateway

Plugin Writers Guide

Table of Contents

Introduction

The Cloud Alert Messaging Gateway is an on-site deployed software that enables organizations to automate their messaging requirements. It can be used to generate alerts for financial transactions, OTPs and other events.

The Messaging Gateway integrates with the organization's database through the use of plugins, which are Java classes placed into the c:\cag\dbc\plugins directory. The creation of such plugins is detailed in this guide.

If you do not have a copy of the messaging gateway, Starter editions of the software are available for free at:

http://cloudalert.cloud

Prerequisites

The following requirements should be met for plugin development:

Developer

- Awareness and experience in Java coding, development and compilation
- Awareness of CLASSPATH, how it is used, and how to set it up
- Awareness of .bat files
- Basic awareness of multi threading, and its effects
- Awareness of SQL and related database-access mechanisms

Software

- A Cloud Alert Messaging Gateway installation that has been configured and setup.
- Any text editor
- Any SQL-based database, with JDBC drivers
- Web browser (Mozilla Firefox 45.0 or higher / IE 11 or higher / Google Chrome / Microsoft Edge)

Adding a Database Driver

For your plugins to connect to your databases, you may need to add additional database drivers. The Messaging Gateway already includes database drivers for Oracle and SQL Server. Note that an Oracle driver is required for the Messaging Gateway to connect to its private store.

The drivers are maintained in the c:\cag\jdbc directory. Each driver is a jar file, and has to be renamed to a filename from d1.jar to d9.jar. This mechanism has been so designed since the Messaging Gateway's classpath cannot be changed, thus necessitating file renames.

In the default installation, the files d1 and d2 are the drivers for Oracle and SQL Server respectively. The files d3 to d9 are dummy files - replace them with your own driver jar.

Since renaming a file may cause you to lose track of them, a comments text file has been included. Ensure that you update the comments file on any change.

Additionally, its recommended that a copy of the driver jar files be placed in the 'drivercopy' directory, with their original names. While this directory is not used by the Messaging Gateway, it is a useful reference to assist you.

Note that the default Oracle driver is not compatible with Oracle 12c. It only supports SIDs, not service names - update to the latest jdbc driver to avail of these features. A newer driver 'ojdbc7.jar' is present in the drivercopy directory - replace d1.jar with it if required. Note that this may affect the connectivity between the Messaging Gateway and its private store; its recommended to contact support before changing the d1.jar file or adding additional Oracle drivers.

The Messaging Gateway is compatible with class versions upto Java 8. Note that updating the Java installation on your OS does not increase this limit.

Creation of the Plugin

A plugin is created by creating a Java class that descends from the abstract class DBCDBPlugin_3, which is the ancestor for all database-access plugins. The '3' represents a version number, and has been embedded in the name so as to make your created plugins immune to changes in future plugin mechanisms.

On extending DBCDBPlugin_3, the abstract functions in it will have to be implemented. They will be called by the system on various events, and the code in these functions thus implements the plugin mechanism.

A sample plugin has been provided in the c:\cag\dbc\plugins directory, alongwith the table structure it has been tested with. You can alter this plugin to match your database structure.

The event functions are detailed below:

Name	**init**

Syntax	`public DBCSQLConnectionTarget init(`
	`String param1, String param2, String param3,`
	`String param4, String param5, String param6,`
	`String param7, String param8, String param9,`
	`String param10)`

Description	This event is called for the plugin to initialize itself. It is called only once during the life cycle of the plugin. It is guaranteed that it will be the first call to the plugin, and no other calls will take place until it has completed.
Parameters	The parameters param1.10 are the values of the parameters specified in the plugin configuration page of the Messaging Gateway's Administration Portal. These parameters can be used for a variety of purposes, mainly as a mechanism for customizing the plugin by the end-customer. Typical uses are: • Database Connection String • Database Driver Class Name • Database Username • Database Password • Logging Options • Debugging Options All the parameters are string fields, and can be used in any way required.
Return value	The plugin should return a valid DBCSQLConnectionTarget object to denote that it has successfully initialized. This will be used as the target database against which all sql statements of this plugin will be executed. Any errors during the plugin initialization because of which this plugin should not be started, can be notified to the Gateway by returning ***null***. The gateway will then ignore this plugin until the next Gateway restart.
Responsibilities	The plugin will typically create a 'DBCSQLConnectionTarget' object, that represents a target to a database. More information on targets is available later in this document.
Multi threading	This event is called from the main plugin thread. For more information, see 'Multi threading Notes' below.

Name	**getPollSqlStatement**
Syntax	`public String getPollSqlStatement(String paExemptionSqlClause);`
Description	This function should return the sql statement to query the database.
Parameters	paExemptionSqlClause: This string field holds a part of a WHERE clause which ignores records that are currently being processed. This clause should be added into the sql statement being returned. This clause typically would look like: `AND (OUTMSGQID<> 1) AND (OUTMSGQID <> 2)` Note that the exemption clause may be blank, but never null. Note that it starts with a space, to prevent typical sql-statement generation errors. The first word in it will be 'AND', when the statement is non-blank. Failure to add the exemption clause into your sql statement will result in multiple messages being sent for the same record.
Return value	This event should return the complete sql statement which will be used to query the database.

Responsibilities	This function should return the sql statement that will then be used to query the database.
	
	The sql statement should return a result of records, ie, it should be a SELECT on a table, view, or other stored procedure that returns results, of which 1 record corresponds to a message to be sent.
	
	It should also include clauses that eliminate unnecessary records, for example, scheduling can be implemented by adding a date checking clause to this statement.
	
	It is compulsary for it to include the exemption clause. Note that the exemption clause starts with an 'AND' when non-blank, so that the developer does not need to add it.
	
	A sample code could be like:
	
	`return("SELECT * FROM OUTMSGQ WHERE STATUS='F'"`
	
	`+paExemptionSqlClause);`

Multi threading	This event is called from the main plugin thread. For more information, see 'Multi threading Notes' below.

Name	**getPacketFromCurrentRecord**

Syntax	`public DBCOutSMSPacket getPacketFromCurrentRecord(DBCSQLQuery q)`
	
	`throws java.sql.SQLException;`

Description	This event is called when the system needs to extract SMS packet information from the current database record.
Parameters	The DBCSQLQuery query component is passed to the event. The fields in the current record of this object should be used to generate the SMS packet.
Return value	This event should generate a DBCOutSMSPacket packet, which should include all the information required to send an SMS. The event can return null if the current record does not need to be sent for any reason. This can be used to implement last-minute filtering of messages to be sent, though for performance reasons, it is recommended that most filters be implemented through the sql statement clauses in *getPollSqlStatement.*
Exceptions	This event may throw an exception. Throwing of an exception causes the plugin to stop processing the existing resultset, recheck the database connection, and then requery the database.

Responsibilities	The code in this event should create a DBCOutSMSPacket, read fields from the query object, and place them into the packet.
	
	Typical code will look like:
	
	`DBCOutSMSPacket Result=new DBCOutSMSPacket();`
	
	`Result.setPhoneNo( q.getString("DESTPHONENO") );`
	
	`Result.setContent( q.getString("CONTENT") );`
	
	`Result.setRequestReceipt( q.getBoolean("REQUESTRECEIPT") );`
	
	`Result.setSendAsFlash( q.getBoolean("SENDASFLASH") );`
	
	`Result.setSenderName( q.getString("SENDERNAME") );`
	
	`Result.setSenderNameIsAlpha( true );`
	
	`return(Result);`
	
	This event is also free to obtain information from sources other than the query object, for example, hardcoded information, other sql statements on the same or other databases. The event may also use additional information present in the query object to correlate on queries on other databases.

Multi threading	This event is called from the main plugin thread. For more information, see 'Multi threading Notes' below.

Name	**getPrimaryValuesFromCurrentRecord**

Syntax	`public DBCDBOutQueuePrimaryValues getPrimaryValuesFromCurrentRecord(DBCSQLQuery q)`
	
	`throws java.sql.SQLException;`

Description	This event is used to extract identifying information from the record.
Parameters	The query object is passed to the event.
Return value	This event should return a newly created DBCDBOutQueuePrimaryValues object, which should include information that uniquely identifies the current record. The event can return null if the current record does not need to be sent for any reason. This can be used to implement last-minute filtering of messages to be sent, though for performance reasons, it is recommended that most filters be implemented through the sql statement clauses in *getPollSqlStatement.*
Exceptions	This event may throw an exception. Throwing of an exception causes the plugin to stop processing the existing resultset, recheck the database connection, and then re-query the database.

Responsibilities	This event should generate a DBCDBOutQueuePrimaryValues object, and load it with data from the current record of the query that will be used to uniquely identify the current record. In most cases, the primary key fields of the resultset in the query can be used. The information placed into this object will be used later to identify the record to update it status once the message is succesfully sent, as well as while generating the exemption sql clause (through the getExemptionSqlForOnePendingRequest function).
	
	The DBCDBOutQueuePrimaryValues has several fields, any of which can be used, as long as the same fields are used consistently.
	
	A typical code for this function would be like:
	
	`DBCDBOutQueuePrimaryValues Result=new DBCDBOutQueuePrimaryValues();`
	
	`Result.setPrimaryField4( q.getInt("OUTMSGQID") );`
	
	`return(Result);`
	
	The information posted into this object is used only while the system is operational, and does not exist beyond a system restart.

Multi threading	This event is called from the main plugin thread. For more information, see 'Multi threading Notes' below.

Name	**getExemptionSqlForOnePendingRequest**
Syntax	`public String getExemptionSqlForOnePendingRequest(DBCDBOutQueuePrimaryValues val);`
Description	This event should return the sql clause part that exempts the record specified by the parameter.
Parameters	The DBCDBOutQueuePrimaryValues that represents a single unique record.
Return value	The string return value should contain the part of the sql clause which causes the main sql statement to ignore the record specified in the parameter. Typically, the return value will be like: `return( "OUTMSGQID <> "+val.getPrimaryField4() );` This event should compulsarily return a valid sql clause. The clause should be complete by itself, and should not start with an AND or OR.
Exceptions	This event should not throw any exceptions.
Responsibilities	Returning the sql clause as specified. The clause can be built out of the fields of the DBCDBOutQueuePrimaryValues object that have been populated in the getPrimaryValuesFromCurrentRecord event.
Multi threading	This event is called from the main plugin thread. For more information, see 'Multi threading Notes' below.

Name	`updateDatabaseOnSendResponse`
Syntax	`public void updateDatabaseOnSendResponse(DBCMessageSendStatusType success, DBCDBOutQueuePrimaryValues prival, String MessageID, String[] AllMessageIDs);`
Description	This event should update the database regarding the status of the send.
Parameters	The DBCMessageSendStatusType object holds the status of the send of the message. The DBCDBOutQueuePrimaryValues identifies the record for which the message was sent. The String MessageID is the ID generated by the remote server for the message. This will be valid only if the message send was successful, otherwise it may be blank or null. If multiple SMSs were generated due to concatenation, it returns the MessageID of the last SMS, after confirming that all the SMSs were sent successfully. The String array AllMessageIDs contains the message ids for all the concatenated parts of this message. If the message did not require concatenation, only AllMessageIDs[0] will have a valid value. If concatenation was used, the number of non-null values will match the number of parts that were used.
Return value	void
Exceptions	This event should not throw any exceptions.
Responsibilities	The record specified must be updated to reflect the send status. The send Status can be obtained from the DBCMessageSendStatusType object, of which more details are provided separately. It is also recommended that the MessageID or AllMessageIDs be stored, either in the same table, or a separate one. This is essential for correlating the delivery reports that are received by the system to the send records.
Multi threading	This event can be called by multiple threads, and simultaneously vis-a-vis the main plugin thread. Ensure that your code meets the multi threading requirements listed in the 'Multi threading Notes'.

Name	**handleReceivedSMS**
Syntax	`public boolean handleReceivedSMS(DBCInSMSPacket packet);`
Description	This event is called when the system receives an incoming (MO) SMS message.
Parameters	The DBCInSMSPacket object that contains the details of the received message.
Return value	The event should return *true* if it has accepted, ie, saved this SMS message. If the event returns *false*, the message will be sent to the next plugin defined after this one, and so on, until one of the plugins accepts it.
Exceptions	This event should not throw any exceptions. Exceptions should be caught and handled to make the event return '*false*'.
Responsibilities	This event can do one or more of the following: • Save this message into a table that stores received messages. • Respond to the message by inserting a record into the polled outbound table. • Evaluate the contents of the message, and choose to handle or ignore it. Since a received message is sent to all plugins until accepted, plugins can be designed to receive and respond to a certain type of message, and ignore others (returning false). For example, a courier organization may implement a plugin that responds to SMSs having the content 'TRACK' followed by a tracking number. They may then temporarily add another plugin to respond to their raffle draw results, which uses the 'RAFFLE' followed by the raffle ticket number. Thus, multiple plugins that receive messages and respond can coexist in the same system, and are unaffected by the installation of other plugins.
Multi threading	This event can be called by multiple threads, and simultaneously vis-a-vis the main plugin thread. Ensure that your code meets the multi threading requirements listed in the 'Multi threading Notes'.

Name	**handleReceivedStatusReport**
Syntax	`public boolean handleReceivedStatusReport(DBCInStatusReport packet);`
Description	This event is called when the system receives an incoming Status report (receipt).
Parameters	The DBCInStatusReport object that contains the details of the received status report.
Return value	The event should return *true* if it has accepted, ie, used this status report. If the event returns *false*, the status report will be sent to the next plugin defined after this one, and so on, until one of the plugins accepts it.
Exceptions	This event should not throw any exceptions. Exceptions should be caught and handled to make the event return '*false*'.
Responsibilities	This event can do one or more of the following: • Update the delivery status of the database record that originated the outbound message, or a separate table that holds MesageIDs of sent messages. • Save this message into a table that stores received status reports. • Respond to the status report by performing some other action, based on the success or failure of the message. For example, another message to a secondary number may be sent on a failure report, by inserting a record into the polled outbound table. Since a received status report is sent to all plugins until accepted, plugins can be designed to receive and respond to a certain type of status report, or only status reports whose MessageIDs are found in the database, and ignore others (returning false). For example, 2 plugins may be present sending messages from 2 separate databases, with MessageIDs being stored in each database for its sent messages. A received status report can poll through the plugins, and the plugin will return true only if it finds the equivalent record in its database. Note that due to concatenation of outbound messages, there may be multiple status reports received for a single outbound record. Thus, the failure to map a status report to a send record should not be treated as an error condition.
Multi threading	This event can be called by multiple threads, and simultaneously vis-a-vis the main plugin thread. Ensure that your code meets the multi threading requirements listed in the 'Multi threading Notes'.

Compiling Plugins

Once you have created your plugin's .java file as per the instructions in the Plugin Developers Guide, you need to compile it to create a .class that can be loaded by the gateway. The javacplugin.bat file in the \cag\dbc\plugins directory will help in the compilation process.

- Open a command prompt, and cd to the \cag\dbc\plugins directory.
- Enter 'javacplugin myjavafile.java', replacing myjavafile.java with the actual name of your java file.
- The compilation process will either display errors, warnings, or nothing if the compilation was successful. Ensure that the .class file is now present.
- The name of the .class file, without its extension, can now be entered into the Administration Console - Configure Plugins - Add - Plugin Class Name. Click Save.
- The gateway must now be restarted for the plugin to be loaded. The View Logs section displays any errors during the loading of the plugin.

Note that your plugin can comprise of multiple classes. In this case, compile all of the java files into classes, but enter the name of only the main file in the Configure Plugins section. The main file is the file that extends DBCDBPlugin or DBCGenericPlugin.

If you wish to use a third-party IDE (for example, Eclipse or Idea) to create your plugins, ensure that the \cag\dbc\pluginstubs directory is added or copied into your project. The database drivers in \cag\lib\jdbc may also need to be added.

Deploying plugins

To deploy a plugin, place the class file and any other dependent files in the c:\cag\dbc\plugins directory. If your class is part of a package, the appropriate directories will have to be created within this directory. Any jar files will have to be unzipped and placed here with their appropriate directories.

Next, use the browser-based administration console to add the plugin to the gateway. Select Manage Plugins - Add, and enter the name of the class (excluding the '.class' extension) into the box provided. Any parameters to be passed to the init function can also be entered here.

The class which extends DBCDBPlugin_x is the name to be entered. Other classes that this main class refers to do not need to be entered - they only need to be present in the c:\cag\dbc\plugins directory.

Click Save. The gateway will now restart. Select View Logs to view any status messages about your plugin.

Multi-threading Notes

The Messaging Gateway is a multi threaded application, with separate threads for each plugin, as well as other threads internal to the engine. This design is thus a high-performance backbone for deployments, but care must be taken in writing plugins to conform to the requirements of
multi threading.

As mentioned earlier, each database plugin runs in its own thread. Simultaneously, events are received in the plugin from the threads belonging to the engine. It is crucial that these threads do not create conditions that will result in erroneous operation or failure.

The events in the plugin have been separated below, based on the thread used to call them:

Plugin - Main Thread Events

Events called from the plugin's main thread.	`init`
	`getPollSqlStatement`
	`getPacketFromCurrentRecord`
	`getPrimaryValuesFromCurrentRecord`
	`getExemptionSqlForOnePendingRequest`

The above events are all called from the main thread, thereby, the following guarantees can be made:

- Any two or more of these above functions in a single plugin instance will never be called simultaneously, ie, an event has to return for the next event to be fired.
- The code inside a single event of a single plugin instance will execute in a single thread, ie, the same event will not be called more than once simultaneously.

Thus, the following allowance can be done on the above events:

- Code in the above events are allowed to share objects among each other without any synchronization, provided the object are not shared with the functions called from other threads (fns specified below). Note that the events above are not allowed to share objects with other plugins.

Events called from other threads

Events called from other threads.	`updateDatabaseOnSendResponse`
	`handleReceivedSMS`
	`handleReceivedStatusReport`

The events specified are called from threads other than the plugin's main thread.

Also, the events may be fired more than once simultaneously.

Also, a message may be received while the main plugin thread is processing through one of its functions.

The code will be multi threading safe if ONE of the following rules is applied:

- Code in any of the functions called by other threads should use only local variables, and not use global variables (recommended)

OR

- If an object defined globally in the plugin is to be used, it should either be:
- Multi threading safe

OR

- Synchronized at every point it is accessed, even if it is accessed at only one point in the code.

It is also highly recommended that such a shared object be created in the init event.

Note that sharing of the DBCSQLConnectionTarget in the sample plugin is safe, since it is a multi threading safe object as long as it is not altered after creation.

Also note that DBCSQLQuery objects are not multithreading-safe, so it is recommended that they be created as local variables.

Details of Other Objects

Other objects and classes that are required for plugin development are documented in the javadoc format. Navigate to the javadoc folder, and use your web browser to open the index.html file.

Additional Information and Support

Should you have any additional questions, comments or feedback on the Messaging Gateway, contact us over email at:

`support@cloudalert.cloud`

or visit our website at

`https://cloudalert.cloud`