



IBM Software Group

WebSphere Enterprise Service Bus V6.2
WebSphere Process Server V6.2
WebSphere Integration Developer V6.2

Message logger mediation primitive



@business on demand.

© 2009 IBM Corporation
Updated June 5, 2009

This presentation provides a detailed look at the message logger mediation primitive.

Goals

- Understand the message logger mediation primitive



Message logger

- ▶ Overview of function
- ▶ Use of terminals
- ▶ Definition of properties
- ▶ Custom logging option
 - Default implementation
 - Providing a custom logging implementation
- ▶ Database option
 - Message log database table
 - Retrieval and usage of log data
- ▶ Error handling
- ▶ Example usage

The goal of this presentation is to provide you with a full understanding of the message logger mediation primitive.

The presentation assumes that you are already familiar with the material presented in the presentations that cover common elements of all mediation primitives, such as properties, terminals, wiring and the use of promoted properties. The general knowledge of mediation primitives they provide is needed to understand the message logger primitive specific material in this presentation.

In this presentation, an overview of the message logger is presented along with information about the primitive's use of terminals and its properties.

The message logger has two different options for logging. The custom logging option is covered first, with a description of the default implementation and then information on how to provide your own custom logging implementation. The database option is covered next, which makes use of a message log database table. It is described, along with some configuration options available to you. A discussion of how you can retrieve and make use of the log data from the database is presented.

The presentation concludes with some error handling information and an example use of a message logger primitive.

Overview of function

- Logs selected content of the message
 - ▶ Message is written in XML format
 - ▶ All or part of the service message object (SMO) can be written
 - Configured using XPath to identify what portion of the message to write
 - Default is the message payload (/body)
- Two implementation options
 - ▶ Log to a relational database
 - ▶ Custom implementation
 - User written logging implementation based on Java™ logging APIs
 - Default implementation provided that logs to a file
- The SMO is not updated



The purpose of a message logger primitive is to log selected content of the service message object, or SMO. The message is written in XML format. You configure the primitive using an XPath expression so that all or part of the SMO is written. The default is to log the message payload, as identified by the XPath expression /body.

The message logger primitive provides you with the choice between two different implementations. One implementation writes log records to a relational database. The other implementation option, which was introduced in version 6.2, is custom logging. It makes use of the Java logging APIs. With this option, you can choose a default implementation provided by the product that writes log records to a file, or you can provide your own Java logging implementation.

The SMO is not updated by the message logger.

Overview of function

- Relational database option
 - ▶ Default database location is the common database used by the server
 - ▶ A pre-configured data source points to the common database
 - ▶ Message logger defaults to JNDI name of pre-configured data source
 - ▶ Configuration options allow use of another database, multiple databases or multiple tables within a database
- Custom logging implementation
 - ▶ Implementation based on Java logging APIs (java.util.logging)
 - ▶ Custom logging provided by implementation of these classes
 - Handler
 - Formatter
 - Filter

The relational database option logs messages to a database. The common database, used by a WebSphere® Process Server or WebSphere Enterprise Service Bus, is the default location for the database table used by the message logger. There is a pre-configured data source in the server runtime environment that identifies the common database containing the message log table. The JNDI name for this data source is used as the default JNDI name when a new message logger primitive is created. There are various configuration capabilities that allow for the use of other databases, multiple databases or multiple tables within a database.

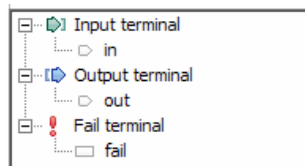
The custom logging implementation is based on the Java logging APIs, found in the package java.util.logging. The classes that are used from these APIs are the Handler, Formatter and Filter.

Terminals

- Terminals:
 - ▶ Input terminal
 - ▶ One output terminal
 - ▶ Fail terminal
- All terminals must be for the same message type



MessageLogger



The message logger primitive has one input terminal, one output terminal and a fail terminal. The output terminal must be for the same message type as the input terminal, because the message logger primitive does not modify the message body. Shown here is a message logger primitive with its terminals and the terminals as seen in the properties view.

Properties

Build Activities | Properties | Problems | Server Logs | Servers

Message Logger : MessageLogger

Description

Terminal Enabled

Details

Promotable Properties

Root: Edit...

Transaction mode: Same

Logging type: Database

Data source name: jdbc/mediation/messageLog

Handler: com.ibm.ws.sbx.mediation.primitives.logger.WESBFileHandler

Formatter: com.ibm.ws.sbx.mediation.primitives.logger.WESBFormatter

Filter: com.ibm.ws.sbx.mediation.primitives.logger.WESBFilter

Literal: {0},{1},{2},{3},{4},{5}

Level: Info

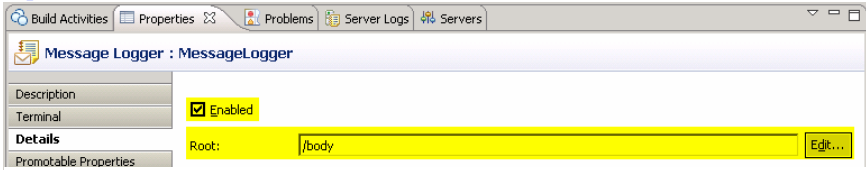
- Logging type
 - ▶ Selects either database or custom logging option
 - ▶ Only some of the properties apply to each option
 - ▶ Properties which are not applicable to the selected logging type:
 - Are ignored by the runtime
 - Are not disabled on the panel, so it is difficult to tell which properties apply



A screen capture of the Details panel of the Properties view is shown here. Highlighted in the middle of the screen is the Logging type property. This is how you choose between the database option and the custom logging option. Each of the remaining properties applies to either both options, or just to the database option or just to the custom logging option. Unfortunately, the panel does not provide any visual clues as to which properties are associated with which option and allows you to edit all of them. At runtime, the runtime engine ignores the properties which are not relevant to the selected option.

IBM Software Group IBM

Properties



- Properties common to both database and custom
 - ▶ Enabled
 - Determines if the message should be logged
 - Typical usage is to promote property so logging can be toggled on and off
 - ▶ Root
 - XPath expression defining portion of SMO to log
 - XPath expression can identify any element or portion of the SMO
 - `/body` is the default (the message payload)

7

Message logger mediation primitive © 2009 IBM Corporation

The properties shown on this slide are those that are common to both the database and custom implementation options.

The Enabled property is a toggle which tells the primitive whether it should actually perform the logging. The typical usage of this property is to promote it, thus enabling logging to be turned on and off administratively. This allows mediation flows to contain logging primitives that are disabled most of the time, but can be administratively enabled when needed for problem determination or other reasons.

The Root property contains an XPath expression that identifies the portion of the SMO that is included in the log message. When creating a new message logger, this property is set to a default value of `/body` indicating the message payload should be logged. The Edit... button opens the XPath Expression Builder dialog which can be used to drill down and identify any portion or element within the SMO that you want to have logged.

Properties

Build Activities | Properties | Problems | Server Logs | Servers

Message Logger : MessageLogger

Description

Terminal Enabled

Details

Promotable Properties

Root: /body [Edit...]

Transaction mode: Same

Logging type: Database

Data source name: jdbc/mediation/messageLog

- Properties only for the database option
 - ▶ Data source name
 - The JNDI name of a data source identifying the database
 - [jdbc/mediation/messageLog](#) is the default value
 - Pre-configured data source uses this JNDI name
 - ▶ Transaction mode:
 - [same](#) – commit database update within the flow's transaction (default)
 - [new](#) – commit database update immediately using a new transaction

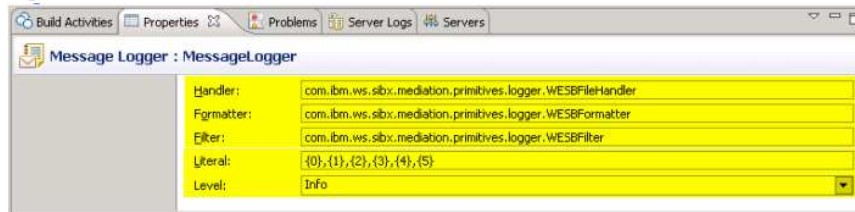


The properties shown on this slide are those that only apply to the database option.

The Data source name property is a JNDI name used to lookup the data source that identifies the database containing the table in which the messages are logged. When creating a new message logger, this property is set to a default value of `jdbc/mediation/messageLog`, which also happens to be the JNDI name for the pre-configured data source identifying the common database.

The Transaction mode property determines when the update to the message log database is committed. The default value is 'same', which means that the update is committed as part of the transaction configured for the mediation flow. The value of `new` indicates the update is committed immediately by using a new transaction.

Properties



- Properties only for the custom option
 - ▶ Handler, Formatter and Filter
 - Implementation classes as defined by Java logging APIs (java.util.logging)
 - Default implementations are provided
 - Handler is required, Formatter and Filter are optional
 - ▶ Literal
 - The message to log, with {n} representing substitution parameters
 - ▶ Level
 - The severity level assigned to this message, as defined by Java logging APIs



The properties shown on this slide are those that only apply to the custom option.

The Handler, Formatter and Filter properties each identify the concrete Java class to be used for the Handler, Formatter and Filter as defined by the Java logging APIs. These APIs are found in the java.util.logging package. There are default implementations provided which are part of the package com.ibm.ws.sibx.mediation.primitives.logger and have the class names WESBFileHandler, WESBFormatter and WESBFilter. When customizing the properties for these classes, the Handler class is required but the Formatter and Filter classes are optional.

The Literal property is the format of the message to be logged, with the numbers in brackets representing substitution parameters.

The Level property defines the severity level assigned to the message. How the level designation affects the behavior of logging is described by the Java logging API documentation.

IBM Software Group IBM

Promotable properties

Property	Promoted	Group	Alias	Alias value	Description
Enabled	<input type="checkbox"/>				
Root	<input type="checkbox"/>				
Transaction mode	<input type="checkbox"/>				
Logging type	<input type="checkbox"/>				
Level	<input type="checkbox"/>				

- Promotable
 - ▶ Enabled
 - ▶ Root
 - ▶ Transaction mode
 - ▶ Logging type
 - ▶ Level
- Not Promotable
 - ▶ Data source name
 - ▶ Handler
 - ▶ Formatter
 - ▶ Filter
 - ▶ Literal

10

Message logger mediation primitive © 2009 IBM Corporation

This slide shows the Promotable Properties panel for the message logger.

The Enabled property is promotable. As described earlier, this allows logging to be toggled on and off administratively.

The Root property is promotable, enabling you to modify what portion of the SMO is written to the log.

Promoting the Transaction mode property can be useful in a case where the mediation flow is failing downstream from the message logger primitive. You can administratively change the transaction mode to new to ensure the log is written even though the mediation itself is failing.

The Logging type property enables you to switch between the database option and custom option. If doing this, you need to ensure the configuration for the message logger is complete with the properties needed by each option specified.

The level option is also promotable.

The remaining properties are not promotable. The Data source name property contains the JNDI name of the database that is used for logging. The property is not promotable because the JNDI name gets associated with a resource reference in a generated EJB. Therefore, this requires a redeployment of the mediation application if it were to change.

Most of the properties for configuring the custom implementation are not promotable, specifically the Handler, Formatter, Filter and Literal properties.

Custom logging usage of Literal property

- Literal property defines the message format
 - ▶ {0} through {5} are substitution parameters representing:
 - {0} – Time stamp – indicates when the message was logged
 - {1} – Message ID – the message ID from the SMO
 - {2} – Mediation name – the name of the message logger primitive that logged the message
 - {3} – Module name – the name of the mediation module containing the message logger
 - {4} – Message – the message defined by the root property
 - {5} – Version – the version of the SMO
 - Example
 - ▶ Literal: MsgID={1} written at {0} from module {3}
 - ▶ Result:

MsgID=C9D315CF-0120-4000-E000-346009034B8E written at 4/21/09 12:57 PM from module CustomMsgLogTest

The next few slides look at some specifics of the custom logging implementation, starting with this slide that describes the Literal property. This property is used to define the message to be written. The message will contain exactly what is defined in the literal, with the exception of six substitution parameters, bracket enclosed numbers from zero through five. Substitution parameter zero is for the timestamp that indicates when the message was logged. Parameter one designates the message ID from the service message object. Parameter two is for the name of the message logger primitive which wrote the log and parameter three is the name of the mediation module containing the message logger. Parameter four represents the message to be logged as defined by the root property. Finally, parameter five defines the SMO version used by the mediation flow that wrote the log.

In the example there is a literal parameter which contains the message to be written, which contains three substitution parameters. A log message resulting from this is shown in small text at the bottom of the slide.

Custom logging default implementation

- The default Handler implementation:
 - ▶ Calls the Filter implementation to determine if the message should be logged
 - ▶ Calls the Formatter implementation to format the message
 - ▶ Writes log records to a file
 - The file is named MessageLog.log
 - The file is located in the system temporary directory
 - Defined in Java by `System.getProperty("java.io.tmpdir");`
 - For example
 - `C:\Documents and Settings\Administrator\Local Settings\Temp`
 - `/var/tmp` or `/tmp`

12

Message logger mediation primitive

© 2009 IBM Corporation

The custom logging default implementation of the Handler class is described here, which is the class `WESBFileHandler`. The implementation first calls the Filter implementation, if configured, to see if the message should be logged. If so, it calls the Formatter implementation, if configured, to format the message. The Handler then writes the formatted log record to a file named `MessageLog.log` located in the system temporary directory. This directory, in Java, is defined by the system property `java.io.tmpdir`. On a Windows® system, this is typically something like

`c:\Documents and Settings\Administrator\Local Settings\Temp`

and on a UNIX® system is typically either

`/var/tmp` or `/tmp`.

Custom logging default implementation

- The default Filter implementation:
 - ▶ Always indicates the message should be logged
- The default Formatter implementation
 - ▶ Formats the insertion parameters
 - Message parameter – insertion parameter {4}
 - If parameter is a DataObject, then it is serialized into XML
 - Remaining parameters are not modified
 - ▶ Inserts substitution parameters into the literal message
 - Uses the `java.text.MessageFormat` class



The default Filter implementation is very simple, always returning true to indicate that the message should be logged.

The default Formatter implementation formats the insertion parameters. Actually, most of the insertion parameters are not changed by the Formatter. Only the message parameter, insertion parameter four, is modified. If it is a DataObject it is converted to a serialized XML string, otherwise it is also left unchanged. The Formatter then uses the `java.text.MessageFormat` class to insert the substitution parameters into the literal string and returns the resulting string.

Custom logging implementation tips

- Understand the Java logging APIs
 - <http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/overview.html>
 - <http://java.sun.com/j2se/1.4.2/docs/api/java/util/logging/package-summary.html>
- Implementing the Handler class
 - ▶ Responsible for exporting the log messages
 - ▶ Extend the `java.util.logging.Handler` abstract class
 - ▶ Implement the abstract method `publish`
 - Receives the log record as a `java.util.LogRecord`
 - Calls `Filter` to determine if record should be logged
 - Calls `Formatter` to format the message
 - Writes the formatted log record to the appropriate output, for example:
 - To a file
 - To a system logging message queue



Now that you understand what the default custom logging classes do, the next couple of slides look at how you can develop your own custom logging classes. You can do this if the behavior of the default implementation does not meet your requirements. First, it is important for you to understand the Java logging APIs, the framework that is the basis for your implementation. The two URLs on this slide point you to an overview of the logging framework and to the javadoc for the interfaces and classes.

The Handler class is responsible for exporting the log messages to whatever medium they are to be sent to. Your handler should extend the abstract class `java.util.logging.Handler` and implement the abstract method `publish`. The `publish` method receives a log record defined by the `java.util.LogRecord` class. Your method needs to call the `Filter` and `Formatter` to determine if the log should be written and to format the log appropriately. You handler then writes the log to the appropriate media, such as to a file or putting the log on a message queue.

Custom logging implementation tips

- **Implementing the Formatter class**
 - ▶ Responsible for formatting the log record for output
 - ▶ Extend the `java.util.logging.Formatter` abstract class
 - ▶ Implement the abstract method `format`
 - Passed the log record as a `java.util.LogRecord`
 - Use the method `getMessage()` to obtain the literal string for the message
 - Use the method `getParameters()` to obtain the six insertion parameters
 - Format and return the message as a string
- **Implementing the Filter class**
 - ▶ Responsible for determining if the log record should be logged
 - ▶ Implement the `java.util.logging.Filter` interface
 - ▶ Implement the method `isLoggable`
 - Passed the log record as a `java.util.LogRecord`
 - Returns a boolean indicating if the record should be logged



The Formatter class is responsible for formatting the log message. Your implementation should extend the abstract class `java.util.logging.Formatter` and implement the method `format` which is passed a `java.util.LogRecord`. The `LogRecord` supports methods `getMessage` to obtain the literal string and `getParameters` to get the substitution parameters. Your code should then format the message as you require and return it as a string.

The Filter class is responsible for indicating if a particular message should actually be logged, indicating this by returning a boolean from the method `isLoggable`. Your implementation needs to inherit the interface `java.util.logging.Filter` and implement the `isLoggable` method. It is passed a `java.util.LogRecord` which you can interrogate to make your determination about logging.

Message log database table

- **Default message log database**
 - ▶ **Common database used by server**
 - Identified by data source with JNDI name jdbc/WPSDB
 - For message logger, identified by data source jdbc/mediation/messageLog
 - ▶ **Schema qualifier and table = ESBLOG.MSGLOG**
- **Options for message log database table:**
 - ▶ **Create ESBLOG.MSGLOG table in a different database**
 - JNDI name used by message logger used to identify the database
 - ▶ **Use different schema qualifier in the same database**
 - For example, MYLOG
 - Resulting in table MYLOG.MSGLOG
 - Set environment variable: ESB_MESSAGE_LOGGER_QUALIFER=MYLOG

The next few slides look at some specifics of the database option, starting with this slide that describes the message log database table. As indicated earlier, the message log table, by default, is contained in the common database used by a WebSphere Process Server or WebSphere Enterprise Service Bus. The server runtime environment provides a variety of options for the type of database used for the common database. In any case, whatever database is used, it is identified by a data source with a JNDI name of jdbc/WPSDB. The default data source used by the message logger identifies the same database and has the JNDI name of jdbc/mediation/messageLog. The message logger primitive writes into the database using the table named MSGLOG with a schema qualifier of ESBLOG.

There are alternatives to the use of the ESBLOG.MSGLOG table in the common database. One approach is to use a different database that contains an ESBLOG.MSGLOG table. When configured this way, it is the use of the JNDI data source configured for the message logger that identifies the database to use. Another approach is to use the common database but write to a MSGLOG table that has a different schema qualifier, for example, MYLOG. When doing this, it is the use of the environment variable ESB_MESSAGE_LOGGER_QUALIFER that identifies the schema qualifier to use. Of course, the two approaches can be combined, using a different database in addition to different schema qualifiers.

Your own message log database table

- Supported databases:
 - ▶ Cloudscape, DB2®, Derby, Informix®, Oracle, Sybase, Microsoft® SQL Server®
 - ▶ Data definition language (DDL) files are provided:
 - <install_root>/util/EsbLoggerMediation/<database_type>/Table.ddl
- Script for creating resources is provided:
 - <install_root>/bin/createMessageLoggerResource.jacl
 - ▶ Can be used to create
 - Database table
 - Data source
 - Schema qualifier



You can configure the message logger primitive to use a database other than the common database. The data definition language (DDL) needed to create the message logger schema and table is provided. There are separate Table.ddl files for each of the supported databases, which include Cloudscape, DB2, Derby, Informix, Oracle, Sybase and Microsoft SQL Server. The DDL files are located in the server directory structure as indicated in the slide.

To help with the configuration of your environment, the script createMessageLoggerResource.jacl is provided in the bin directory. It can be used to create a database table, data source or schema qualifier.

Your own message log database table

- Options for using your own message log database
 - ▶ Option one:
 - Delete the pre-configured data source
 - Create a data source with the default JNDI name
 - Use the default name in the message logger primitives
 - ▶ Option two:
 - Create a data source with a unique JNDI name
 - Configure the message logger primitives to use the unique name
- Using multiple message log tables
 - ▶ Multiple databases and multiple data sources
 - Uses option two above
 - ▶ Single database with different schema qualifiers
 - Required approach for z/OS® and i5/OS®



When using your own database, it is best to have a strategy on how you plan to configure your data source and your message loggers. The first option is to delete the pre-configured data source and create a new data source for your database that uses the default JNDI name. This approach allows you to continue to use the default JNDI name for each of your message logger primitives. The second option is to create a new data source with its own unique JNDI name and configure your message loggers to use the new JNDI name. The first approach makes the configuration of your message loggers easier and less prone to error because the JNDI name property does not need to be changed from its default value.

You might also want to have multiple tables used for logging. In this case, there are two approaches. The first is to follow option two, creating multiple databases and data sources, and then configuring each individual message logger to use an appropriate data source. The second approach is to use only one database containing multiple schema qualifiers, with a message log table contained with each schema qualifier. When this is done, the environment variable `ESB_MESSAGE_LOGGER_QUALIFER` must be set to indicate the appropriate schema qualifier to use. This approach is required when running on the z/OS and i5/OS platforms which provide the capability for only a single database.

Retrieval of log data

- Possible message log data usage:
 - ▶ Audit trails of enterprise service bus message handling
 - ▶ Data mining of business data contained in messages
 - ▶ Statistics of service usage
- No tools or applications are provided
 - ▶ Users must create their own tools
- Message log table schema:

Column	Type	Key	Description
TimeStamp	TIMESTAMP	Y	UTC timestamp of when message was logged
MessageID	VARCHAR	Y	Message ID from the Service Message Object
MediationName	VARCHAR	Y	Mediation primitive that logged the message
ModuleName	VARCHAR	N	Mediation module containing mediation primitive
Message	CLOB	N	Requested portion of Service Message Object in XML
Version	VARCHAR	N	The Service Message Object version

There are many possible uses of the log data contained in the message log database. For example, the log might be used to maintain an audit trail of the message handling within the enterprise service bus. Another possibility is to do some data mining of business data that is contained in the messages. A third possibility might be to compute statistics about service usage through the bus. Although there are these and many other possible uses of the log data, there are no tools provided to extract or analyze the data contained in the log. You must provide your own applications for extracting and analyzing the data based on your own requirements. The table in the slide shows the schema for the message log database. There is a timestamp containing the time the message was logged, a unique message ID, and the name of the message logger primitive that wrote the log, which together form the key. Additional fields include the name of the mediation module, the message content in XML format as defined by the Root property, and finally the SMO version associated with this log message. You need to understand this schema in order to develop an application to retrieve and analyze the log message data.

Error processing

- **MediationRuntimeException** thrown for:
 - ▶ Root XPath value of “null”
 - ▶ Incorrect JNDI name for data source
 - ▶ Custom logging Handler, Formatter or Filter class not found
- **MediationConfigurationException** (Fail terminal flow)
 - ▶ Problems accessing database
- **Failure of flow downstream of message logger primitive**
 - ▶ Transaction mode = new – message is logged
 - ▶ Transaction mode = same - message is rolled back if the mediation flow has been configured to run in a global transaction
- **Root XPath value not found in Service Message Object**
 - ▶ Not considered an error condition
 - ▶ The log is written with the Message field empty

20

Message logger mediation primitive

© 2009 IBM Corporation

The error processing details and considerations are examined in this slide.

A **MediationRuntimeException** is thrown for the case where the root property has been specified as a null XPath value. When using the database option, an incorrect JNDI name for the data source causes this exception. When using the custom implementation option, this exception occurs if the class specified for the Handler, Formatter or Filter can not be found.

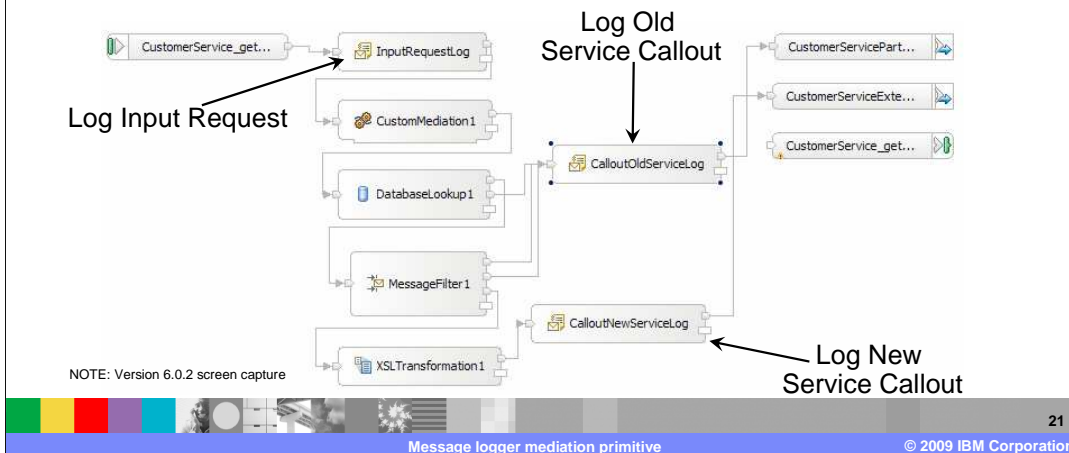
A **MediationConfigurationException** occurs for any kind of problems accessing the message log database, such as not being able to establish a connection. If the Fail terminal is wired, that flow is followed rather than the exception being thrown.

When using the database option, if there is a failure in the mediation flow downstream from the message logger, the message normally remains logged in the database. However, if the transaction mode has been set to same and the flow has been configured to run in a global transaction, the message is removed from the log by the global transaction rollback.

It is not considered an error condition when the root property contains an XPath expression that is not found in the Service Message Object. The log message is still written but has an empty message field.

Example usage

- Example – Use log to enable statistics of service usage
 - ▶ Mediation routes requests to an old or new service
 - ▶ Want to be able to track usage of the old versus new service
 - ▶ Log the input request and log the callouts to the old and new services



This slide illustrates a possible use of the message logger primitive used with the database option. The screen capture is taken from the mediation flow editor using WebSphere Integration Developer version 6.0.2. You might notice some differences in the visual appearance in later versions, but the flow being described is the same.

In this example, the requirement is to enable the keeping of statistics about service usage as requests flow through the enterprise service bus. The scenario involves a flow where the requestor uses an interface that is for the original service provider but there is now also a new service provider with a new interface. Based on some criteria involving the values in the message body a decision is made to use the old or the new provider. To meet this requirement, appropriate log messages are written so that statistics can be computed from the log database regarding usage of the old and new services. Looking at the flow diagram, you can see that there is a message logger at the beginning of the flow that records every request. There is also a message logger before the callout node to each of the service providers, so for any given request flow, there are two messages logged. Not shown in this screen capture is the message logger in the response flow which logs every response as it goes back to the requestor. Given this set of logs, it is possible to write an application that computes service usage statistics for the old and new versions of the service.

Example usage

- Example log data
 - Input request, service callout (old or new) and response all logged

Message IDs on request flow the same
Response flow has its own unique message ID

Second request/response
First request/response

Callout to old service
Callout to new service

P	TIMESTAMP	MESSAGEID	MEDIATIONNAME	MODULENAME	MESSAGE
30	2006-01-12 13:15:46	d9800bc0-0801-0000-0080-9f2d60343154	InputRequestLog	CustomerRoutingMediation	<?xml version="1.0
31	2006-01-12 13:15:47	d9800bc0-0801-0000-0080-9f2d60343154	CalloutOldServiceLog	CustomerRoutingMediation	<?xml version="1.0
32	2006-01-12 13:15:47	d9870bc0-0801-0000-0080-9f2d60343154	ResponseLog	CustomerRoutingMediation	<?xml version="1.0
33	2006-01-12 13:15:54	56a00bc0-0801-0000-0080-9f2d60343154	InputRequestLog	CustomerRoutingMediation	<?xml version="1.0
34	2006-01-12 13:15:54	56a00bc0-0801-0000-0080-9f2d60343154	CalloutNewServiceLog	CustomerRoutingMediation	<?xml version="1.0
35	2006-01-12 13:15:54	3da10bc0-0801-0000-0080-9f2d60343154	ResponseLog	CustomerRoutingMediation	<?xml version="1.0

Message IDs on request flow the same even
when message type changes during the flow

Message logger mediation primitive

© 2009 IBM Corporation

This screen capture shows the contents of a message log produced from the example on the previous slide. It shows two service requests, the first of which used the old service provider and the second of which used the new service provider. There are several things you can take note of in this screen capture. First, note that the columns for the database include the timestamp, message id, mediation name, module name and the message. There is also an SMO version column which does not show in this screen capture.

There are three logs for each request, which represent the incoming message, the callout to the old or new service provider and the response from the provider. In the mediation name column you can see that the first request went to the old service and the second request went to the new service.

The message ids are also interesting to examine. On the first request, the first and second logs, both of which are on the request flow, have the same message id, whereas the third log for the response flow has a different id. From this you see that the request and response have unique message ids. On the second request, the one that uses the new service, the SMO body was changed during the request flow by an XSLT primitive to match the new service interface. Notice that the message ids are still the same, showing that the unique message ID is associated with an SMO throughout a flow even if the structure of the SMO body is modified by a primitive.

Summary

- Examined the message logger mediation primitive



Message logger

- ▶ Overview of function
- ▶ Use of terminals
- ▶ Definition of properties
- ▶ Details of the custom logging option
- ▶ Details of the database option
- ▶ Error handling
- ▶ Example usage

23

Message logger mediation primitive

© 2009 IBM Corporation

In summary, this presentation provided details regarding the message logger mediation primitive. It presented an overview of the message logger, along with information about the primitive's use of terminals and its properties. Details of the custom logging option were presented, including an explanation of the default implementation and considerations for providing your own implementation. Details of the database option were then covered, explaining that the message logger makes use of a message log database table. A description of some configuration options available to you was presented followed by a discussion of how you can retrieve and make use of the log data from the database. Some error handling information was described that applies to both the database and custom options. Finally, an example use of a message logger primitive was presented.

Feedback

Your feedback is valuable

You can help improve the quality of IBM Education Assistant content to better meet your needs by providing feedback.

- Did you find this module useful?
- Did it help you solve a problem or answer a question?
- Do you have suggestions for improvements?

Click to send e-mail feedback:

mailto:iea@us.ibm.com?subject=Feedback_about_WBPMv62_MessageLoggerPrimitive.ppt

This module is also available in PDF format at: [../WBPMv62_MessageLoggerPrimitive.pdf](http://www.ibm.com/developerWorks/education/library/wbpmv62/MessageLoggerPrimitive.pdf)



You can help improve the quality of IBM Education Assistant content by providing feedback.

Trademarks, copyrights, and disclaimers

IBM, the IBM logo, ibm.com, and the following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

DB2 i5/OS Informix WebSphere z/OS

If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of other IBM trademarks is available on the Web at "Copyright and trademark information" at <http://www.ibm.com/legal/copytrade.shtml>

Microsoft, SQL Server, Windows, and the Windows logo are registered trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

EJB, Java, and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Product data has been reviewed for accuracy as of the date of initial publication. Product data is subject to change without notice. This document could include technical inaccuracies or typographical errors. IBM may make improvements or changes in the products or programs described herein at any time without notice. Any statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. References in this document to IBM products, programs, or services does not imply that IBM intends to make such products, programs or services available in all countries in which IBM operates or does business. Any reference to an IBM Program Product in this document is not intended to state or imply that only that program product may be used. Any functionally equivalent program, that does not infringe IBM's intellectual property rights, may be used instead.

THE INFORMATION PROVIDED IN THIS DOCUMENT IS DISTRIBUTED "AS IS" WITHOUT ANY WARRANTY, EITHER EXPRESS OR IMPLIED. IBM EXPRESSLY DISCLAIMS ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT. IBM shall have no responsibility to update this information. IBM products are warranted, if at all, according to the terms and conditions of the agreements (for example, IBM Customer Agreement, Statement of Limited Warranty, International Program License Agreement, etc.) under which they are provided. Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products in connection with this publication and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products.

IBM makes no representations or warranties, express or implied, regarding non-IBM products and services.

The provision of the information contained herein is not intended to, and does not, grant any right or license under any IBM patents or copyrights. Inquiries regarding patent or copyright licenses should be made, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.

© Copyright International Business Machines Corporation 2009. All rights reserved.

Note to U.S. Government Users - Documentation related to restricted rights-Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract and IBM Corp.