File handling in WebSphere Message Broker V6.1

Kathryn McMullan (kathryn_mcmullan@uk.ibm.com), Staff Software Engineer, WebSphere Message Broker Test Team, IBM
Brian Stewart (bri@uk.ibm.com), Staff Software Engineer, WebSphere Message Broker Test Team, IBM
Ben Thompson (bthomps@uk.ibm.com), Senior IT Specialist, IBM

Summary: This article describes the new file handling capabilities in WebSphere Message Broker V6.1, including the new FileInput and FileOutput nodes, and stream parsing.

Tags for this article: file, handling, wmb

Date: 18 Jun 2008
Level: Intermediate
Also available in: Chinese

Activity: 12307 views
Comments: 0 (View | Add comment - Sign in)

Introduction

This article shows you how to use the new file handling capabilities in IBM® WebSphere® Message Broker V6.1 (hereafter called Message Broker). It is intended to help Message Broker developers and architects use the FileInput and FileOutput nodes to implement common file handling scenarios. You should have some experience with Message Broker, though the article should be useful to both beginning and experienced users.

WebSphere Message Broker V6.1 is the first release to provide file handling capability as part of the core product. The FileInput and FileOutput nodes should be your first choice when implementing file interfaces with Message Broker for both new and existing message flows.

In the past, message flows may have used Message Broker File Extender, WebSphere Transformation Extender, Support Pac IA0X, or other plug-in node solutions to provide file handling.

Message Broker File Extender is based on technology from the SPAZIO family of products from Primeur, and runs on AIX®, Linux®, Sun Solaris®, and Windows®. The File Extender nodes are not available on z/OS®.

WebSphere Transformation Extender for Message Broker is based on the former Mercator / Ascential multi-input, multi-output transform engine. In the past, Message Broker customers have used Transformation Extender to handle very large files.

Message Broker V6.1 is an advanced Enterprise Service Bus built for universal connectivity, routing, and transformation of data in heterogeneous IT environments. Its support for file inputs and file outputs on both distributed and z/OS platforms make it an ideal choice when integrating with file based applications. The new native file support provides:

- Large file handling
- Batch processing
- Stream parsing

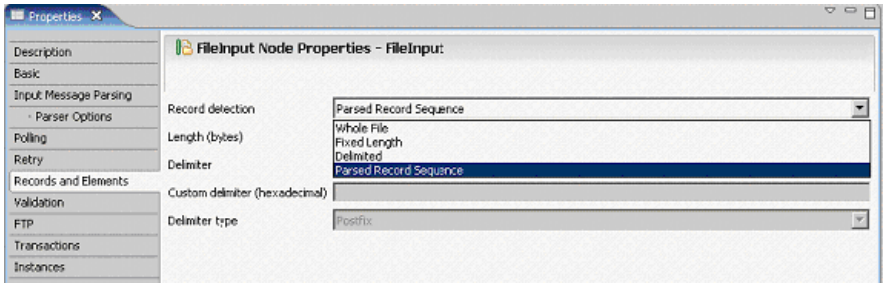
The design of the new nodes makes migration from other solutions a simple task:

- Existing message set models can be used by the FileInput node for parsing individual records and for record detection (identifying where one record finishes and the next one starts).
- The Local Environment tree is used for storing and propagating input file metadata.
- The FileInput node provides the same Validation, Transactions and Instances tabs as other Message Broker V6.1 nodes.

FileInput Node record detection

The FileInput node offers you an array of methods to parse content from an input file. When deciding which option to use, the most important consideration is how much data from the file should be sent through the message flow with the propagation of each record. The FileInput node's Records and Elements tab specifies how an input file should be interpreted with regard to its constituent records. Set properties on this tab and on the Input Message Parsing tab to control the number of messages that are sourced from the input file and propagated through the flow:

Figure 1. FileInput node properties specify the record detection technique



The following figures summarise the different models that can be invoked . The red bars indicate the part of the file propagated to the rest of the flow. The green boxes symbolise mark-up provided by delimiters:

Figure 2. Whole file -- Entire file is propagated using one propagation.

Table of contents

Figure 3. Fixed length -- Each propagation is split from the next by the FileInput node counting its length. No delimiters are required to separate each propagation from the next.

Figure 4. Delimited -- Each propagation is split from the next by the FileInput node identifying delimiter characters. The input file contains a delimiter between each propagation. The delimiters themselves are not propagated through the message flow.

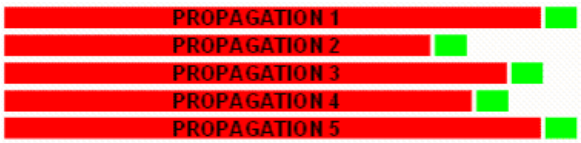
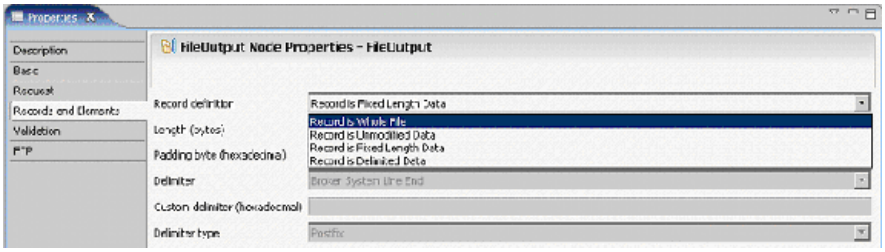


Figure 5. Parsed record sequence -- No delimiters are required to separate each propagation from the next, as long as the parser can determine where one propagation finishes and the next starts. Each record can be a different length length if used in conjunction with XMLNSC or MRM TDS options.

FileOutput node record definition

The FileOutput node gives you an array of methods to specify how output file content should be constructed from the individual records. The Records and Elements tab uses Record definition to specify the output file's construction:

Figure 6. FileOutput node properties specify the record definition technique



The following diagrams summarise the different models you can invoke. The red bars indicate the part of the file propagated from the main part of the flow to the FileOutput node. The green boxes symbolise mark-up provided by the delimiter specified on the FileOutput node's properties. The blue boxes symbolise mark-up provided by the padding bytes specified on the FileOutput node's properties.

Figure 7. Record is Whole File. The entire file content is propagated using one propagation.

Figure 8. Record is Unmodified Data. No delimiters are written between each propagation.

Figure 9. Record is Fixed Length Data. Each propagation is split from the next by the FileOutput node padding its length to a

required length using the padding bytes. No delimiters are used in the output file.

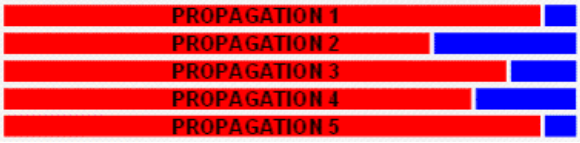
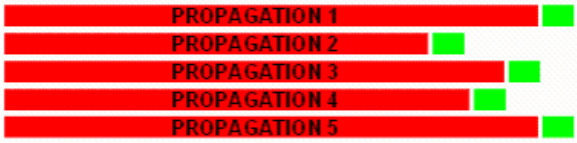


Figure 10. Record is Delimited Data. Each propagation is split from the next by the FileOutput node adding delimiter characters between them.



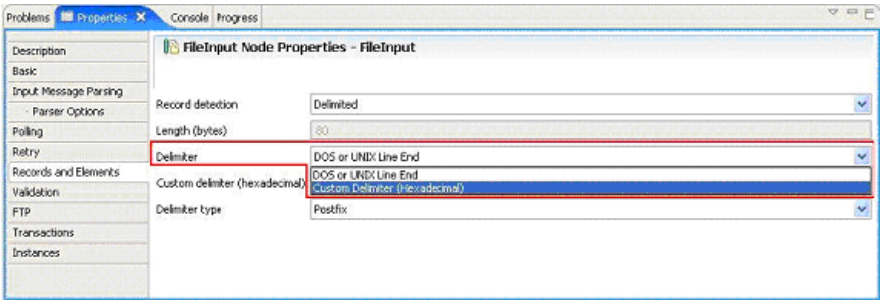
Delimited record detection using the FileInput node

On the Records and Elements tab of the FileInput node, selecting Delimited tells the node that the separation of data in the input file into separate records for propagation should be done using a delimiter. The delimiter is never included in the propagated message.

Configuring the delimiter

The delimiter can be set to either DOS or UNIX Line End, or Custom Delimiter (Hexadecimal):

Figure 11. The FileInput node properties specify the delimiter



When the delimiter is set to DOS or Unix Line End, the node recognises a line end as the delimiter. On Windows this is a carriage return character followed by a line feed character (<CR><LF> or X'0D0A'). On Unix this is a single line feed character (<LF> or X'0A'). The node treats both of these sequences as delimiters irrespective of the platform on which the broker is being run. If both appear in the same file then the node treats both as delimiters.

The node does not recognise a line end as found in EBCDIC files on z/OS, which is the newline byte (X'15'). To recognise this line end as a delimiter, you can set the node to have a custom delimiter with a value of 15.

When the delimiter is set to Custom Delimiter (Hexadecimal), the node uses a byte or sequence of bytes specified in the Custom Delimiter property as the delimiter. A valid custom delimiter must have an even number of hexadecimal digits. For example, you could enter 0D0A, which consists of four hexadecimal digits (two bytes) and represents X'0D0A'. The maximum sequence length is 32 hexadecimal digits (16 bytes). If no custom delimiter is specified, then the default of X'0A' is used, which represents a single line feed.

Configuring the delimiter type

The FileInput node considers each occurrence of the delimiter in the input file as either separating (infix) or terminating (postfix) each record. If the file begins with a delimiter, the node treats the (zero length) file contents preceding that delimiter as a record and propagates an empty record to the flow.

The default delimiter type is Postfix. This means that each delimiter terminates a record. If the file ends with a delimiter, no empty record is propagated after the delimiter. If the file does not end with a delimiter, the file is processed as if a delimiter followed the final bytes of the file.

Alternatively, a delimiter type of Infix can be specified. This means that each delimiter separates the records. If the file ends with a delimiter the (zero length) file content following the final delimiter is still propagated although it contains no data.

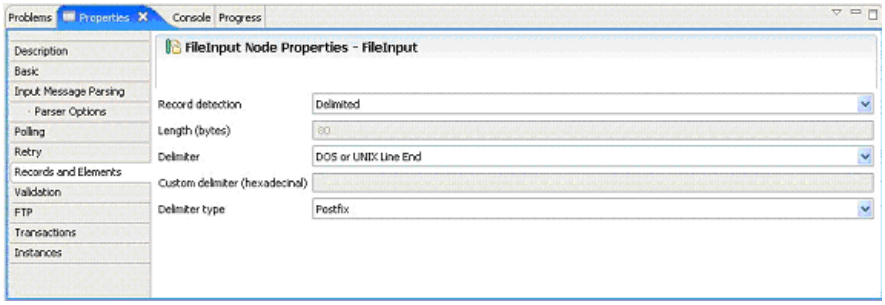
Example of reading a file where records are separated by a DOS or Unix Line End

Assume the input file in this example has the following content. Each line ends with a line terminator (on Windows this is X'0D0A', on Unix this is X'0A').

```
<AccNo>12345</AccNo>
<AccNo>34567</AccNo>
<AccNo>56789</AccNo>
```

The following properties are selected on the Records and Elements tab of the FileInput node:

Figure 12. FileInput node properties



The FileInput node detects records that end with a DOS or UNIX line end and propagates a message for each one that it finds. The DOS or UNIX line end is not part of any of the messages. The result is the propagation of three messages, as follows:

Message 1: `<AccNo>12345</AccNo>`

Message 2: `<AccNo>34567</AccNo>`

Message 3: `<AccNo>56789</AccNo>`

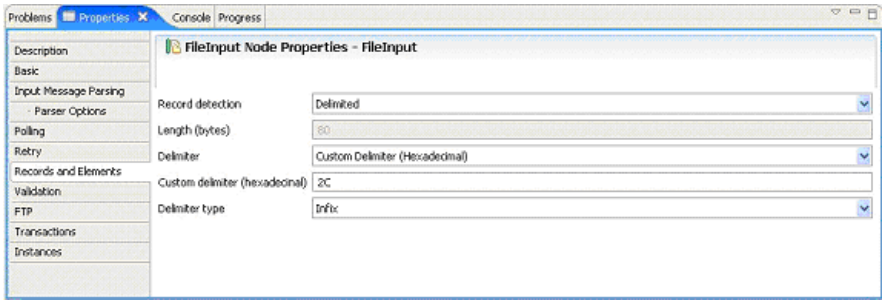
Example of reading a file where records are separated by a custom delimiter

Assume the input file in this example has the following content. There is no line terminator at the end of the file.

```
<AccNo>12345</AccNo>,<AccNo>34567</AccNo>,<AccNo>56789</AccNo>
```

The following properties are selected on the Records and Elements tab of the FileInput node:

Figure 13. FileInput node properties



The hexadecimal X'2C' represents a comma character in ASCII. On other systems, a different hexadecimal code might need to be used (this sequence does not undergo code page conversion).

The FileInput node detects the comma character and separates records with it. Because the value of the Delimiter type property is Infix, there does not need to be a comma at the end of the file. The comma character is not part of any propagated message. The result is the propagation of three messages, as follows:

Message 1: `<AccNo>12345</AccNo>`

Message 2: `<AccNo>34567</AccNo>`

Message 3: `<AccNo>56789</AccNo>`

There are no commas in the bodies of the messages in this example. If a comma had occurred within the message body, it would have caused a split between records at that point, resulting in incorrectly formed messages being propagated to the rest of the flow.

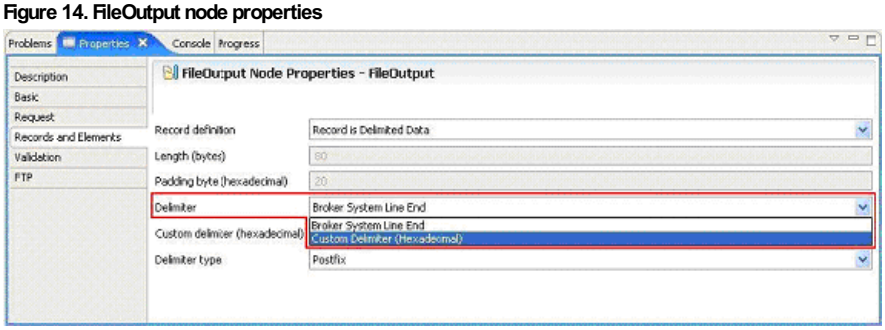
Delimited record definition using the FileOutput node

On the Records and Elements tab of the FileOutput node, selecting Record is Delimited Data tells the node that the separation of data derived from the input message should be done using a delimiter. The file is finished only when a message is received on the Finish File terminal.

Configuring the delimiter

The delimiter can be set to either Broker System Line End, or Custom Delimiter (Hexadecimal).

Figure 14. FileOutput node properties



When the delimiter is set to Broker System Line End, the node uses a line end sequence of bytes as the delimiter as appropriate for the platform on which the broker is being run. On Windows, this is a carriage return character followed by a line feed character

(<CR><LF> or X'0D0A'). On Unix this is a single line feed character (<LF> or X'0A'). On z/OS this is a 'newline' character (X'15').

When the delimiter is set to Custom Delimiter (Hexadecimal), the node uses the explicit delimiter sequence specified in the Custom delimiter property as the delimiter to be used. A valid custom delimiter must have an even number of hexadecimal digits, and the maximum sequence length allowed is 16 bytes (32 hexadecimal digits). The default is X'0A' which represents a single line feed.

Configuring the delimiter type

The default delimiter type is Postfix. This means the delimiter is added after each record that is written.

Alternatively, a delimiter type of Infix can be specified. This means the delimiter is only inserted between any two adjacent records.

Example of writing a file where records are separated by a DOS or Unix line end

Assume the following three input messages are sent to the In terminal of the FileOutput node, followed by a final message sent to the Finish File terminal of the same node. It does not matter what the final message contains.

Message 1: `<AccNo>12345</AccNo>`

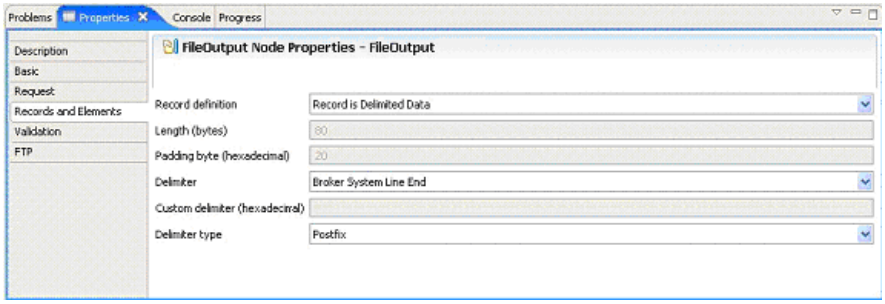
Message 2: `<AccNo>34567</AccNo>`

Message 3: `<AccNo>56789</AccNo>`

Final message: `<Anything></Anything>`

The following properties are selected on the Records and Elements tab of the FileOutput node:

Figure 15. FileOutput node properties



The FileOutput node writes one file. The file contains three records each terminated by a line terminator as appropriate for the platform on which the broker is being run. It has the following content:

```
<AccNo>12345</AccNo>
<AccNo>34567</AccNo>
<AccNo>56789</AccNo>
```

Example of writing a file where records are separated by a custom delimiter

Assume the following three input messages are sent to the In terminal of the FileOutput node, followed by a final message sent to the Finish File terminal of the same node. It does not matter what the final message contains.

Message 1: `<AccNo>12345</AccNo>`

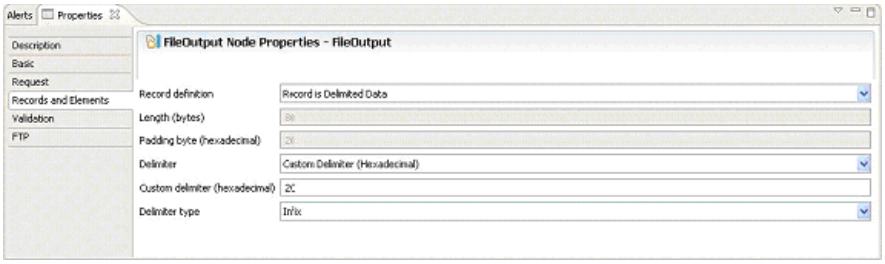
Message 2: `<AccNo>34567</AccNo>`

Message 3: `<AccNo>56789</AccNo>`

Final message: `<Anything></Anything>`

The following properties are selected on the Records and Elements tab of the FileOutput node:

Figure 16. FileOutput node properties



The hexadecimal digits X'2C' represent a comma character in ASCII. On other systems, a different hexadecimal sequence might need to be used (this sequence does not undergo code page conversion)

The FileOutput node writes one file which contains a single line. It has the following content.

```
<AccNo>12345</AccNo>,<AccNo>34567</AccNo>,<AccNo>56789</AccNo>
```

Note that the use of Infix as the delimiter type means that there is no comma added after the last record is written, so the delimiter is only inserted between any two adjacent records.

Transferring large files

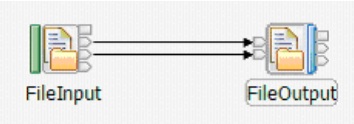
The file nodes can be used to transfer files from one directory location to another. If FTP is enabled, the files can be transferred to or from a directory on a remote FTP server. This is covered in section **Polling an FTP Server for FileInput node content**

Transferring large files using the File nodes can be achieved by splitting the data into smaller sized chunks and transferring the data 'as is' chunk by chunk.

For example, consider an arbitrary file of size 139,456,234 KB. This file could be transferred in 20,000,000 sized chunks by configuring the node properties as follows:

1. On the `Input Message Parsing` tab of the FileInput node set the Message domain to BLOB as the data is simply going to be transferred through the flow without being parsed.
2. On the `Records and Elements` tab of the FileInput node set `Record Detection` to `Fixed Length`. On this tab also set `Length (bytes)` to the number of bytes you're going to transfer in each chunk. So in this example set this to `20000000`.
3. Connect the `End of Data` terminal of the FileInput node to the `Finish File` terminal of the FileOutput node. This will cause the output file to be closed once the last chunk is transferred.
4. On the `Records and Elements` tab of the FileOutput Node set `Record Definition` to `Record isUnmodified Data`.

Figure 17. Message flow to transfer large files



In this example, the incoming file size of 139,456,234 KB does not split evenly into 20000000 byte chunks. This means each record will be 20000000 bytes except for the last record which will be 3183616 bytes long.

Padding output records to a fixed length

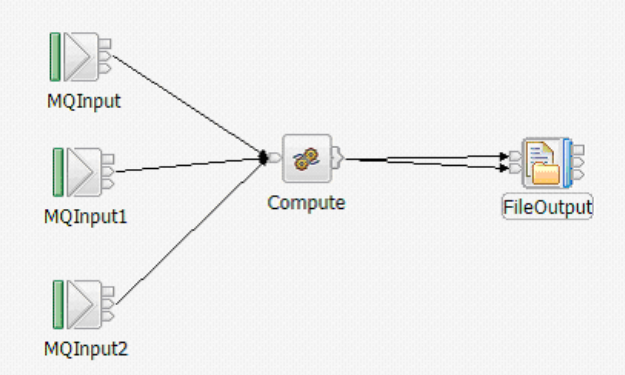
In this example the input file contains a number of records which are of different lengths. These records are written to a single output file. The application which will process the output file expects each record to be 120 bytes long. The `Padding byte` field on the `Records and Elements` tab of the FileOutput node can be used to pad the variable length records to a fixed length. In this example, by setting the following values on the `Records and Elements` tab, all output records will be padded to 120 bytes in length with the hexadecimal value 20 (the space character on ASCII platforms):

1. On the Records and Elements tab set the Record Definition to Record is Fixed Length Data
2. Set the Length (bytes) to the value `120`
3. Set the Padding byte (hexadecimal) to the value `20`

Transforming MQ messages to a file

In this example three MQInput nodes accept XML messages which are all written to a single output file. Each MQ message is appended to the same output file. The following flow can be used to achieve this:

Figure 18. Example message-to-file message flow



The three MQInput nodes (which have the `Message Domain` property set to `XMLNSC`) propagate messages to a Compute node. Because the FileOutput node is receiving input from three sources, a message needs to be sent to the `Finish File` terminal to close the file. It doesn't matter what the content of this message is, as long as the `Finish File` terminal receives a message. The file is then closed and moved from the `mqsitransit` directory to the output directory.

In this example, ESQL is added to the Compute node to determine when to send a message to the `Finish File` terminal. When all MQInput nodes have sent their last message the last of these is propagated to the `Finish File` terminal causing the file to close. The Compute node's `Out1` terminal is connected to the `Finish File` terminal of the FileOutput node.

```
DECLARE LAST_MESSAGE SHARED INTEGER 0;
CREATE COMPUTE MODULE InputFlow_Compute
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
SET OutputRoot = InputRoot;
IF InputRoot.XMLNSC.Dept.LastMessage = 'YES' THEN
BEGIN ATOMIC
SET LAST_MESSAGE = LAST_MESSAGE + 1;
END;
END IF;
IF LAST_MESSAGE = 3
THEN
PROPAGATE TO TERMINAL 'out' DELETE NONE;
PROPAGATE TO TERMINAL 'out1' DELETE NONE;
SET LAST_MESSAGE = 0;
ELSE
PROPAGATE TO TERMINAL 'out' DELETE NONE;
```

```
END IF;
RETURN FALSE;
END;
```

On the FileOutput node switch to the `Records and Elements` tab and set `Record Definition` to `Record is unmodified data`. On the `Basic` tab set the `Directory` and `File Name or pattern` to the location and name of the output file.

Transforming file records to MQ messages

This example shows how records read in from a file by a FileInput node are propagated through a flow and then output as MQ messages. The input records, which contain xml data, are read in from the input file and parsed by the XMLNSC parser. This is an extract from the input file:

```
<Dept><DeptId>124511</DeptId><Sale>34.99</Sale><ItemCode>19449539</ItemCode></Dept>
<Dept><DeptId>124512</DeptId><Sale>29.99</Sale><ItemCode>24873586</ItemCode></Dept>
<Dept><DeptId>124513</DeptId><Sale>44.99</Sale><ItemCode>54852744</ItemCode></Dept>
<Dept><DeptId>124514</DeptId><Sale>99.99</Sale><ItemCode>85222843</ItemCode></Dept>
<Dept><DeptId>124511</DeptId><Sale>34.99</Sale><ItemCode>19449539</ItemCode></Dept>
```

By using the `Parsed Record Sequence` property in the FileInput node each closing Root tag on the input data is used to mark the end of a record. Each record is then propagated through the flow as an individual message in the form:

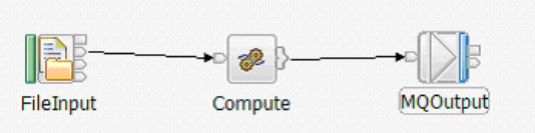
```
<Dept><DeptId>124511</DeptId><Sale>34.99</Sale><ItemCode>19449539</ItemCode></Dept>
```

The following properties are set in the FileInput node to use `Parsed Record Sequence` record detection with the XMLNSC parser:

- On the `Input Message Parsing` tab set `Message domain` to `XMLNSC`
- On the `Records and Elements` tab set `Record detection` to `Parsed Record Sequence`

You could directly connect the `Out` terminal of the FileInput node to the `In` terminal of the MQOutput node. Any transformation node could be added between these nodes to enable the modification of each file record. The FileInput node's `End of Data` terminal is not connected in this example as the output node is not a FileOutput node.

Figure 19. Example file-to-message message flow

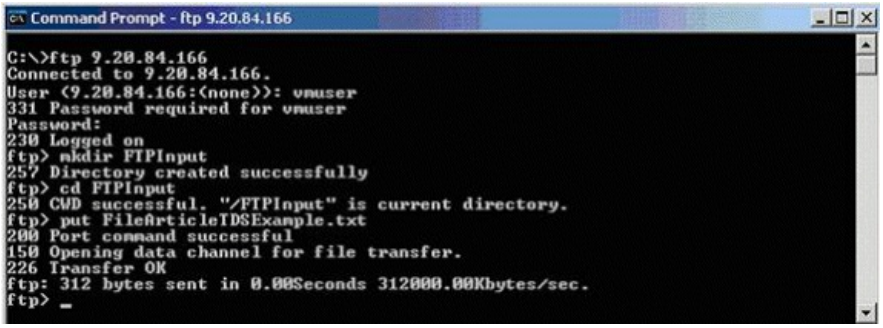


Each record is output from the MQOutput node as a separate message.

Polling an FTP server for FileInput node content:

The FileInput nodes can also be used to execute a poll of a remote FTP server in order to retrieve input files. First, check that the FTP server is running and that you have sufficient authority using a standard interface outside of the broker. In the screen shot below, a command line access to an FTP server is shown. After logging on, a directory was created and an input file for a message flow was copied.

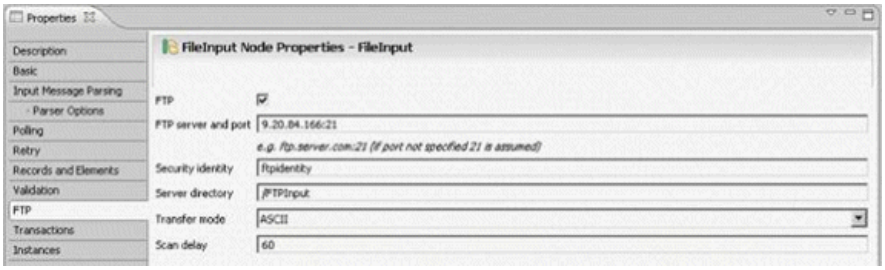
Figure 20. Command-line FTP instructions



The FileInput node's FTP tab is used to configure the properties the broker needs in order to retrieve files when polling the server. Further abstraction in setting these properties is also possible through the definition of a configurable service, using the broker command mqsicreateconfigurableservice. More information on this topic is available from the Information Center

Consider a FileInput node with its properties set as shown in the screen shot below:

Figure 21. FileInput node properties

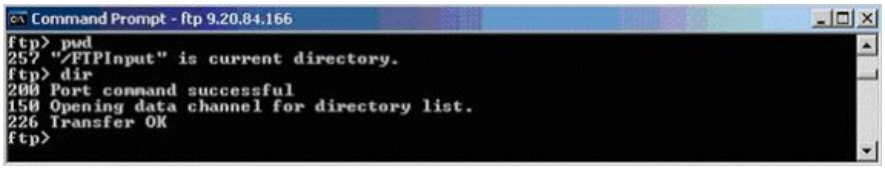


The Security identity property causes the broker to locate the username and password pair from its own security store. The mqsisetdbparms command is used to configure these settings. Make sure that the runtime broker is stopped before issuing the command:

Figure 22. Setting a security identity for the broker

Start the broker and deploy a message flow which uses a FileInput node configured as previously specified. You can use one of the message flows supplied with this article as an example. Check that the message flow performs as if the input file had been placed in the broker's local file system. Make sure that the input file is removed from the FTP server (as shown in the command window below) and that the output file is created as configured in the message flow's FileOutput node.

Figure 23. Command-line FTP prompt showing file has been processed



When processing files from a remote FTP server, the file is transferred first and then the transferred copy is processed as though it had been placed in the local input directory. If the remote file is successfully transferred, then it is deleted from the FTP server. If problems occur when parsing the file's contents (after successful FTP transfer) then the archive (mqsiarchive) and backout (mqsibackout) subdirectories of the input directory are populated according to the backout policy specified on the FileInput node's Retry tab.

Using file metadata from the LocalEnvironment variables

LocalEnvironment variables containing information about the files read in and written to by a flow can be accessed, and in some cases modified. Complete list of the file-related LocalEnvironment variables.

This section gives examples of accessing and modifying some of these LocalEnvironment variables.

Using the LocalEnvironment variables to query the file record number

The `LocalEnvironment.File` fields are set by the FileInput node. In this example the date is only added to the first file record. Compute node ESQL is written to query the `LocalEnvironment.File.Record` variable, which contains the record number of the current record. The date is only added to the file record when this variable is equal to 1. The ESQL used in the Compute node is

```
IF InputLocalEnvironment.File.Record = 1
 THEN
   SET OutputRoot.XMLNSC.Dept.(XMLNSC.Attribute)Date = CURRENT_DATE;
END IF;
```

and the first record output from the flow will have the form

```
<Dept Date="2007-12-06"><DeptId>124511</DeptId>
<Sale>34.99</Sale><ItemCode>19449539</ItemCode></Dept>
```

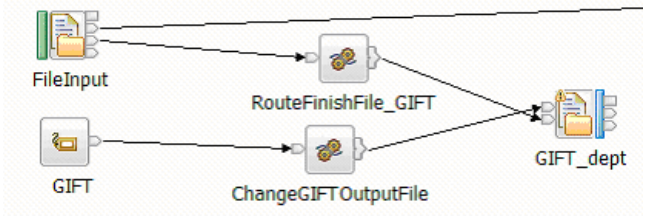
Using the LocalEnvironment to change the output file name

`LocalEnvironment.Destination.File.Directory` and `LocalEnvironment.Destination.File.Name` can be modified to override the name and directory of the output file set in the FileOutput node. For example, this ESQL will modify the file name

```
SET OutputLocalEnvironment.Destination.File.Name = 'GIFT_dept.xml';
```

If you use the `LocalEnvironment` in this way, make sure the "Finish File"(empty message sent to the Finish File terminal) propagation also has its `LocalEnvironment` set in the same way. This ensures that when the `Finish File` terminal of the FileOutput node receives it, the correct file will be closed.

Figure 24. Example message flow extract for changing an output file name



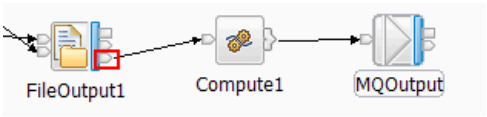
Check the `Request directory property location` and `Request file name property location` settings in the `Request` tab of the FileOutput node. These should be set to point at the `LocalEnvironment`, as shown in the figure below.

Figure 25. FileOutput node properties

Using the LocalEnvironment to query the name of an output file

The `LocalEnvironment.WrittenDestination.File` variables contain information about the file that has previously been written by the FileOutput node.

Figure 26. Example message flow extract for querying the file name used by the FileOutput node



The ESQL in the Compute node:

```
SET OutputRoot.XMLNSC.Dept.OutputFile =
   InputLocalEnvironment.WrittenDestination.File.Directory || '\' ||
InputLocalEnvironment.WrittenDestination.File.Name;
```

This uses the LocalEnvironment to find out the name of the file written by the FileOutput node. In this example, the file name is then output as an MQ message. Notice the Compute node is wired to the `End of Data` terminal. If the Out terminal had been wired, the Compute node would have received a propagation for every record.

The FileInput node and the MRM parser's delimiter support

The FileInput node has the ability to specify a custom delimiter as a sequence of hexadecimal digits. For example, to specify a line feed as the delimiter, you would enter a value of 0Ain the Custom delimiter (hexadecimal) property on the node.

The MRM Tagged/Delimited String (TDS) parser offersthree different syntaxes for specifying delimiters:

- Standard characters, such as a space character
- The mnemonic for a space character, which is <SPACE>
- The Unicode value for a space character, which is <U+0020>

More information on MRM delimiter syntaxes.

The MRM Custom Wire Format (CWF) parser offers tree different syntaxes for specifying padding characters (CWF has no concept of a delimiter):

- Standard characters in quotes such as ' ' or " " to specify a space character.
- Hexadecimal notation such as 0x20to specify a space character
- Adecimal value such as 32to specify a space character
- AUnicode value such as U+0020 to specify a space character

More information on these padding character syntaxes.

Import the resources into the Message Broker Toolkit

The Download section of this article contains a single zip file named FileExamples.zip. This is a Project Interchange file which can be imported into the WMBv6.1 Toolkit. All of the message flow and message set projects referred to by this article will be available in the imported workspace.

Reusing MRM definitions with the FileInput node

Message flow - WholeFile.msgflow

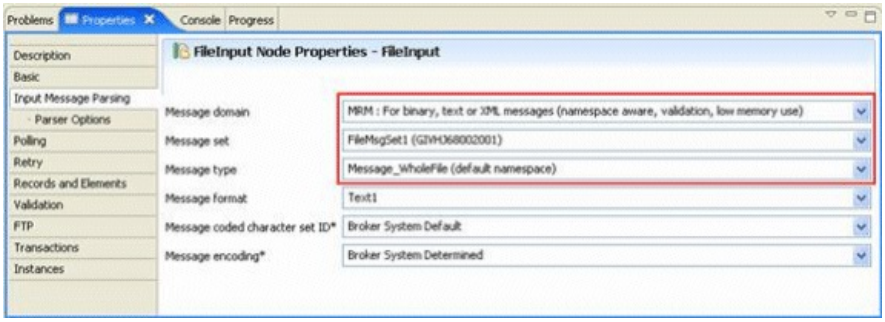
The message flow can be found in the File Msg Flows project and is shown in the figure below:

Figure 27. Message flow WholeFile.msgflow

The flow takes a file as input from one directory, and writes it to another directory. The output file is identical to the input file. A trace node is used to provide useful output of what is propagated from the FileInput node. This can be compared with the trace from the second example message flow (Delimited.msgflow).

The FileInput node has the following properties set. In the Records and Elements tab, the `Record` detection is set to Whole File. On the Input Message Parsing tab, the properties are as shown in the Figure below. Note that the MRM message type is specified as Message_WholeFile (this is defined in the message definition file MsgDefFile.mxsd). The message set project is named File`MsgSet1`.

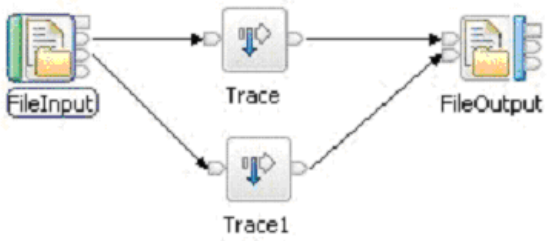
Figure 28. FileInput node properties in WholeFile.msgflow



Message flow - Delimited.msgflow

The message flow can be found in the File MsgFlows project and is shown in the figure below.

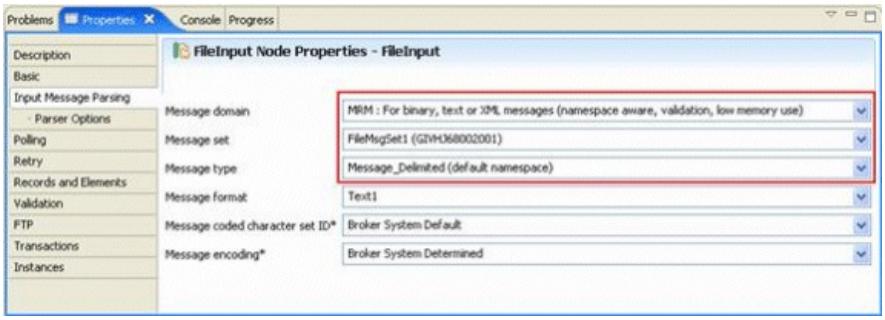
Figure 29. Message flow Delimited.msgflow



The flow takes a file as input from one directory, and writes it to another directory. The output file is identical to the input file. Trace nodes are used to provide useful output of what is propagated from the FileInput node. This can be compared with the trace from the first example message flow (`WholeFile.msgflow`).

The FileInput node has the following properties set. On the Records and Elements tab, the `R`ecord detection is set to Delimited (using DOS or UNIX Line End as the Delimiter, and Postfix as the Delimiter type). On the Input Message Parsing tab, the properties are as shown in the Figure below. Note that the MRM message type is specified as Message_Delimited (this is defined in the message definition file MsgDefFile.mxsd). The message set project is named File`MsgSet1`.

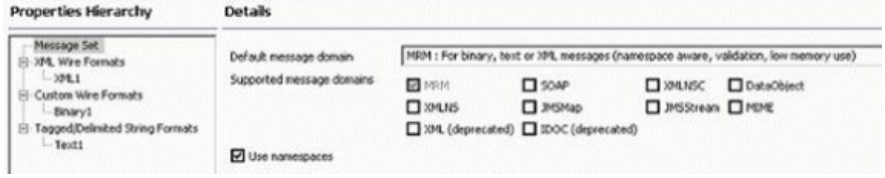
Figure 30. FileInput node properties in Delimited.msgflow



Message set - messageSet.mset

The message set can be found in the project named `FileMsgSet1`. The message set's `Default message domain` needs to be set appropriately (MRM for this example) in order for a Broker Archive File to be built correctly ready for deployment. Note that this behaviour is new with version 6.1.

Figure 31. MessageSet.mset file properties



Message definition file - MsgDefFile.mxsd

The message definition file can be found in the project named File `MsgSet1`.

Figure 32. Message definition file MsgDefFile.mxsd

Figure 32.

MsgDefFile.mxsd

Structure	Type	Min Occurs	Max Occurs
MsgDefFile.mxsd			
Messages			
Header	HeaderType		
Record	RecordType		
Trailer	TrailerType		
Message_Delimited	Message_DelimitedType		
Header	HeaderType	0	1
Department	xsd:string	1	1
NumberOfRecords	xsd:int	1	1
Total	xsd:decimal	1	1
Record	RecordType	0	1
ItemName	xsd:string	1	1
DateOfReturn	xsd:date	1	1
ReasonForReturn	xsd:string	1	1
Amount	xsd:decimal	1	1
Trailer	TrailerType	0	1
SubTotal1	xsd:decimal	1	1
SubTotal2	xsd:decimal	1	1
SubTotal3	xsd:decimal	1	1
Total	xsd:decimal	1	1
Message_WholeFile	Message_WholeFileType		
Header	HeaderType	1	1
Record	RecordType	1	-1
Trailer	TrailerType	1	1
Types			
Groups			
Elements and Attributes			

Note that `Message_Delimited` is built upon a `choice` type, where as `Message_WholeFile` is built upon a `sequence` (which includes an unbounded number of occurrences of the Record element.)

These two message definitions have purposefully been built using the same element references, in order to show reuse of existing definitions by the File nodes.

Input file - WholeFile_input.txt

The input file can be found in the project named File`MsgF`lows. It contains the following data:

```
HDR:RefundsDepartment,7,57.62
REC:widget,2007-05-13,faulty,9.99
REC:asset,2007-06-19,unwanted gift,5.50
REC:thing,2007-07-07,faulty,6.66
REC:widget,2007-08-30,exchange,9.99
REC:asset,2007-09-01,faulty,5.50
REC:widget,2007-09-30,exchange,9.99
REC:widget,2007-10-10,faulty,9.99
TRL:39.96,11.00,6.66,57.62
```

This file is used as input to the message flow`WholeFile.msgflow`.

Input file - Delimited_input.txt

The input file can be found in the project named `File MsgFlows`. It contains exactly the same data as the **WholeFile_input.txt** file.

This file is used as input to the message flow Delimited `.msgflow`.

Broker archive file - FileArchive.bar

The broker archive file can be found in the project named File`MsgF`lows. It contains compiled versions of the two message flows and a dictionary file for the message set.

Running the message flows

By default, the sample is setup for Windows and assumes the following directories have been created on the file system before the broker archive is deployed.

`C:\File Examples\FILEIN` -- The directory being read by the FileInput node of each flow

`C:\File Examples\FILEOUT` -- The directory being written to by the FileOutput node of each flow

`C:\File Examples\TRACE` -- The directory being written to by the Trace node(s) of each flow

Deploy the broker archive to the broker.

To run either flow, place a copy of the relevant input file in the directory being read by the FileInput node (by default this is C:\FileExamples\FILEIN).

An output file will be produced in directory C:\FileExamples\FILEOUT, and a trace file (or files) will be produced in directory C:\FileExamples\TRACE.

WholeFile.msgflow receives an input file named `WholeFile_input.txt`and creates an output file named `WholeFile_output.txt`. Delimited.msgflow receives an input file named `Delimited_input.txt`and creates an output file named `Delimited_output.txt`

The content of the output files will be identical to whichever input file is used to drive the flow.

Comparison of propagations between the two message flows

`WholeFile.msgflow` only involves one propagation from the FileInput node. In contrast, `Delimited .msgflow` involves many separate propagations from the FileInput node.

This is because the FileInput node in WholeFile.msgflow detects a record as being the entire input file, and therefore propagates only once. Delimited.msgflowhowever uses a FileInput node that detects a record as ending with a carriage return followed by a line feed (CRLF).

`WholeFile.msgflow` does not perform record detection in its FileInput node, instead the CRLF between each line in the input data is interpreted by the TDS parser. In the case of `Delimited.msgflow`, the CRLFs *are* being used by the FileInput node to determine where one propagation ends and the next starts.

Trace output from WholeFile.msgflow

The trace output for `WholeFile .msgflow`can be found in the file `TRACETREE_WHOLEFILE.txt`. It will contain the following content (Note that trace of the Properties folder has been removed from this snippet):

```
(0x01000021):MRM      = (
 (0x01000013):Header  = (
  (0x0300000B):Department     = 'RefundsDepartment'
  (0x0300000B):NumberOfRecords = 7
  (0x0300000B):Total          = 57.62
 )
 (0x01000013):Record  = (
  (0x0300000B):ItemName       = 'widget'
  (0x0300000B):DateOfReturn    = DATE '2007-05-13'
  (0x0300000B):ReasonForReturn = 'faulty'
  (0x0300000B):Amount         = 9.99
 )
 (0x01000013):Record  = (
  (0x0300000B):ItemName       = 'asset'
  (0x0300000B):DateOfReturn    = DATE '2007-06-19'
  (0x0300000B):ReasonForReturn = 'unwanted gift'
  (0x0300000B):Amount         = 5.50
 )
 (0x01000013):Record  = (
  (0x0300000B):ItemName       = 'thing'
  (0x0300000B):DateOfReturn    = DATE '2007-07-07'
  (0x0300000B):ReasonForReturn = 'faulty'
  (0x0300000B):Amount         = 6.66
 )
 (0x01000013):Record  = (
  (0x0300000B):ItemName       = 'widget'
  (0x0300000B):DateOfReturn    = DATE '2007-08-30'
  (0x0300000B):ReasonForReturn = 'exchange'
  (0x0300000B):Amount         = 9.99
 )
 (0x01000013):Record  = (
  (0x0300000B):ItemName       = 'asset'
  (0x0300000B):DateOfReturn    = DATE '2007-09-01'
  (0x0300000B):ReasonForReturn = 'faulty'
  (0x0300000B):Amount         = 5.50
 )
 (0x01000013):Record  = (
  (0x0300000B):ItemName       = 'widget'
  (0x0300000B):DateOfReturn    = DATE '2007-09-30'
  (0x0300000B):ReasonForReturn = 'exchange'
  (0x0300000B):Amount         = 9.99
 )
 (0x01000013):Record  = (
  (0x0300000B):ItemName       = 'widget'
  (0x0300000B):DateOfReturn    = DATE '2007-10-10'
  (0x0300000B):ReasonForReturn = 'faulty'
  (0x0300000B):Amount         = 9.99
 )
    (0x01000013):Trailer = (
   (0x0300000B):SubTotal1 = 39.96
   (0x0300000B):SubTotal2 = 11.00
   (0x0300000B):SubTotal3 = 6.66
   (0x0300000B):Total     = 57.62
  )
 )
```

Trace output from Delimited.msgflow

The trace output for `Delimited .msgflow` can be found in two files. The first file, TRACETREE_DELIMITED.txt, will contain the following content (Note that trace of the Properties folder has been removed from this snippet):

Apropagation of the Header element...

```
(0x01000021):MRM      = (
 (0x01000013):Header = (
  (0x0300000B):Department     = 'RefundsDepartment'
  (0x0300000B):NumberOfRecords = 7
  (0x0300000B):Total          = 57.62
 )
)
```

and then several propagations of the Record element...

```
(0x01000021):MRM      = (
 (0x01000013):Record = (
  (0x0300000B):ItemName       = 'widget'
  (0x0300000B):DateOfReturn    = DATE '2007-05-13'
  (0x0300000B):ReasonForReturn = 'faulty'
  (0x0300000B):Amount         = 9.99
 )
)
```

until the final propagation of the Trailer element...

```
(0x01000021):MRM       = (
  (0x01000013):Trailer = (
    (0x0300000B):SubTotal1 = 39.96
    (0x0300000B):SubTotal2 = 11.00
    (0x0300000B):SubTotal3 = 6.66
    (0x0300000B):Total     = 57.62
  )
)
```

The second trace file, TRACEFILE_EOF.txt contains the following content (Note that trace of the Properties folder has been removed from this snippet):

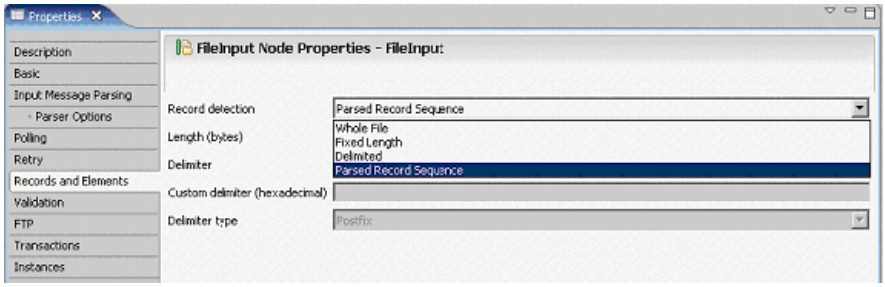
```
(0x01000000):BLOB      = (
  (0x03000000):UnknownParserName = 'NONE'
    (0x03000000):BLOB            = X''
    )
```

This is because the message sent from the FileInput's End of Data terminal is always an empty BLOB message.

Record detection of Parsed Record Sequence

On the `Records and Elements` tab, selecting `Parsed Record Sequence` tells the FileInput node that the separation of the data in the input file into records for propagation should be done using the properties on the `Input Message Parsing` tab alone. In other words, rather than rely on a record Length or Delimiter specified on the node itself, one of the broker's parsers should be used to decide where one record ends and the next begins.

Figure 33. FileInput node properties



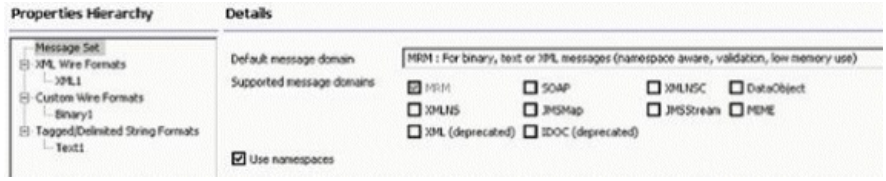
At the initial release of WebSphere Message Broker v6.1, the `Parsed Record Sequence` setting supports the use of the following parsers to carry out record detection:

- XMLNSC domain parser
- MRM TDS domain parser
- MRM CWF domain parser

If a flow developer deploys a FileInput node with Parsed Record Sequence set as `Record Detection`, with any other parser setting not in the above list (MRM XML for example) for the Input Message Parsing parameter, then at runtime an exception BIP6066E will be raised.

The message set's `Default message domain` needs to be set appropriately (MRM for this example) in order for a Broker Archive File to be built correctly ready for deployment. This behaviour is new with V6.1.

Figure 34. MessageSet.mset file properties



The MRM domain can support multiple Wire Formats (which are shown on the left of the screen under the Properties Hierarchy). In the examples which follow, a single message set is used to model one logical model with two physical formats - a `Tagged/Delimited String Format` named `Text1` and a `Custom Wire Format` named `Binary1`.

Example of Parsed Record Sequence using XMLNSC Parser:

The following example shows the content of an input file which is parsed by the FileInput node in the message flow named ParsedRecordSequence_XMLNSC.msgflow. This flow can be found in the project named FileMsgFlows. The colour coding demonstrates how the data should be separated into records for propagation through the message flow.

Figure 35. File input from ParsedRecordSequence_XMLNSC_input.txt

Figure 35

Note that there is no overall root tag for the entire file. Each record is a well-formed XML message with a single root tag named `Invoice`. The XMLNSC parser interprets the XML tags to determine where one Invoice ends and the next one begins. Each

Invoice record is propagated through the message flow separately. Tracing the flow shows four propagations as follows:

Propagation 1:

```
(0x01000000):XMLNSC    = (
  (0x01000000):Invoice = (
    (0x01000000):Customer = (
      (0x03000000):FirstName   = 'Fred'
      (0x03000000):LastName    = 'Bloggs'
      (0x03000000):AddressLine = '123 Happy Street'
      (0x03000000):City        = 'Joyville'
    )
    (0x01000000):Product  = (
      (0x03000000):Description   = 'WIDGET'
      (0x03000000):NumberOfItems = '3'
      (0x03000000):UnitPrice     = '19.99'
      (0x03000000):Total         = '59.97'
    )
  )
 )
```

Propagation 2:

```
(0x01000000):XMLNSC    = (
  (0x01000000):Invoice = (
    (0x01000000):Customer = (
      (0x03000000):FirstName   = 'John'
      (0x03000000):LastName    = 'Doe'
      (0x03000000):AddressLine = '123 Sad Avenue'
      (0x03000000):City        = 'Sadstate'
    )
    (0x01000000):Product  = (
      (0x03000000):Description   = 'THING'
      (0x03000000):NumberOfItems = '2'
      (0x03000000):UnitPrice     = '3.50'
      (0x03000000):Total         = '7.00'
    )
  )
 )
```

Propagation 3:

```
(0x01000000):XMLNSC    = (
 (0x01000000):Invoice = (
   (0x01000000):Customer = (
     (0x03000000):FirstName   = 'Gordon'
     (0x03000000):LastName    = 'Brown'
     (0x03000000):AddressLine = '10 Downing Street'
     (0x03000000):City        = 'London'
   )
   (0x01000000):Product  = (
     (0x03000000):Description   = 'ASSET'
     (0x03000000):NumberOfItems = '1'
     (0x03000000):UnitPrice     = '49.99'
     (0x03000000):Total         = '49.99'
   )
 )
)
```

Propagation 4:

```
(0x01000000):XMLNSC    = (
 (0x01000000):Invoice = (
   (0x01000000):Customer = (
     (0x03000000):FirstName   = 'George'
     (0x03000000):LastName    = 'Bush'
     (0x03000000):AddressLine = '1600 Pennsylvania Avenue'
     (0x03000000):City        = 'Washington'
   )
   (0x01000000):Product  = (
     (0x03000000):Description   = 'WIDGET'
     (0x03000000):NumberOfItems = '4'
     (0x03000000):UnitPrice     = '19.99'
     (0x03000000):Total         = '79.96'
   )
 )
)
```

Example of Parsed Record Sequence using MRM CWF Parser:

The following example shows the content of an input file which is parsed by the FileInput node in the message flow named `ParsedRecordSequence_MRMCWF.msgflow`. This flow can be found in the project named `FileMsgFlows`. The colour coding demonstrates how the data should be separated into records for propagation through the message flow.

Figure 36. File input from ParsedRecordSequence_MRMCWF_input.txt

This example demonstrates how a Record Detection of Parsed Record Sequence can be used with the MRM CWF parser, to propagate separate fixed length records through a message flow. Each record matches to the definition of the Invoice message's CWF wire format layer. The MRM CWF parser uses the record's properties (the length of each field) to determine where one Invoice ends and the next one begins. Each Invoice record is propagated through the message flow separately. Tracing the flow would show four propagations in exactly the same way as the other examples.

Example of Parsed Record Sequence using MRM TDS Parser:

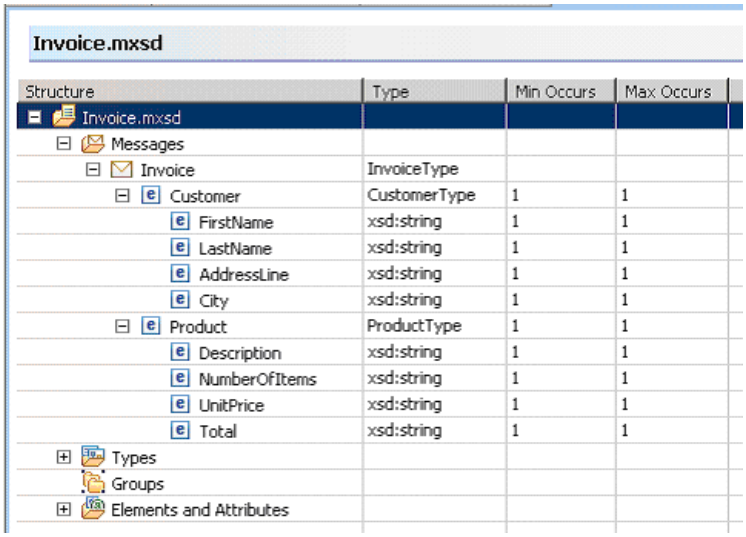
The following example shows the content of an input file which is parsed by the FileInput node in the message flow named `ParsedRecordSequence_MRMTDS.msgflow`. This flow can be found in the project named FileMsgFlows. The colour coding demonstrates how the data is separated into records for propagation through the message flow.

Figure 37. File input from ParsedRecordSequence_MRMTDS_input.txt


```
(Customer:(FirstName:Fred,LastName:Bloggs,AddressLine:123 Happy Street,City:Joyville)#Product:(Desc
ription:WIDGET,NumberOfItems:3,UnitPrice:19.99,Total:59.97))(Customer:(FirstName:John,LastName:Do
e,AddressLine:123 Sad Avenue,City:Sadstate)#Product:(Description:THING,NumberOfItems:2,UnitPrice:3.
50,Total:7.00))(Customer:(FirstName:Gordon,LastName:Brown,AddressLine:10 Downing Street,City:London
)#Product:(Description:ASSET,NumberOfItems:1,UnitPrice:49.99,Total:49.99))(Customer:(FirstName:Ge
orge,LastName:Bush,AddressLine:1600 Pennsylvania Avenue,City:Washington)#Product:(Description:WIDGE
T,NumberOfItems:4,UnitPrice:19.99,Total:79.96))
```


There is no delimiter character placed between each record. It is the definition of the MRM TDS message which determines where one propagation should finish and the next begins. The message definition which is used to parse the record is shown in the figure below. Invoice.mxsd can be found in the message set project named FileMsgSet2.

Figure 38. Message definition file Invoice.mxsd

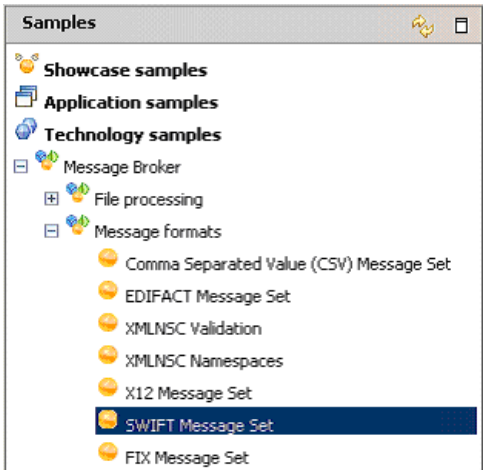


The MRM TDS properties of the message provide the parser with information about the Delimiters, Group Indicators, and Group Terminators, which allow the parser to resolve where one message finishes and the next begins.

Example of Parsed Record Sequence using SWIFT MRM TDS Parser:

From the WebSphere Message Broker Toolkit Help menu, open the Samples Gallery and import the SWIFT Message Set sample:

Figure 39. SWIFT Message Set sample



This example is concerned with the imported Message Set project, named `SWIFT Message Sets`. The imported message flow project, named `SWIFT Message Flows`, can be ignored for our purposes. The message flow used in this example is named ParsedRecordSequence_SWIFT.msgflow and can be found in the message flow project named FileMsgFlows.

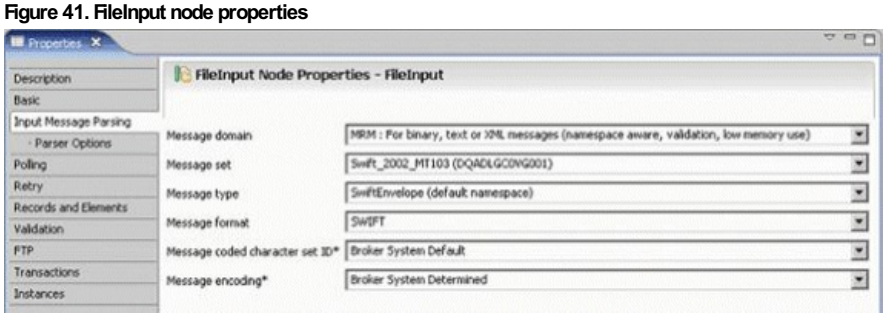
The input file named `ParsedRecordSequence_SWIFT_input.txt` contains four SWIFT messages concatenated one after another. The content of each message is taken from the enqueue message file provided with the SWIFT Sample.

Figure 40. File input from ParsedRecordSequence_SWIFT_input.txt

```
{1:F01IBMADEF0AXXX0000000000}{2:I103IBMADEF0AXXXN}{3:{108:103   0001ACK   }}{4:
:20:16x  … Further content including CRLF characters …
}{1:F01IBMADEF0AXXX0000000000}{2:I103IBMADEF0AXXXN}{3:{108:103   0001ACK   }}{4:
:20:16x  … Further content including CRLF characters …
}{1:F01IBMADEF0AXXX0000000000}{2:I103IBMADEF0AXXXN}{3:{108:103   0001ACK   }}{4:
:20:16x  … Further content including CRLF characters …
}{1:F01IBMADEF0AXXX0000000000}{2:I103IBMADEF0AXXXN}{3:{108:103   0001ACK   }}{4:
:20:16x  … Further content including CRLF characters … }
```

Setting properties on the FileInput node's Input Message Parsingtab, in combination with a Record detection of Parsed Record Sequence, causes the MRM TDS Parser to use the SWIFT Message Set sample to determine where one SWIFT record finishes, and the next begins.

Figure 41. FileInput node properties



Each record (symbolised by a different colour in the earlier figure) is propagated through the message flow, one after another. To view the individual messages being propagated through the flow, locate the example trace file, named C:\FileExamples\TRACE\TRACETREE_PARSEDRECORDSEQUENCE_SWIFT.TXT

Routing within the message flow based on file content

Using the RouteToLabel node is one method of routing file records through a message flow based on message content. In this example, the FileInput node reads an input file as shown:

```
<Dept><DeptId>124511</DeptId><Sale>34.99</Sale><ItemCode>19449539</ItemCode></Dept>
<Dept><DeptId>124512</DeptId><Sale>29.99</Sale><ItemCode>24873586</ItemCode></Dept>
<Dept><DeptId>124513</DeptId><Sale>44.99</Sale><ItemCode>54852744</ItemCode></Dept>
<Dept><DeptId>124514</DeptId><Sale>99.99</Sale><ItemCode>85222843</ItemCode></Dept>
<Dept><DeptId>124511</DeptId><Sale>34.99</Sale><ItemCode>19449539</ItemCode></Dept>
```

Each department id `<DeptId>` listed in the file corresponds to a department name in the store. The department id in each file record is used to route records to one of four labels: `ELECTRICAL`, `GIFT` ,`KITCHEN` or `HOME`, corresponding to the four department names in the store:

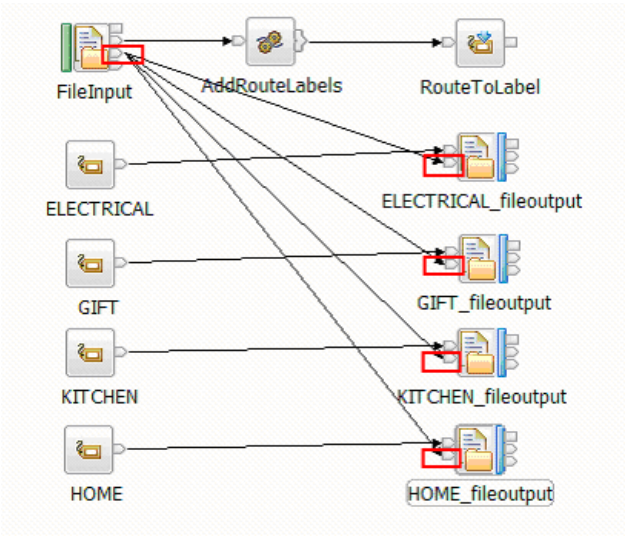
```
<Dept><DeptId>124511</DeptId><Sale>34.99</Sale><ItemCode>19449539</ItemCode></Dept>
```

ACompute node is used to set up labels for the RouteToLabel node. The extract of ESQL below shows how the labels are set up to use the `<DeptId>` in the incoming message.

```
SET OutputLocalEnvironment.Destination.RouterList.DestinationData[1].labelname =
  CASE (InputRoot.XMLNSC.Dept.DeptId )
    WHEN '124511' THEN 'ELECTRICAL'
    WHEN '124512' THEN 'KITCHEN'
    WHEN '124513' THEN 'GIFT'
    WHEN '124514' THEN 'HOME'
    ELSE 'UNKNOWN_DEPT'
  END;
```

The four label nodes in the flow are connected to four FileOutput nodes, which produce one output file for each department. Each FileOutput node needs to receive a "Finish File" message into its `Finish File` terminal in order to close the four separate output files. This is the reason for the connections shown from the FileInput node's `End of Data` terminal.

Figure 42. Routing within a message flow based on file content



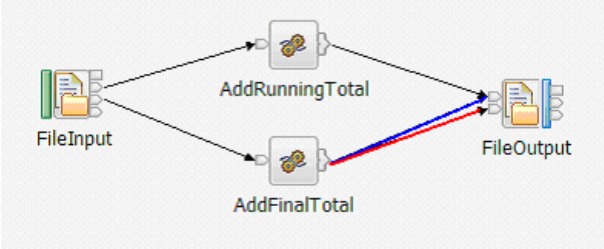
Transforming file record content and adding a new final record

In this example, the FileInput node reads an input file as shown:

```
<Dept><DeptId>124513</DeptId><Sale>34.99</Sale><ItemCode>19449539</ItemCode></Dept>
<Dept><DeptId>124511</DeptId><Sale>19.99</Sale><ItemCode>24873586</ItemCode></Dept>
<Dept><DeptId>124514</DeptId><Sale>59.99</Sale><ItemCode>54852744</ItemCode></Dept>
```

Each record has an element, `<Sale>`, which contains the cost of an item sold in the department. A Compute node, named `AddRunningTotal`, is used to keep a running total (as an ESQL shared variable named `running Total`) of all the items which have passed through the flow. The `runningTotal` is added to each record before it is written to the output file. The final total of all items is added to the output file after **all** the records from the input file have been processed. The message flow below shows how this can be achieved.

Figure 43. Message flow transforming file record content



Within the Compute node named `AddRunningTotal` the following ESQL keeps a running total, and also adds a new element to each record showing the total sales so far.

```
SET OutputRoot = InputRoot;
BEGIN ATOMIC
  SET runningTotal =
     runningTotal + CAST(InputRoot.XMLNSC.Dept.Sale  AS DECIMAL);
END;
SET OutputRoot.XMLNSC.Dept.TotalSales = runningTotal;
```

Having kept a running total of the sales we need to add a final record to the output file containing the total sales for the day. This can be done by adding a Compute node before the `Finish File` terminal on the FileOutput node.

Shown below is the ESQL in the `AddFinalTotal` Compute node, which creates a new element named `FinalSales` and assigns to it the contents of the shared variable `runningTotal`. The node will propagate this data to the `In` terminal of the FileOutput node, so that the data can be written to the output file (the blue connection from the `out1` terminal shown). The Compute node will also propagate the "Finish File" message to the `Finish File` terminal of the FileOutput node so that the output file can be closed (the red connection from the `out` terminal shown): `SET OutputRoot.XMLNSC.Dept.FinalSales = runningTotal;`

```
PROPAGATE TO TERMINAL 'out1' DELETE NONE;
SET OutputRoot = InputRoot;
SET runningTotal = 0.0;
RETURN TRUE;
```

The records written to the output file will now have the running total added. There will be one additional record at the end of the file showing the final total.

The extract below shows the records written by the FileOutput node. Each record contains the running total element `TotalSales` which is added by the `AddRunningTotal` compute node. The last record shows the final sales element `FinalSales` added by the compute node `AddFinalTotal`.

```
<Dept><DeptId>124513</DeptId><Sale>34.99</Sale><ItemCode>19449539</ItemCode>
   <TotalSales>34.99</TotalSales></Dept>
<Dept><DeptId>124511</DeptId><Sale>19.99</Sale><ItemCode>24873586</ItemCode>
   <TotalSales>54.98</TotalSales></Dept>
<Dept><DeptId>124514</DeptId><Sale>59.99</Sale><ItemCode>54852744</ItemCode>
   <TotalSales>114.97</TotalSales></Dept>
<Dept><FinalSales>114.97</FinalSales></Dept>
```

Using XPath in the FileOutput node to selectively write an element

On the `Request` tab on the FileOutput node you can enter an XPath expression in the `Data Location` field to select what data from the input record is to be written to the output file. Consider an input file,containing records of the following format:

```
<Dept><DeptId>124511</DeptId><Sale>34.99</Sale><ItemCode>19449539</ItemCode></Dept>
```

To write just the ItemCode of each record to an output file, set Data Location on the Request tab of the FileOutput node to $Body/Dept/ItemCode.

Transactionality

It is important for message flow developers to understand the behaviour of the FileInput and FileOutput nodes with respect to transactions. The FileInput node provides parameters which control the behaviour of the containing message flow, in circumstances where message flow actions are carried out successfully, and in circumstances where they fail. The FileInput node has a **Transactions** tab which is used to set the `Transaction mode` to be either Yes or No. The transaction setting on this node determines whether the rest of the nodes in the message flow should be executed under syncpoint or not. The File nodes themselves are not transactional, so this property affects other nodes in the flow, but not the File nodes. The default setting for this property is Yes. This value means that flow updates (for example, inserting in a database, and writing a message to a WMQ queue) should be treated transactionally if possible. Flow developers may also be familiar with the concept of a Message Flow property CommitCount which, for WebSphere MQ, messages specifies how many input messages are processed by a message flow before a syncpoint is taken (by issuing an MQCMIT). This property does not affect flow behaviour when a message flow is not processing WebSphere MQ messages, so should not be confused with configuring the File nodes. There is no equivalent batching property for use with the File nodes.

The interactions between a message flow containing a File node (either input node or output node) and the file system itself are not transactional. This kind of coordination would only be possible with the support of a truly transactional file system. Therefore the file writes, moves, deletes and appends which are choreographed through configuration of File nodes in message flows, are never backed out or committed in batches under the control of the flow.

Multiple Record Detection policies (Whole File ,Fixed Length, Delimited, Parsed Record Sequence) are supplied which determine how the input file's content should be handled by a message flow.

The FileInput node's **Records and Elements** tab includes a **Record detection** property, which controls how many propagations result from a single input file. Selecting **Whole File** causes the entire content of the input file to be propagated as a single logical tree through the message flow. Selecting **Fixed Length**, **Delimited**, or **Parsed Record Sequence** causes the content of the input file to be broken down into separate records and propagated individually through the message flow. In cases where the Whole File content is propagated, and the flow is terminated with a FileOutput node, once the flow completes successfully, the output file will be automatically written to the output file system. In cases where content is propagated record by record through the message flow, the output file will only be written to the output file system when the FileOutput node receives a message to its **Finish File** terminal.

The FileInput node allows flow developers to specify what actions should be carried out when a message flow fails to successfully process an input file. On the `Retry` tab, the `Action on failing file` property specifies what action should be carried out on the input file after all attempts to process its contents have failed. The file can be moved to a Backout Subdirectory (with or without a timestamp appended to the file name), or can be deleted.

The FileInput node also lets you specify what action should be carried out on the original input file when a message flow has successfully completed processing it. On the**Basic** tab, the **Action on successful processing** property specifies that the original file can be moved to an Archive Subdirectory (with or without a timestamp appended to the file name), or can be deleted.

In the case of a catastrophic failure during the processing of a file, such as a hardware failure or anything that causes an immediate stop of the Message Broker's execution group, the original input file will be left on the file system and processed again from the beginning when the broker is restarted. The broker does not record how far through processing a multi-record file it is, so there is no concept of the broker flow restarting from the position where it stopped.

Input file polling and message flow scalability

The FileInput node property named `Polling interval` controls the frequency with which the FileInput node accesses the file system looking for files to process. An initial scan of the input directory when the flow is started populates a list of input files that match the input pattern and are waiting to be processed by the message flow. The message flow processes these files according to the point at which they were last modified (oldest file first). Each file is parsed by the FileInput node using a single thread from the available pool of instances (like all message flow input nodes, the FileInput node can be configured to use a thread pool associated with the message flow or a thread pool associated with the individual node). This architecture guarantees that the records in a particular input file will be processed in the order in which they appear in the file. You can scale the broker to operate on more than one input file at the same time by increasing the number of instances available in the thread pool.

As soon as a flow instance finishes processing its input file, it returns to the thread pool and begins work on the next input file from the list. When all files from the list have been processed, the flow returns to the input directory to seek new files which have arrived since the previous scan. If no further input file is available (the directory contains no usable files that match the input pattern.), then the FileInput node waits for a period defined by the Polling interval, preventing the the FileInput node from continually accessing the file system and consuming system resources. Set the polling interval according to your application requirements. The default of 5 seconds is good for testing new flows because it reduces the time taken to repeat a test run. For a production system with many files or a networked file system, you may find that longer intervals such as 60 seconds or even 300 seconds reduce system load.

Back to top

Download

Description	Name	Size	Download method
Code sample	FileExamples.zip	124 KB	HTTP

Information about download methods

Resources

- WebSphere Message Broker developer resources page
  Technical resources to help you use WebSphere Message Broker for connectivity, universal data transformation, and enterprise-level integration of disparate services, applications, and platforms to power your SOA.

- WebSphere Message Broker product page
  Product descriptions, product news, training information, support information, and more.

- WebSphere Message Broker information center
  Asingle Eclipse-based Web portal to all WebSphere Message Broker V6 documentation, with conceptual, task, and reference information on installing, configuring, and using your WebSphere Message Broker environment.

- WebSphere Message Broker documentation library
  WebSphere Message Broker specifications and manuals.

- WebSphere Message Broker forum
  Get answers to your technical questions and share your expertise with other WebSphere Message Broker users.

- WebSphere Message Broker support page
  Asearchable database of support problems and their solutions, plus downloads, fixes, problem tracking, and more.

- Redbook: Patterns: SOADesign Using WebSphere Message Broker and WebSphere ESB
  Patterns for e-business are a group of proven, reusable assets that can be used to increase the speed of developing and deploying e-business applications. This Redbook shows you how to use WebSphere ESB together with WebSphere Message Broker to implement an ESB within an SOA. Includes scenario to demonstrate design, development, and deployment.

- WebSphere SOAsolutions developer resources page
  Get technical resources for WebSphere SOAsolutions.

- developerWorks WebSphere Business Integration zone
  For developers, access to WebSphere Business Integration how-to articles, downloads, tutorials, education, product info, and more.

- WebSphere Business Integration products page
  For both business and technical users, a handy overview of all WebSphere Business Integration products

- WebSphere forums
  Product-specific forums where you can get answers to your technical questions and share your expertise with other WebSphere users.

- Most popular WebSphere trial downloads
  No-charge trial downloads for key WebSphere products.

- developerWorks technical events and Webcasts
  Free technical sessions by IBM experts that can accelerate your learning curve and help you succeed in your most difficult software projects. Sessions range from one-hour Webcasts to half-day and full-day live sessions in cities worldwide.

About the authors

Kathryn McMullan is a Staff Software Engineer on the WebSphere Message Broker Test Team at the IBM Hursley Software Lab in the UK. She has been involved in testing WebSphere Message Broker for over six years. Kathryn has previously worked on CICS. You can contact Kathryn at kathryn_mcmullan@uk.ibm.com.

Brian Stewart is a Staff Software Engineer on the WebSphere Message Broker Test Team at the IBM Hursley Software Lab in the UK. He has more than eight years experience in software development. Since joining IBM Hursley in 1998, he has worked with the CICS and WebSphere product families on the distributed and z/OS platforms. He has worked on WebSphere Message Broker for the past three years and specialises in the Java Compute node, user-defined extensions, and Message Broker setup. You can contact Brian at bri@uk.ibm.com.

Ben Thompson is a Senior IT Specialist in IBM Software Group EMEA Laboratory Services in Hursley, UK. He has worked with distributed transactional middleware for seven years, and has extensive experience designing and implementing solutions using the WebSphere product portfolio with IBM customers worldwide. You can reach Ben at bthomps@uk.ibm.com.

Rate this article

Comments

Notify me when a comment is added

1000 characters left

Post

Be the first to add a comment

Back to top