Using the Collector node in WebSphere Message Broker V6.1

Andrew Coleman

May 15, 2008

The new WebSphere Message Broker V6.1 Collector node can collect groups of messages according to various configurable parameters, enabling new message processing scenarios in which messages can cross-reference other messages that pass through the broker at different times. Using a simple online ordering example, this article shows you how to use this new Collector node.

Introduction

IBM® WebSphere® Message Broker integrates applications by routing and transforming messages from one endpoint to another. The usual process is "one message in, one message out," and the information within each message can be filtered (some information removed), augmented (information added, such as from a database), or transformed (from one format to another). Sometimes a single input message is processed to produce several output messages, either by shredding a large message into smaller ones, or by publishing a message to a number of subscribing applications.

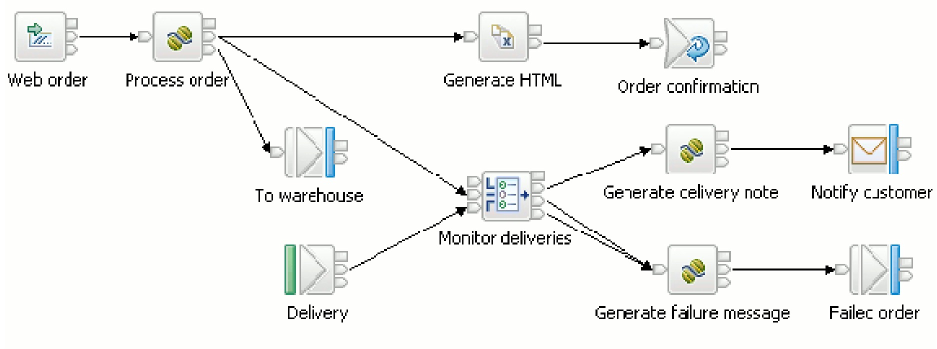
WebSphere Message Broker V6.1 now provides a generic mechanism to gather a number of input messages into a single message collection and then process that collection as a single entity. Thus, you can write mediations in which the message consumer requires information from more than one message producer. This article introduces this concept using a simple online ordering scenario, and describes how the new Collector node is used to create these message collections.

A simple event-driven application

To illustrate the concept of message collections, consider the following simplified online ordering system. Two departments manage order handling:

- **Admin** -- Handles ordering and billing using WebSphere Message Broker. This department deals with credit card data and maintains personal information, so untrusted personnel cannot access this information.
- **Warehouse** -- Takes an order, including items, quantities, and mailing address, then packages and dispatches the order. An MQ message is sent back to the admin department (DELIVERY queue) when dispatched. Personnel in this department do not see credit card information or e-mail addresses.

Figure 1. Message flow to processes Web order



The following sections describe the message flow that processes the Web order and show how the Collector node is used to coordinate the processing of messages from diverse inputs.

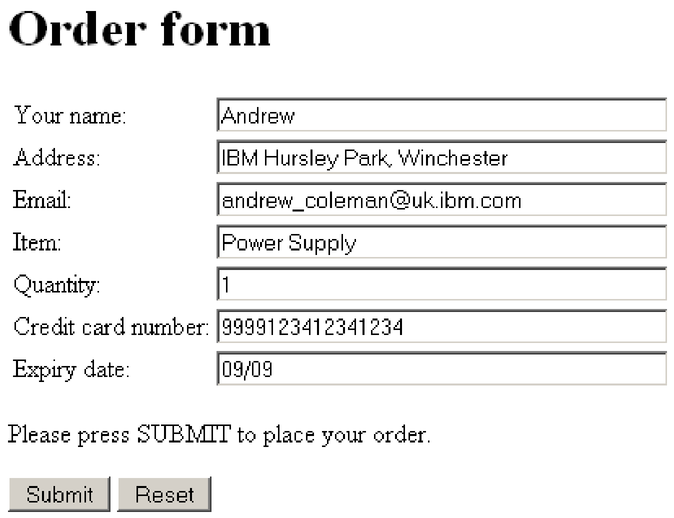
More than one input

This message flow receives input from two sources:

- From a Web page as an HTML order form
- From an MQ queue containing dispatch messages from the warehouse

The message flow receives orders from the Web page, extracts data from them, and sends the data to the warehouse. The warehouse is obliged to dispatch the order within three hours of receipt and send a dispatch message back to flow (second input). The message flow then correlates the dispatch message with the original order message in order to generate an e-mail notification to the customer. If a dispatch message is not received from the warehouseds within 3 hours of order placement, the delay is flagged as a violation of the service agreement by placing a message on a failure queue.

Figure 2. First input to the flow is an HTML order form



Order form

Your name:	Andrew
Address:	IBM Hursley Park, Winchester
Email:	andrew_coleman@uk.ibm.com
Item:	Power Supply
Quantity:	1
Credit card number:	9999123412341234
Expiry date:	09/09

Please press SUBMIT to place your order.

The message is submitted as a POST request to the HTTPInput node (Web order), which is passed into a JavaCompute node (Process order) as an application/x-www-form-urlencoded string using a BLOB parser. The string value for the input data in Figure 2 is (ignoring newlines):

```
name=Andrew&address=IBM+Hursley+Park%2C+Winchester&email=andrew_coleman%40uk.ibm.com
&item=Power+Supply&quantity=1&creditcard=9999123412341234&expiry=09%2F09&Submit=Submit
```

The following Java code shows how this message can be processed and transformed.

```
import java.util.*;
import com.ibm.broker.javacompute.MbJavaComputeNode;
import com.ibm.broker.plugin.*;

public class ProcessOrder extends MbJavaComputeNode
{
  private int orderNumber = 10000;

  public void evaluate(MbMessageAssembly inAssembly) throws MbException
  {
    MbOutputTerminal out = getOutputTerminal("out");
    MbOutputTerminal alt = getOutputTerminal("alternate");

    MbMessage inMessage = inAssembly.getMessage();

    MbElement blobElement =
            inMessage.getRootElement().getFirstElementByPath("/BLOB/BLOB");
    String request = new String((byte[])blobElement.getValue());
    // extract order information
    Map<String, String=> queryTable = new Hashtable<String, String=>();
    String fields[] = request.split("&");
    for(String field : fields)  // iterate over the array
    {
      String keyValue[] = field.split("=");  // extract name/value pairs
      queryTable.put(keyValue[0], keyValue[1]);  // populate hash table
    }

    // now extract the information we need
    String item = queryTable.get("item");             // item
    String quantity = queryTable.get("quantity");     // quantity
    String name = queryTable.get("name");             // name
    String address = queryTable.get("address");       // address
    String email = queryTable.get("email");           // email
    String creditcard = queryTable.get("creditcard"); // credit card number
    String expiry = queryTable.get("expiry");         // credit card expiry date

    orderNumber++; // increment the order number

    MbMessage processMessage = new MbMessage();
    MbMessageAssembly processAssembly =
       new MbMessageAssembly(inAssembly, processMessage);
    MbElement processRoot = processMessage.getRootElement();
    copyMessageHeaders(inMessage, processMessage);

    // copy the request identifier into the reply identifier (required for HTTPReply node)
    MbElement localEnv = inAssembly.getLocalEnvironment().getRootElement();
    MbElement requestId =
            localEnv.getFirstElementByPath("/Destination/HTTP/RequestIdentifier");
    processRoot.getFirstElementByPath("/Properties/ReplyIdentifier")
            .setValue(requestId.getValue());

    // build the 'admin' message
    MbElement body = processRoot.createElementAsLastChild(MbXMLNSC.PARSER_NAME);
    MbElement doc = body.createElementAsLastChild(MbXMLNSC.FOLDER, "order", null);
    doc.createElementAsLastChild(MbXMLNSC.FIELD, "orderID", orderNumber);
    doc.createElementAsLastChild(MbXMLNSC.FIELD, "item", item);
    doc.createElementAsLastChild(MbXMLNSC.FIELD, "quantity", quantity);
    doc.createElementAsLastChild(MbXMLNSC.FIELD, "name", name);
    doc.createElementAsLastChild(MbXMLNSC.FIELD, "address", address);
    doc.createElementAsLastChild(MbXMLNSC.FIELD, "email", email);
    doc.createElementAsLastChild(MbXMLNSC.FIELD, "creditcard", creditcard);
    doc.createElementAsLastChild(MbXMLNSC.FIELD, "expiry", expiry);

    out.propagate(processAssembly);

    // build the 'warehouse' message (restricted information)
    MbMessage warehouseMessage = new MbMessage();
    MbMessageAssembly warehouseAssembly =
            new MbMessageAssembly(inAssembly, warehouseMessage);
   MbElement warehouseRoot = warehouseMessage.getRootElement();
    // copy properties header
    warehouseRoot.addAsFirstChild(inMessage.getRootElement().getFirstChild().copy());
    body = warehouseRoot.createElementAsLastChild(MbXMLNSC.PARSER_NAME);
    doc = body.createElementAsLastChild(MbXMLNSC.FOLDER, "order", null);
    doc.createElementAsLastChild(MbXMLNSC.FIELD, "orderID", orderNumber);
    doc.createElementAsLastChild(MbXMLNSC.FIELD, "item", item);
    doc.createElementAsLastChild(MbXMLNSC.FIELD, "quantity", quantity);
    String mailingLabel = name + "\n" + address;
    doc.createElementAsLastChild(MbXMLNSC.FIELD, "mailingLabel", mailingLabel);

    alt.propagate(warehouseAssembly);
  }

  public void copyMessageHeaders(MbMessage inMessage, MbMessage outMessage)
  throws MbException
  {
    MbElement outRoot = outMessage.getRootElement();

//  iterate though the headers starting with the first child of the root element
    MbElement header = inMessage.getRootElement().getFirstChild();
    while (header != null && header.getNextSibling() != null)
            // stop before the last child (body)
    {
      // copy the header and add it to the out message
      outRoot.addAsLastChild(header.copy());
      // move along to next header
      header = header.getNextSibling();
    }
  }

}
```

The JavaCompute node splits this URLEncoded string into separate name/value pairs and populates a hash table. It then builds two messages:

- The first message is used to generate an HTML reply back to the Web browser and is also passed to the Collector node.
- The second message is sent to the warehouse and contains only the information needed to dispatch the order, which does not include credit card data or e-mail address.

First response

The Web page that sent the POST request expects a timely response, generally in HTML, which can be generated from the incoming XML message using XSLT. The following stylesheet formats a reply. It is executed by the XMLT node (Generate HTML). The output from the stylesheet is propagated to the HTTPReply node (Order confirmation) for delivery to the Web browser:

```
<?xml version=&quot;1.0&quot; encoding=&quot;UTF-8&quot;?=>
<xsl:stylesheet xmlns:xsl=&quot;http://www.w3.org/1999/XSL/Transform&quot;
    version=&quot;1.0&quot;
    xmlns:xalan=&quot;http://xml.apache.org/xslt&quot;=>
<xsl:output method=&quot;html&quot;/=>
```

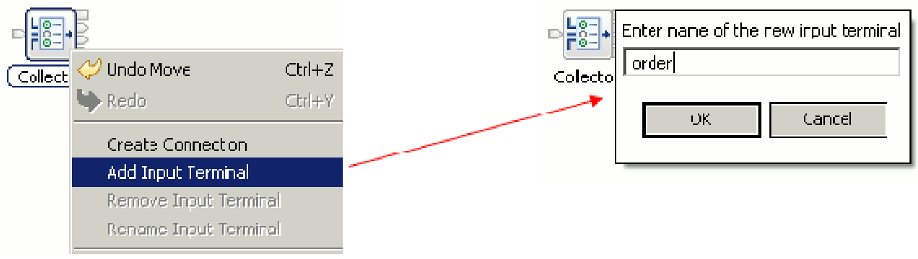
```
  <xsl:template match=&quot;/&quot;=>
    <html=>
      <head=>
        <title=>Order confirmation</title=>
      </head=>
      <body=>
        <p=>
          Dear <xsl:value-of select=&quot;/order/name&quot; />,
        </p=>
        <b=>Thank you for your order.</b=>
        <p=>
          It has been sent to our warehouse and will be
          dispatched within 3 hours. You will receive an email
          when it is dispatched.
        </p=>
        <p=>
          Order reference number:
          <xsl:value-of select=&quot;/order/orderID&quot; />
        </p=>
      </body=>
    </html=>
  </xsl:template=>

</xsl:stylesheet=>
```

Collecting and correlating the order and dispatch messages

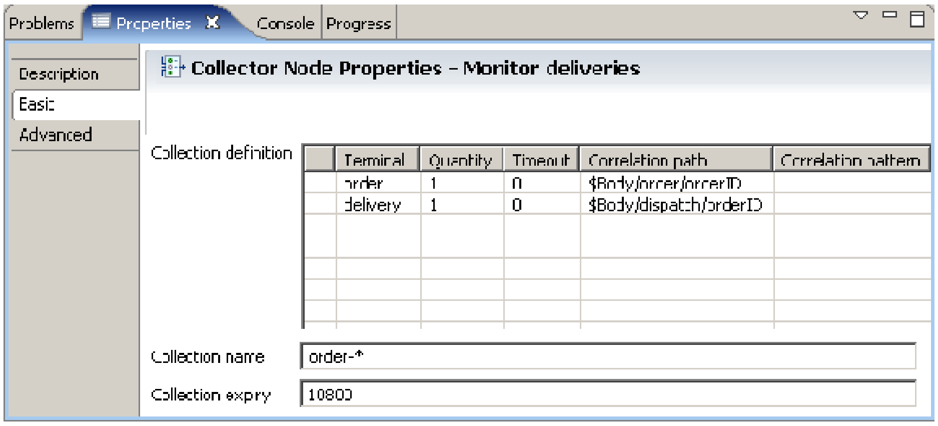
When the warehouse generates a dispatch message, an e-mail must be sent to the customer. The customer's e-mail address was part of the original order message that passed through the message flow earlier, but was not included in the MQ message sent to the warehouse. Such situations are common in event processing systems, in which an event is meaningful only within the context of other events. In this case, the dispatch message has to be processed while still having access to the original order message. Here is where the Collector node helps: the node is configured with two dynamic input terminals, one to receive the original order message and the other to receive the dispatch message from the warehouse, as shown in Figure 3:

Figure 3. Creating dynamic input terminals on a Collector node by right-clicking on the node



The node properties define how these messages are grouped together. Each input terminal has a quantity property set to 1 (see Figure 4 for details). Setting quantity means that a collection message will not be propagated until that quantity has been reached. In this case, it is a quantity of one message of each type. It is also necessary to correlate the incoming messages according to a common orderID. The Correlation path property on each terminal is set to the location in the respective messages that contain the orderID field. The expression to determine the location is expressed using XPath 1.0 syntax:

Figure 4. Collector node properties: One line in the table is created for each input terminal.



Collector Node Properties - Monitor deliveries

Collection definition

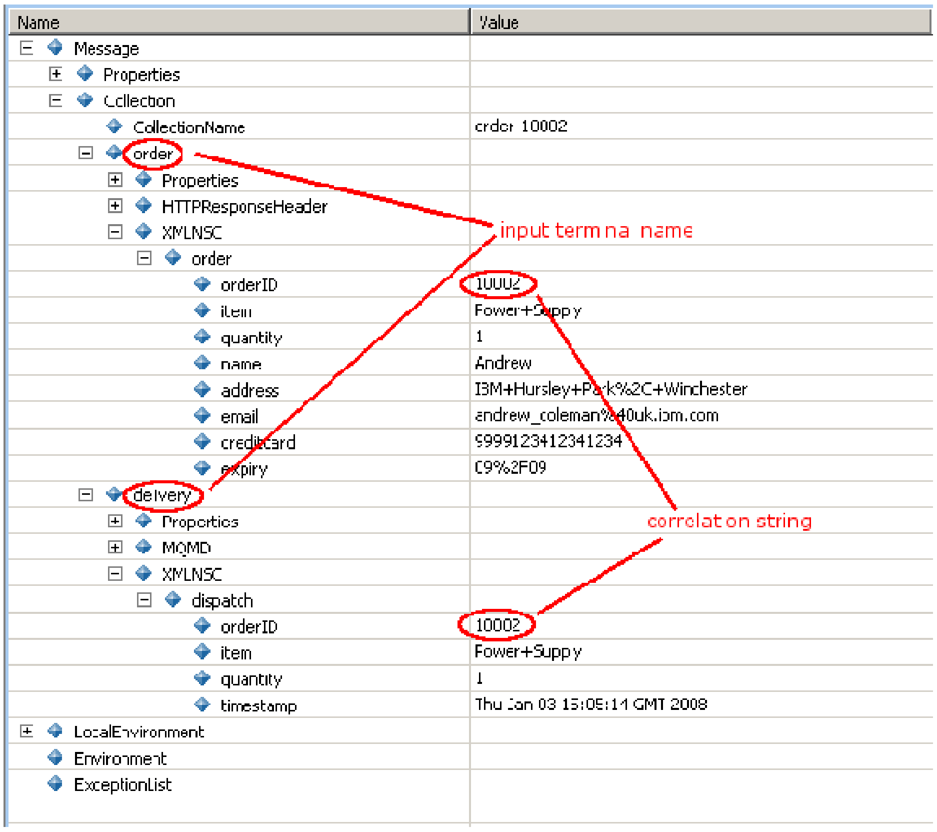
Terminal	Quantity	Timeout	Correlation path	Correlation pattern
order	1	0	$Body/order/orderID	
delivery	1	0	$Body/dispatch/orderID	

Collection name: order-*

Collection expiry: 10800

The collection message produced by the Collector node has the structure shown in Figure 5 below. The message body is a folder named Collection, which has an attribute named CollectionName with a value specified in the node properties. Each message in the collection is held as a sub-tree under a folder named after the input terminal on which the message arrived:

Figure 5. Structure of the collection message



```
Name                                Value
Message
  Properties
  Collection
    CollectionName                  order 10002
    order
      Properties
      HTTPResponseHeader
      XMLNSC
        order
          orderID                   10002
          item                      Power+Supply
          quantity                  1
          name                      Andrew
          address                   IBM+Hursley+Park%2C+Winchester
          email                     andrew_coleman%40uk.ibm.com
          creditcard                9999123412341234
          expiry                    09%2F09
    delivery
      Properties
      MQMD
      XMLNSC
        dispatch
          orderID                   10002
          item                      Power+Supply
          quantity                  1
          timestamp                 Thu Jan 03 15:05:14 GMT 2008
LocalEnvironment
Environment
ExceptionList
```

Message grouping and message batching

The Collector node has a flexible mechanism for defining message collections, with two common patterns for grouping messages:

- **Grouping** -- You can combine messages from different sources and of different formats into the same collection, by adding a new dynamic input terminal to the Collector node for each input message type. The scenario in this article uses this technique. Usually, you need to correlate the messages by matching some aspect of the message content, as described in the next section.
- **Batching** -- You can form collections from a single source by collecting multiple messages on a single input terminal to form a larger batched message. This is the opposite of message shredding, in which a large message is split into smaller component messages. Two parameters (per input terminal) on the Collector node control message batching:
    - **Quantity** -- Configures how many messages arriving on this input terminal should be accepted into each collection. Zero (or unset) means infinite (can only be infinite if Timeout is finite). Default value is 1.
    - **Timeout** -- Maximum time in seconds that the input terminal should accept messages following the arrival of the first message. Zero (or unset) means infinite (can only be infinite if Quantity is finite). If both Quantity and Timeout are finite, then the event handler is satisfied when the first of these two conditions is met. Default value is unset (infinite / no timeout).

You can combine both grouping and batching techniques in a single collection definition. You can also batch complete collections for later propagation. In addition to the dynamic input terminals that take incoming messages, there is a static input terminal called Control, whose purpose is to allow an external resource to trigger the output from the Collector node. Details are controlled via the Event coordination property settings:

- **Disabled** -- Messages to the control terminal are ignored, and collections are propagated when they are complete. This is the default setting.
- **All complete collections** -- When collections are complete, they are held on a queue. When a message is received on the control terminal, all collections on this queue are propagated to the Out terminal.
- **First complete collection** -- When collections are complete, they are held on a queue. When a message is received on the control terminal, the first collection on this queue is propagated to the Out terminal. If the queue is empty when this control message arrives, the Collector node is put into a pending state such that when the next complete collection becomes available, it is immediately propagated to the Out terminal.

The content of the message received on the control terminal is not examined, and this message is discarded on receipt. Incomplete collections that have exceeded the Collection expiry are immediately propagated to the Expire terminal regardless of the setting of the Event coordination property (see Collection expiry section).

Correlation strings and wildcards

Understanding message correlation is crucial to getting the most value from the Collector node. Without message correlation, the Collector node simply groups messages based on their order of arrival. While this may be suitable for some applications, most event-driven systems require messages to be grouped according to either content or a property in a message header. The

Collector node ensures that all messages within a collection have identical correlation strings. The correlation string is calculated from two properties: the correlation path and the correlation pattern. These properties are repeated for each dynamic input terminal, allowing messages of different format to be grouped in a collection.

The correlation path is a standard XPath 1.0 expression that is evaluated against the incoming message to produce a string. The string could simply be the string value of the element pointed to by the location path, but more complex expressions can be built up in XPath involving numeric and string operators. The resultant string is then compared with the correlation pattern, if that has been configured. The correlation pattern must contain one and only one wildcard character *. The portion of the string that matches that wildcard becomes the final correlation string. For example, if the correlation path pointed to a field in the message containing a filename, data1.txt and the correlation pattern was set to *.txt, then the correlation string would be data1. This mechanism is a simple way of extracting sub-strings from a message element, although XPath itself has the ability to manipulate strings.

The value of this correlation string * can be referenced in the Collection name property of the node. The value of this property gets attached to each collection message in an attribute called CollectionName. The example above shows how the collection name of order-* was expanded to order-10002 at runtime. The wildcard value is also written into the LocalEnvironment in a field called WildcardMatch in the Wildcard folder.

Processing the collection message

You cannot propagate a message collection directly to an output terminal because it has no single serialized form, and the component messages could be owned by more than one parser (one could be XML and the other could be binary). So the collection message must first be transformed in a Compute node using either ESQL or Java. In this example we want to extract information contained in both parts of the collection and generate an e-mail to the customer. The following code snippet shows how to do so in a JavaCompute node (Generate delivery note), using XPath to extract values from the collection:

```
MbMessage outMessage = new MbMessage();
MbMessageAssembly outAssembly = new MbMessageAssembly(inAssembly, outMessage);
MbElement outRoot = outMessage.getRootElement();

// extract required info from the collection
String orderNumber =
 (String)inMessage.evaluateXPath(&quot;string(/delivery/XMLNSC/dispatch/orderID)&quot;);
String name =
 (String)inMessage.evaluateXPath(&quot;string(/order/XMLNSC/order/name)&quot;);
String email =
 (String)inMessage.evaluateXPath(&quot;string(/order/XMLNSC/order/email)&quot;);
String dispatchTime =
 (String)inMessage.evaluateXPath(&quot;string(/delivery/XMLNSC/dispatch/timestamp)&quot;);

// create email header and setup To: and Subject: fields
MbElement header = outRoot.createElementAsLastChild(&quot;EMAILHDR&quot;);
header.createElementAsLastChild(MbElement.TYPE_NAME_VALUE, &quot;To&quot;,
        URLDecoder.decode(email, &quot;UTF-8&quot;));
header.createElementAsLastChild(MbElement.TYPE_NAME_VALUE, &quot;Subject&quot;,
        &quot;Order &quot; + orderNumber + &quot; has been dispatched&quot;);
```

```
String content = &quot;Dear &quot; + name + &quot;,\n\nYour order was dispatched at &quot;
        + dispatchTime + &quot;.  The order number is &quot; + orderNumber;

// the body of the email is written as a BLOB
outRoot.createElementAsLastChildFromBitstream(content.getBytes(), MbBLOB.PARSER_NAME,
        null, null, null, 0, 0, 0);
```

The message is finally propagated to an EmailOutput node (Notify customer) for delivery via a specified SMTP server.

Collection expiry

One problem with holding on to messages waiting for a collection to complete is the possibility that one of the required component messages does not arrive in a timely manner. The Collector node has a mechanism to guard against this eventuality in the form of a Collection expiry property. If configured, any partly built collection will not be held for more than the time specified in this property. Upon expiry, the messages received so far will be built into a message collection and propagated to the Expire terminal on the Collector node.

This expiry mechanism is put to good use in this scenario. The Warehouse is supposed to dispatch each order and send a message back to the Admin message flow within three hours of receipt. By setting the Collection expiry property of the Collector node to 3 hours (10800 seconds), the incomplete collection will be sent to the Expiry terminal if this condition is not met. The collection can then be routed for special handling in the message flow. The following Java code extracts the WildcardMatch field from the LocalEnvironment generated by the Collector node, and generates an error message to go on a FAILURE MQ queue:

```
MbMessage outMessage = new MbMessage();
MbMessageAssembly outAssembly = new MbMessageAssembly(inAssembly, outMessage);
MbElement outRoot = outMessage.getRootElement();

// incomplete collection - extract the orderID from the WildcardMatch field
String orderNumber = (String)inAssembly.getLocalEnvironment().getRootElement()
                          .evaluateXPath(&quot;string(Wildcard/WildcardMatch)&quot;);
String content = &quot;Order &quot; + orderNumber +
        &quot; was not dispatched within agreed time limit&quot;;
outRoot.createElementAsLastChildFromBitstream(content.getBytes(), MbBLOB.PARSER_NAME,
                                              null, null, null, 0, 0, 0);
```

Conclusion

Using a Web-based ordering system as an example, this article has introduced the WebSphere Message Broker V6 Collector node and shown you how to configure it to enable message flows to cross-reference messages that pass through the flow at different times and in different contexts. This technique forms the basis of event-driven processing systems that require the correlation of messages to extract meaningful information. The article also described additional features of the Collector node, including grouping or batching matches based on a count or a period of time, as well as specifying a correlation path and expressions that are different for each message source.

The Collector node contains a comprehensive set of functions that support event processing across a range of transports, thus enabling you to build new and more complex processing scenarios with WebSphere Message Broker in which processing is no longer controlled by a single event, but instead by a series of events. Add to this the ability to group messages not just by time of arrival but also based on message content, and you have some powerful message processing capabilities at your disposal.