# Grouping messages using the WebSphere MQ Java and JMS APIs

This article introduces the message group support in WebSphere MQ, and explains how you can use it to provide logical message ordering and enable the grouping of related messages. The article demonstrates grouping using the WebSphere MQ Java classes and shows you how to implement this same behavior using the JMS API. It then proposes a solution and illustrates how to apply it when receiving message groups asynchronously in WebSphere Application Server or some other J2EE application server.

## Introducing message groups

IBM® WebSphere® MQ cannot always guarantee the ordering of messages between a sending and receiving application. If three messages are sent in the order `A B C`, they may not arrive in that same order if, for example, the intervening network distributes the messages across a cluster and then recombines them. But what if message order is vital to the functioning of the application? Imagine a scenario in which message `B` tells the application to ignore the previous message. The meaning of the sequence would then be very different if the messages arrived in the order `C B A`.

WebSphere MQ addresses this issue through message grouping. The originating application can specify that it is sending messages `A`, `B` and `C` as part of a group. Each message in the group is assigned a sequence number starting at 1. The receiving application can then specify that it wishes to receive the messages in this logical order, as opposed to the physical order in which the messages arrive at a destination. Now, even if message `B` or `C` arrives first, they will not be passed to the application immediately because they do not have sequence number 1.

Message groups can serve another purpose. Sometimes, message order may not matter, but it may be important that a collection of messages are processed together, both spatially and temporally. For example, consider an application that sends a message each time an item is added to an online shopping cart. The items in the shopping cart may need to be processed together, perhaps to aggregate them into a single order message. You could manage this aggregation by placing the messages in a message group. The recipient of the messages can specify that they do not want to receive any of the messages in the group until all of them have arrived at the destination. In this scenario, it is also important that all of the messages are received in the same place. If, for scalability reasons, there are multiple consumers at the destination, it is vital that all messages representing items in the same order are delivered to the same consumer, and message groups can ensure that this happens. The concept of message groups is distinct from that of message segmentation, which means that a large message has been broken down into smaller messages on sending and should be re-assembled into the original message upon receipt. Each of the entities in a message group is an entire message. You can break down the messages within a message group using message segmentation, but we will not consider that option in this article.

## Using the WebSphere MQ Java API

### Exception handling

To improve clarity, exception handling logic has been removed from the code listings in this article. For the example classes containing the complete code, see the Download section.

We'll now take a look at the practicalities of sending and receiving message groups using the WebSphere MQ Java™ API.

### Sending a message group

Listing 1 below illustrates the code required to use the WebSphere MQ Java API to send a group of five messages to the queue `default` on the queue manager `QM_host`:

**Listing 1. Sending a message group using the WebSphere MQ Java API**

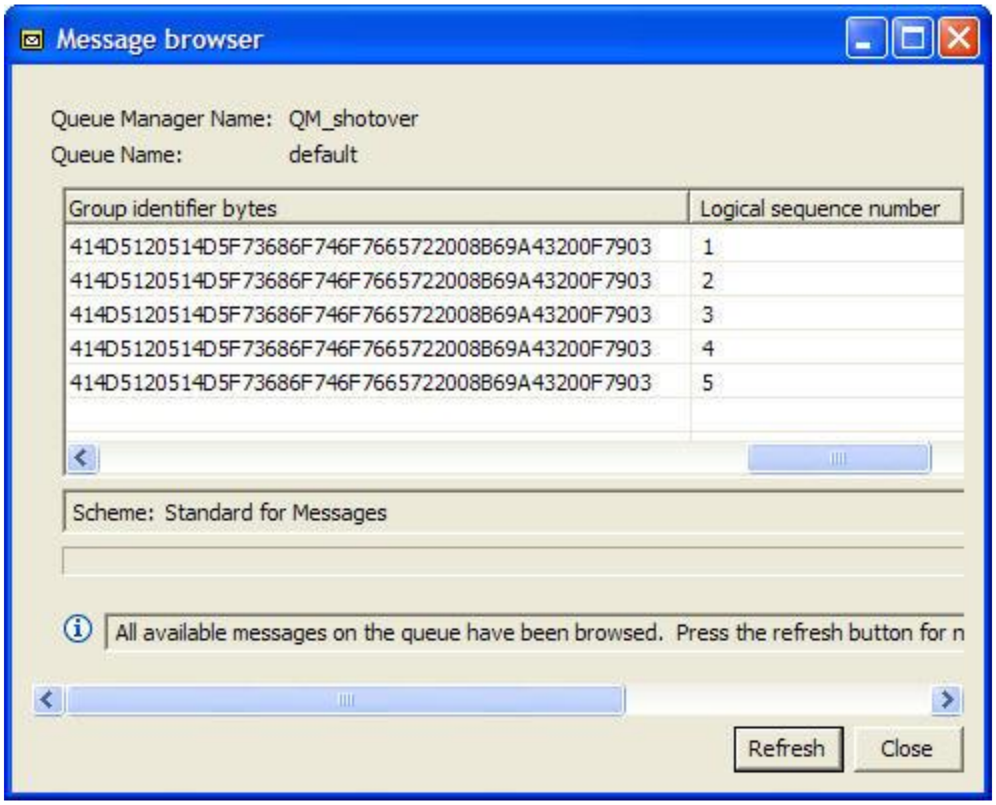
```
MQQueueManager queueManager = new MQQueueManager("QM_host");
MQQueue queue = queueManager.accessQueue("default", MQC.MQOO_OUTPUT);

MQPutMessageOptions pmo = new MQPutMessageOptions();
pmo.options = MQC.MQPMO_LOGICAL_ORDER;

for (int i = 1; i <= 5; i++) {

    MQMessage message = new MQMessage();
    message.format = "MQSTR";
    message.writeString("Message " + i);

    if (i < 5) {
        message.messageFlags = MQC.MQMF_MSG_IN_GROUP;
    } else {
        message.messageFlags = MQC.MQMF_LAST_MSG_IN_GROUP;
    }

    queue.put(message, pmo);
}

queue.close();
queueManager.disconnect();
```

The example begins by connecting to the queue manager and opening a queue handle for output. From the point of view of message groups, the first important thing to note is that the constant `MQPMO_LOGICAL_ORDER` is added to the put message options. This value indicates to the queue manager that the application is going to put each of the messages in the group in sequential order and that all of the messages in one group will be put by this client before any of the messages in the next.

The code then loops five times, putting a new message on each occasion. (The message format is set to `MQSTR` so that we may later receive the messages as JMS text messages.) For the first four messages, the message flag `MQMF_MSG_IN_GROUP` is set to indicate that the message should be part of the current group. The fifth message has the message flag `MQMF_LAST_MSG_IN_GROUP` set to indicate that it is the final message in the group. The next time a message is put with the `MQMF_MSG_IN_GROUP` flag, a new group will automatically be started.

The example ends by closing the queue handle and disconnecting from the queue manager. Having run this code, Figure 1 below illustrates the result of browsing the group of messages using WebSphere MQ Explorer:

**Figure 1. Browsing the message group in WebSphere MQ Explorer**



Queue Manager Name: QM_shotover
Queue Name: default

| Group identifier bytes | Logical sequence number |
|---|---|
| 414D5120514D5F73686F746F7665722008B69A43200F7903 | 1 |
| 414D5120514D5F73686F746F7665722008B69A43200F7903 | 2 |
| 414D5120514D5F73686F746F7665722008B69A43200F7903 | 3 |
| 414D5120514D5F73686F746F7665722008B69A43200F7903 | 4 |
| 414D5120514D5F73686F746F7665722008B69A43200F7903 | 5 |

Scheme: Standard for Messages

All available messages on the queue have been browsed. Press the refresh button for n

Each of the messages has been allocated the same 24-byte group identifier along with a logical sequence number running from 1 through 5.

The use of the `MQPMO_LOGICAL_ORDER` put message option is purely for convenience. It is possible for an application to not use this flag and to set the group identifier and sequence numbers explicitly, which may be necessary if messages are to be sent out of order or interleaved with other message groups. The message flags should still be set to indicate that a message is in a group and whether it is the last message. Another scenario where this may be useful is if the messages in a group are spread over a long period of time. An application may send the first few messages in a group using logical message ordering, after which the system fails. When the application restarts, it can continue with the message group by sending the next message without logical ordering and explicitly set the group identifier to be the one used for the earlier messages along with the next sequence identifier. At this point, it may switch back to using logical message ordering for the subsequent messages. The queue manager will then continue to use the same group identifier and increment the sequence number each time.

You can combine the use of message groups with transactions. If the first message is put under a transaction, then all of the other messages must be put under a transaction if they use the same queue handle. However, each message does not need to be in the same transaction.

## Receiving a message group

Having sent our messages in a group, we now want to receive them again in the same order. Listing 2 below gives an example of how to do this using the WebSphere MQ Java API:

**Listing 2. Receiving a message group using the WebSphere MQ Java API**

```
MQQueueManager queueManager = new MQQueueManager("QM_host");
MQQueue queue = queueManager.accessQueue("default", MQC.MQOO_INPUT_AS_Q_DEF);

MQGetMessageOptions gmo = new MQGetMessageOptions();
gmo.options =  MQC.MQGMO_LOGICAL_ORDER | MQC.MQGMO_ALL_MSGS_AVAILABLE;

gmo.matchOptions = MQC.MQMO_NONE;

MQMessage message = new MQMessage();

do {

    queue.get(message, gmo);
    int dataLength = retrievedMessage.getDataLength();
    System.out.println(message.readStringOfCharLength(dataLength));

    gmo.matchOptions = MQC.MQMO_MATCH_GROUP_ID;

} while (gmo.groupStatus != MQC.MQGS_LAST_MSG_IN_GROUP);

queue.close();
queueManager.disconnect();
```

As before, the code begins by connecting to the queue manager and opening the queue handle, but this time for retrieving messages using the default queue definition. We specify two get message options: `MQGMO_LOGICAL_ORDER` indicates that we wish to receive the messages in logical order, in other words, the message with sequence number 1 should be received first, then number 2, and so on. The second option,`MQGMO_ALL_MSGS_AVAILABLE`, indicates that we don't wish to receive any messages in a group until all of the messages are available. This option prevents us from starting to process messages in a group only to discover that subsequent messages haven't been sent yet, let alone arrived.

For the first `get`, we specify that no match options are required -- we are prepared to receive the first message in any group. For subsequent iterations, we specify the `MQMO_MATCH_GROUP_ID` option, which indicates that we only wish to receive a message with a matching group identifier. We reuse the same message object for each iteration and consequently, for the second get, it will contain the group identifier of the first message received. Each get updates the group status field of the get message options. When it is set to `MQGS_LAST_MSG_IN_GROUP`, we know that we have received all of the messages in the group.

As in the previous listing, be sure to clean up on completion by closing the queue handle and disconnecting from the queue manager.

## Using the WebSphere MQ JMS API

## Specification versions

The JMS examples in this article use the unified domain interfaces from JMS 1.1. However, they could be rewritten to use the point-to-point or publish-subscribe interfaces available with earlier versions of WebSphere MQ JMS. Similarly, the MDB example available in the Download section could be made to work in a J2EE 1.3 application server by using an EJB 2.0 deployment descriptor and the JMS 1.0.2b interfaces.

At this point you may ask why the examples are using the WebSphere MQ Java API. In this age of standards, shouldn't we be using the Java Message Service (JMS) API? Unfortunately, as with most standard specifications, JMS represents the lowest common denominator in terms of functionality supported by messaging systems. Consequently, not all of the behavior supported by WebSphere MQ can be expressed via this API, and message groups fall into that category. The JMS specification does define two properties named `JMSXGroupID` and`JMSXGroupSeq`, and specifies that they represent the identity of the group a message is part of, and the sequence number within that group. The JMS specification does not, however, provide any support for using these properties. All is not lost though -- with a few workarounds, we can still replicate our existing behavior using these properties.

### Sending a message group

Let's start by looking at the sending application. As mentioned above, the put message option `MQPMO_LOGICAL_ORDER` was simply an instruction to the queue manager to automatically allocate message group identifiers and sequence numbers. The example in Listing 3 below demonstrates how, in the absence of this option in the JMS API, we can set these properties explicitly.

**Listing 3. Sending a message group using the WebSphere MQ JMS API**

```
MQConnectionFactory factory = new MQConnectionFactory();
factory.setQueueManager("QM_host")
MQQueue destination = new MQQueue("default");
destination.setTargetClient(JMSC.MQJMS_CLIENT_NONJMS_MQ);
Connection connection = factory.createConnection();
```

```
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
MessageProducer producer = session.createProducer(destination);

String groupId = "ID:" + new BigInteger(24 * 8, new Random()).toString(16);

for (int i = 1; i <= 5; i++) {

    TextMessage message = session.createTextMessage();
    message.setStringProperty("JMSXGroupID", groupId);
    message.setIntProperty("JMSXGroupSeq", i);

    if (i == 5) {
        message.setBooleanProperty("JMS_IBM_Last_Msg_In_Group", true);
    }

    message.setText("Message " + i);
    producer.send(message);

}

connection.close();
```

The example begins by programmatically constructing a connection factory and destination. These administered objects could also have been obtained from a repository such as Java Naming and Directory Interface (JNDI). Next we create the usual JMS artifacts required to send a message, then generate a group identifier by taking a random 24-byte `BigInteger` and converting it to a hexadecimal string. As with message identifiers, the WebSphere MQ JMS API expects the group identifier to be prefixed by the string `ID:`

The code then iterates to send the five messages. The group identifier is set to the `JMSXGroupId` string property and the sequence number to the`JMSXGroupSeq` integer property. The API assumes that if the group identifier is set, then the message is part of a group. Therefore all that remains is to indicate the last message in the group, which we do by setting the Boolean property `JMS_IBM_Last_Msg_In_Group` to true.

If you run this code and then browse the queue using WebSphere MQ Explorer, you will see that the message descriptor properties have been set as before. We should now be able to run our original WebSphere MQ Java receiver to pick up this group of messages. To make this possible, we specified the target client as `MQJMS_CLIENT_NONJMS_MQ` in Listing 3 above, which ensures that an RFH2 header is not added to the body of the message, which would confuse the non-JMS client.

## Receiving a message group

Unfortunately, replicating the behavior for receiving message groups with JMS is not quite so easy. Logical ordering is fairly simple to do using message selectors. We start with a selector matching any message with a sequence number of 1, and once we have that message, we determine which group it is in. We then set up a second selector that specifies a sequence number of 2 along with the group identifier. We continue incrementing the sequence number until we receive a message that has the `JMS_IBM_Last_Msg_In_Group` property set. The hard part is trying to reproduce the behavior of the `MQGMO_ALL_MSGS_AVAILABLE` option. Listing 4 shows one possible solution:

**Listing 4. Receiving a message group using the WebSphere MQ JMS API**

```
MQConnectionFactory factory = new MQConnectionFactory();
factory.setQueueManager("QM_host")
MQQueue destination = new MQQueue("default");
Connection connection = factory.createConnection();
connection.start();
Session session = connection.createSession(true, Session.AUTO_ACKNOWLEDGE);

MessageConsumer lastMessageConsumer =
    session.createConsumer(destination, "JMS_IBM_Last_Msg_In_Group=TRUE");
TextMessage lastMessage = (TextMessage) lastMessageConsumer.receiveNoWait();
lastMessageConsumer.close();

if (lastMessage != null) {

    int groupSize = lastMessage.getIntProperty("JMSXGroupSeq");
    String groupId = lastMessage.getStringProperty("JMSXGroupID");

    boolean failed = false;

    for (int i = 1; (i < groupSize) && !failed; i++) {

        MessageConsumer consumer = session.createConsumer(destination,
            "JMSXGroupID='" + groupId + "'AND JMSXGroupSeq=" + i);
        TextMessage message = (TextMessage)consumer.receiveNoWait();

        if (message != null) {
            System.out.println(message.getText());
        } else {
            failed = true;
        }

        consumer.close();

    }

    if (failed) {
        session.rollback();
    } else {
        System.out.println(lastMessage.getText());
        session.commit();
    }

}
```

```
connection.close();
```

As before, we begin by creating all the JMS resources necessary to consume a message. This time we must start the connection; otherwise we won't receive any messages at all. The key to this solution is that we attempt to receive the last message in a group first. We do this by creating a consumer with a message selector of `JMS_IBM_Last_Msg_In_Group=TRUE`. If the messaging topology is such that ordering cannot be guaranteed, then we cannot be 100% sure that all of the other messages in the group have arrived. In a moment we'll see how we deal with the case when they have not all arrived.

If we receive the last message in a group, then we can obtain the group identifier and use its sequence number to determine the size of the group. With this information, we then iterate, attempting to receive all of the other messages in the group in sequence order starting at the beginning. We achieve this by setting up a new consumer on each iteration that selects based on the group identifier and sequence number that we require. Now, if for some reason the message has not arrived yet, we will get `null` back from the `receiveNoWait` method. At this point we set the `failed`flag, which causes us to drop out of the loop.

Look back at the line where we created the JMS session. You'll see that, unlike the previous example code, the first parameter has been set to`true` to indicate that the send and receive operations performed on this session should be performed as part of a local transaction. This means that, if the `failed` flag has been set because a message has not been received, we can roll back the transaction and return all of the other messages in the group back to the queue. If we did successfully receive all of the messages, then we must remember to commit the transaction. Otherwise, when we close the connection, the transaction will be rolled back on our behalf.

Unfortunately, whilst we are repeatedly rolling back one incomplete group, there may be other groups available on the destination that are complete but we are not reaching. This problem can be solved by temporarily excluding the incomplete group from the message selector, or by copying the group to an in-memory or persistent store for completion later. The next section examines how to address this problem in one particular environment -- the application server.

## Receiving message groups in a J2EE application server

Listing 4 above shows how to receive a message group using the WebSphere MQ JMS API. But in an application server environment, message receipt is typically performed via a message-driven bean (MDB). How can we adapt our approach so that it will work in this situation? It is still possible to configure a message selector for an MDB but it is statically defined by the administrator. We will therefore configure this selector with`JMS_IBM_Last_Msg_In_Group=TRUE` so that the MDB is always passed the last message in a group. We need this first message to be received as part of a transaction, and therefore the MDB should also be configured to use Container Managed Transactions (CMT) with a transaction attribute of `RequiresNew`. Here is the code for the `onMessage` method of our MDB:

**Listing 5. A message-driven bean for receiving message groups**

```
public void onMessage(Message lastMessage) {

    InitialContext context = new InitialContext();
    ConnectionFactory factory =
        (ConnectionFactory) context.lookup("java:comp/env/jms/factory");
    Destination destination =
        (Destination) context.lookup("java:comp/env/jms/destination");
    Connection connection = factory.createConnection();
    connection.start();
    Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

    int groupSize = lastMessage.getIntProperty("JMSXGroupSeq");
    String groupId = lastMessage.getStringProperty("JMSXGroupID");

    boolean failed = false;

    for (int i = 1; (i < groupSize) && !failed; i++) {

        MessageConsumer consumer = session.createConsumer(destination,
            "JMSXGroupID='" + groupId + "'AND JMSXGroupSeq=" + i);
        TextMessage message = (TextMessage) consumer.receiveNoWait();

        if (message != null) {
            System.out.println(message.getText());
        } else {
            failed = true;
        }

        consumer.close();

    }

    if (failed) {
        _context.setRollbackOnly();
        Thread.sleep(5000);
    } else {
        System.out.println(((TextMessage) lastMessage).getText());
    }
    connection.close();

}
```

As appropriate for a J2EE environment, the example now obtains the connection factory and destination from JNDI. The transacted parameter passed when the JMS session is created is ignored this time -- all messages will be received as part of global transaction started by the container. If the receipt of one of the messages fails, we mark the global transaction for rollback and then sleep for 5 seconds. Since the transaction is not actually rolled back until the method exits, the messages that have been received are held locked on the destination, which prevents other instances of the MDB from trying to receive this same incomplete group. Instead, these other instances can be attempting to receive other groups from the destination. When the method finally exists, the transaction will be rolled back and the messages will be available for receipt again, at which point hopefully the entire group has arrived.

In this environment, try to limit the number of times that you attempt to receive the group, since each time you roll back a transaction, the message delivery count is incremented for the messages that were received. If you call the above code repeatedly, this count may reach the backout threshold configured on the destination, at which point the messages will be re-queued and no longer available for receipt.

If the entire message group is received successfully, then the method completes and the container commits the transaction, thereby removing the messages from the destination.

## Conclusion

This article has described what messages groups are and why you might want to use them. You have seen sample code showing how message groups can be sent and received using both the WebSphere MQ Java and JMS APIs, as well as how an MDB may be used to receive a group of messages in a J2EE application server. While somewhat simplified, these examples should help you understand how message groups can be handled effectively in these environments.