

---

# Messaging Design Considerations

Until now we have been focusing on the concepts and semantics of JMS messaging. This chapter expands on those concepts and introduces some of the design considerations that need to be addressed when building messaging systems.

## Internal Versus External Destination

The Java Message Service API consists of a set of standard interfaces that must be implemented by open source or commercial vendor products called *JMS providers*. There are many different types of open source and commercial JMS providers ranging from J2EE application servers (e.g., JBoss, WebLogic, IBM WebSphere, Oracle AS) to standalone solutions (e.g., ActiveMQ, SonicMQ, IBM WebSphere MQ).

Because most Java EE application servers can also serve as JMS providers, a common design choice is deciding whether to leverage your current application server environment or use an external JMS provider. After all, why complicate your architecture and incur additional middleware licensing fees if you can simply utilize your existing application server? While this seems like a straightforward question, there are several implications and limitations to using an application server as a JMS provider. As you will see in this section, using an existing Java EE application server environment versus an external standalone JMS provider is an important design decision that carries with it many implications with respect to an overall messaging solution.

There are two primary deployment topologies with respect to JMS providers: *Internal Destination* and *External Destination*. The Internal Destination topology refers to queues and topics that are administered by an application server (e.g., WebLogic) that also hosts web-based or server-based applications, whereas the External Destination Topology refers to queues and topics that are administered on a dedicated system outside of the context of web-based or server-based applications. The following sections describe the details and design considerations surrounding each of these design choices.

## Internal Destination Topology

As stated earlier, the Internal Destination topology refers to queues and topics that are administered by an application server that also hosts web-based or server-based applications. Since all Java EE 4 and above application servers are also JMS providers, this is something that is fairly easy to configure and use. Use of the Internal Destination topology is common for message-driven beans (Chapter 8), but is certainly not a requirement.

As illustrated in Figure 11-1, with the Internal Destination topology queues and topics are administered by the Java EE application server, which also hosts the applications deployed as either WAR (Web Archive) files, EAR (Enterprise Archive) files, or simple JAR (Java Archive) files. External message consumers or receivers must connect to that application server (usually through TCP/IP) to send and receive messages.

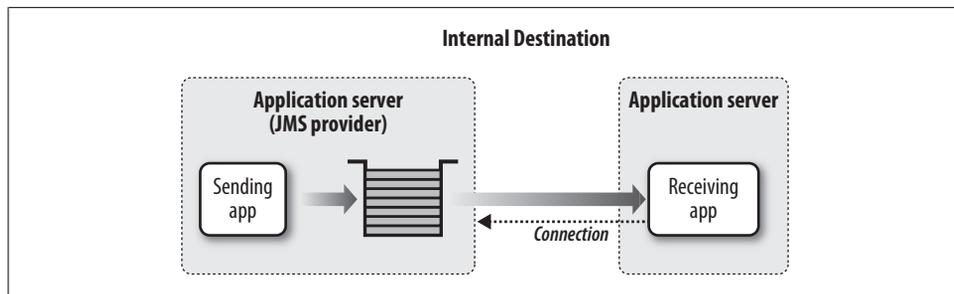


Figure 11-1. Internal Destination

There are a variety of issues associated with the Internal Destination topology that are important to consider. First, using a Java EE application server as the JMS provider restricts message producers and consumers to the Java platform (i.e., JMS) only. This means that your messaging solution will not support heterogeneous messaging clients. While this restriction may not be an issue for you now, it may be an issue with respect to future expansion. With mergers and acquisitions on the rise, many companies find heterogeneous integration an important capability a system must support in order to maintain architectural vitality.

Maintaining a healthy separation of concerns is another issue with the Internal Destination topology. Application servers that play a dual role of being an application hosting server and a messaging provider at the same time can slow down a system and create significant system bottlenecks. A messaging server consumes a significant amount of system resources, particularly with respect to CPU and available thread count. When an application server instance hosts web-based or server-based applications and acts as a JMS provider at the same time, resources needed by the applications may be consumed by message processing, thereby starving applications of much-needed system resources.

Message server availability is another issue associated with the Internal Destination topology. When the application server instance needs to be taken down for system maintenance or application deployment, your messaging system comes down as well, preventing external message producers or consumers from getting to the queues and topics.

Based on these issues, it is generally not a good idea to use an application server as a dual purpose server for both applications and message processing. That said, there are times when it might make sense to use the Internal Destination topology. One of these use cases is when you have a self-contained Java-based application that utilizes messaging to decouple internal components to reduce bottlenecks and increase scalability and throughput. In this scenario, it is unlikely that the queues and topics will be used outside of the context of the application, so using the Internal Destination topology in this case makes sense.

## External Destination Topology

With the *External Destination topology*, the JMS provider is deployed as a dedicated server that is separate from any Java EE application servers used to host applications (including messaging producers and consumers). As illustrated in Figure 11-2, the JMS provider server instance is only responsible for managing the queues and topics, leaving the Java EE application servers free to host web-based or server-based applications. This topology supports a healthy separation of concerns between application hosting and providing messaging services, resolving several of the issues encountered with the Internal Destination topology.

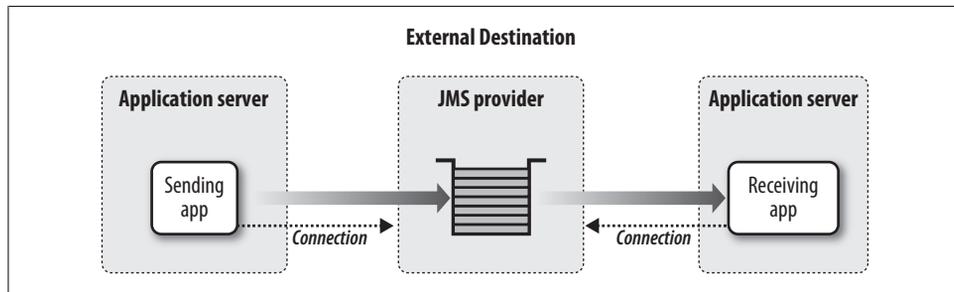


Figure 11-2. External Destination

While the external JMS provider can be deployed using a dedicated Java EE application server instance, the limitation of JMS-only messaging would still apply. For this reason, a standalone JMS provider such as ActiveMQ, SonicMQ, or IBM WebSphere MQ (to name a few) is typically used instead. These messaging providers support the JMS API, but also expose a native API for the programming languages supported by that provider. For example, as of the time of this writing ActiveMQ (a popular open source messaging

provider) supports thirteen different languages and platforms, including C, C++, C#, Perl, Ruby, and Smalltalk.

The External Destination topology supports heterogeneous integration through messaging, and also provides a high degree of separation between applications and the JMS provider. For instance, application servers can be taken down for maintenance or application deployment without affecting the rest of the system from a messaging standpoint. Other applications can continue to send and receive messages while application servers are unavailable.

The External Destination topology also allows the JMS provider to be co-located on the same physical machine or deployed on a dedicated physical machine. The choice between where the JMS provider is deployed is largely based on the throughput required by your messaging solution. Systems with low messaging throughput requirements (e.g., 50 messages per second) might be good candidates for co-location, whereas systems with high messaging throughput requirements (e.g., 2,000 messages per second) might warrant a separate physical machine.

## Request/Reply Messaging Design

In Chapter 4 we introduced point-to-point messaging using a simple request/reply model. In this scenario the message producer (`QBorrower`) sent a loan request to the message consumer (`QLender`) and waited (blocking wait) for a response from the `QLender` on whether the loan was approved or denied. To implement the request/reply model we used a technique known as *message correlation*, where messages sent to the response queue were correlated with the original message using the `JMSMessageID` and `JMSCorrelationID`. The following are the original `QBorrower` and `QLender` listings used to implement the request/reply scenario.

```
public class QBorrower {
    ...
    public QBorrower(String queuecf, String requestQueue,
                    String responseQueue) {
        try {
            ...
            // Lookup the request and response queues
            requestQ = (Queue)ctx.lookup(requestQueue);
            responseQ = (Queue)ctx.lookup(responseQueue);
        }
        ...
    }

    private void sendLoanRequest(double salary, double loanAmt) {
        try {
            // Create JMS message
            MapMessage msg = qSession.createMapMessage();
            msg.setDouble("Salary", salary);
            msg.setDouble("LoanAmount", loanAmt);
            msg.setJMSReplyTo(responseQ);
        }
    }
}
```

```

// Create the sender and send the message
QueueSender qSender = qSession.createSender(requestQ);
qSender.send(msg);

// Wait to see if the loan request was accepted or declined
String filter =
    "JMSCorrelationID = '" + msg.getJMSMessageID() + "'";
QueueReceiver qReceiver = qSession.createReceiver(responseQ, filter);
TextMessage tmsg = (TextMessage)qReceiver.receive(30000);
if (tmsg == null) {
    System.out.println("QLender not responding");
} else {
    System.out.println("Loan request was " + tmsg.getText());
}
}
...
}
...
}

```

To implement request/reply message processing in the preceding `QBorrower` class we had to create a `Queue` object for the response queue, create a message selector based on the `JMSCorrelationID` header property, and then create a `QueueReceiver` using the response queue and message selector. Then, in the `QLender` class, we had to set the `JMSCorrelationID` header property to the `JMSMessageID` of the original message when sending the reply:

```

public class QLender implements MessageListener {
    ...
    public void onMessage(Message message) {
        try {
            ...
            // Send the results back to the borrower
            TextMessage tmsg = qSession.createTextMessage();
            tmsg.setText(accepted ? "Accepted!" : "Declined");
            tmsg.setJMSCorrelationID(message.getJMSMessageID());

            // Create the sender and send the message
            QueueSender qSender =
                qSession.createSender((Queue)message.getJMSReplyTo());
            qSender.send(tmsg);
            ...
        }
    }
}

```

The message correlation code just shown was necessary to ensure that the message being received by the response queue was intended for the loan request originally sent. Keep in mind, other borrower clients may be making loan requests at the same time, creating multiple response messages in the loan response queue.

Another technique for accomplishing the same thing but with far less code is using the `javax.jms.QueueRequestor` class (or `javax.jms.TopicRequestor` class for topic-based

request/reply). With the `QueueRequestor` class, senders and receivers do not have to worry about setting the `JMSCorrelation` header property or even creating the corresponding `QueueReceiver` to receive the reply. Instead, the `QueueRequestor` class creates a unique *temporary queue* whose reference is passed to the message receiver through the `JMSReplyTo` header property. This temporary queue only has context for the communications between a sender and receiver for a specific request. By using a temporary queue, the `QBorrower` can be assured that the next available message in that queue is a response to the prior loan request message sent.

Sending a message and waiting (while blocking) for the response is done through a single request method call on the `QueueRequestor` class. The following revised `QBorrower` uses the `QueueRequestor` class to send the loan request and wait for a response:

```
public class QBorrower {

    public QBorrower(String queuecf, String requestQueue) {
        try {
            ...
            // Lookup the request queue
            requestQ = (Queue)ctx.lookup(requestQueue);
            ...
        }
        ...
    }

    private void sendLoanRequest(double salary, double loanAmt) {
        try {
            // Create JMS message
            MapMessage msg = qSession.createMapMessage();
            msg.setDouble("Salary", salary);
            msg.setDouble("LoanAmount", loanAmt);

            QueueRequestor requestor = new QueueRequestor(qSession, requestQ);
            TextMessage tmsg = (TextMessage)requestor.request(msg);
            if (tmsg == null) {
                System.out.println("Lender not responding");
            } else {
                System.out.println("Loan request was " + tmsg.getText());
            }
        }
        ...
    }
    ...
}
```

There are several differences between the modified `QBorrower` just shown and the original `QBorrower` from Chapter 4, even though they both do exactly the same thing. First, notice that the `QBorrower` constructor no longer requires a response queue name argument, nor does it need to do a JNDI lookup on the response queue. Since the `QueueRequestor` class creates a temporary queue for the loan response message, you no longer have to specify an administered queue to handle responses, nor do you need to manage

a separate queue in the JMS provider. The `QueueRequestor` class will take care of creating and destroying the temporary queue.

Second, notice that the `sendLoanRequest` method is significantly shorter than the prior version from Chapter 4. As a matter of fact, the act of sending the loan request and waiting for the response has been reduced to only two lines of code:

```
QueueRequestor requestor = new QueueRequestor(qSession, requestQ);
TextMessage tmsg = (TextMessage)requestor.request(msg);
```

The first line constructs a new `QueueRequestor` object using the `QueueSession` and loan request queue. The second line then sends the message and automatically blocks and waits for the response. A reference to the temporary queue created by the `QueueRequestor` class is passed to the receiver (in this case `QLender`) through the `JMSReplyTo` message header property.

Now that we are using a temporary queue, we can remove the code in the `QLender` class that set the `JMSCorrelationID`. Notice that because the `QLender` was already agnostic as to the response queue, the other code stays the same:

```
public class QLender implements MessageListener {
    ...
    public void onMessage(Message message) {
        try {
            ...
            // Send the results back to the borrower
            TextMessage tmsg = qSession.createTextMessage();
            tmsg.setText(accepted ? "Accepted!" : "Declined");

            //since we are using a temporary queue, we no longer
            //need to set the JMSCorrelationID property
            tmsg.setJMSCorrelationID(message.getJMSMessageID());

            // Create the sender and send the message
            QueueSender qSender =
                qSession.createSender((Queue)message.getJMSReplyTo());
            qSender.send(tmsg);
            ...
        }
    }
}
```

While this alternative request/reply technique simplifies the source code and reduces the number of required administered queues, it does have some limitations. First, there is no way to specify a timeout value with the `request` method on the `QueueRequestor` as we did with the `receive` method on the `QueueReceiver`:

```
//QueueReceiver
TextMessage tmsg = (TextMessage)qReceiver.receive(30000);

//QueueRequestor
TextMessage tmsg = (TextMessage)requestor.request(msg);
```

This is somewhat an issue in that if the message consumer is not responding, the message producer will appear to “hang” and might require a restart. If you did restart the message producer, the original message would still be sitting on the request queue waiting to be processed by the message consumer. Implementing a timeout would require you to override the `javax.jms.QueueRequestor.receive()` method, thereby possibly creating nonportable JMS code. When considering this approach the benefits of streamlined code should be weighed against the lack of a receive timeout capability.

Another limitation of the `QueueRequestor` is that the `QueueSession` cannot be transacted, and will not support the `CLIENT_ACKNOWLEDGE` message acknowledgment mode. These limitations should also be considered before using the `QueueRequestor` technique.

## Messaging Design Anti-Patterns

There are several messaging-related anti-patterns that manifest themselves in production environments. An anti-pattern is a practice that is repeated but produces negative results (unlike a pattern, which is a repeatable process that produces positive results). Three of the most common messaging anti-patterns are the *single-purpose queue*, *message priority overuse*, and *message header misuse*. This section will cover the details of each of these anti-patterns and describe ways to avoid them.

### Single-Purpose Queue

A common messaging anti-pattern is designing a system with only a single purpose queue. Typically this problem manifests itself when a single queue handles different types of messages (e.g., book orders, order status requests, and order cancellations), but problems can also occur when a single purpose queue is used for the same type of message (e.g., book orders). We will start with the first scenario since it is most common, and then move onto the second scenario, which is a little more subtle.

Systems that use a single purpose queue often have a single message listener class that acts as a router. The router listener receives the next message on the queue, determines the message type, and redirects processing to some other class to process that message. This design scenario is illustrated in Figure 11-3.

The routing rules used by the listener router can be based on the JMS message type (e.g., `TextMessage`, `StreamMessage`), a custom message property, or even the `JMSType` message header property. Since the `JMSType` header property may be used by the JMS provider, it is not a good idea to use it to store your own custom routing information or message type. In the following example, a single queue (`requestQueue`) is used to handle book orders, order status requests, and order cancellations. A custom message property is used to store the message type that is used to route the message to the particular class responsible for processing that message:

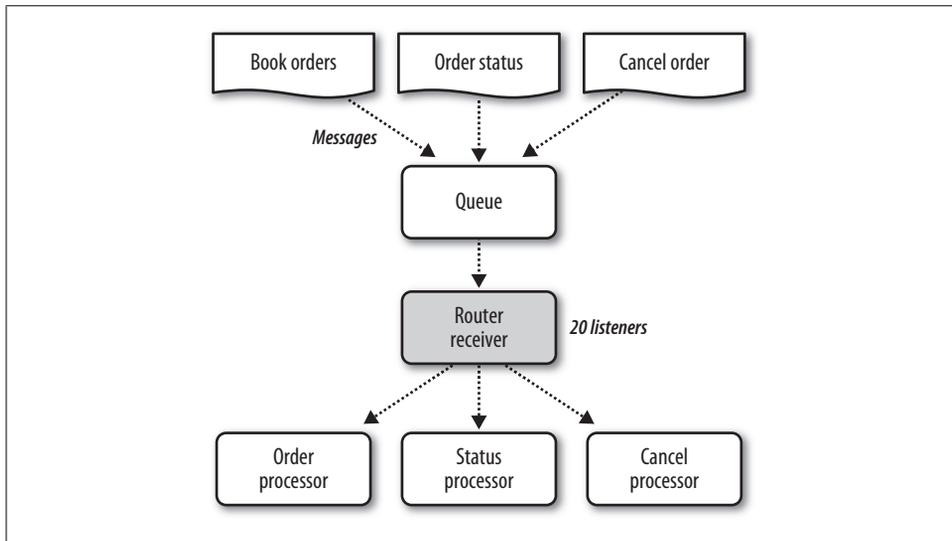


Figure 11-3. Single-purpose queue—different message types

```

public class QRouter implements MessageListener {

    private OrderProcessor orderProcessor = null;
    private StatusProcessor statusProcessor = null;
    private CancelProcessor cancelprocessor = null;

    ...
    public void onMessage(Message message) {
        try {
            //get the message payload
            String xml = ((TextMessage)message).getText();

            //get the message type
            int type = message.getIntProperty("type");
            if (type == NEW_BOOK_ORDER) {
                orderProcessor.placeOrder(xml);
            } else if (type == ORDER_STATUS) {
                statusProcessor.checkOrderStatus(xml);
            } else if (type == CANCEL_ORDER) {
                cancelprocessor.cancelOrder(xml);
            } else {
                throw new Exception("Invalid Order Type: " + type);
            }
        }
        ...
    }
    ...
}

```

In this code snippet, the XML application data is extracted from the message payload, then the message type is extracted from the message properties. Notice that the message type comes from a custom application property (`type`), not a standard header property. The `QRouter` class then analyzes the `type` value and redirects processing to one of three processor classes.

At first glance, this messaging design strategy may seem attractive due to the simplicity of the message processing. Adding a new messaging type (e.g., `VIEW_ORDER_HISTORY`) is simply a matter of adding the additional `if` statement in the `QRouter` class and adding the class or method to process that request. Since there is only a single queue, no additional messaging configuration is necessary to add the additional request:

```
public void onMessage(Message message) {
    try {
        ...
        //get the message type
        int type = message.getIntProperty("type");
        if (type == NEW_BOOK_ORDER) {
            orderProcessor.placeOrder(xml);
        } else if (type == ORDER_STATUS) {
            statusProcessor.checkOrderStatus(xml);
        } else if (type == CANCEL_ORDER) {
            cancelProcessor.cancelOrder(xml);
        } else if (type == VIEW_ORDER_HISTORY) {
            orderProcessor.viewHistory(xml);
        } else {
            throw new Exception("Invalid Order Type: " + type);
        }
    }
    ...
}
```

While this design approach seems to provide a lot of flexibility to the architecture, several inefficiencies manifest themselves in this design. First, since all messages are sent to the same queue, it is not possible to load balance the system based on the message type. For instance, in the preceding example, assume that there are 20 concurrent `QRouter` listener threads. This means 20 messages can be processed at the same time. However, let's say the distribution of message types is 80% `NEW_BOOK_ORDER`, 10% `ORDER_STATUS`, 7% `CANCEL_ORDER`, and 3% `VIEW_ORDER_HISTORY`. Therefore, if the queue contains 100 new book orders and a message is sent to cancel an order, the `CANCEL_ORDER` message will be placed on the queue at position 101 and not received until the previous 100 book orders are processed.

The issue is that the `QRouter` listener simply pulls off the next available message in the queue, regardless of the type. With this design it is not possible to tune the system to provide optimized throughput for the different message types. Using the original example, a better design approach would be to have three separate queues to handle the three message types and three separate message listeners, one for each queue. Figure 11-4 illustrates this approach.

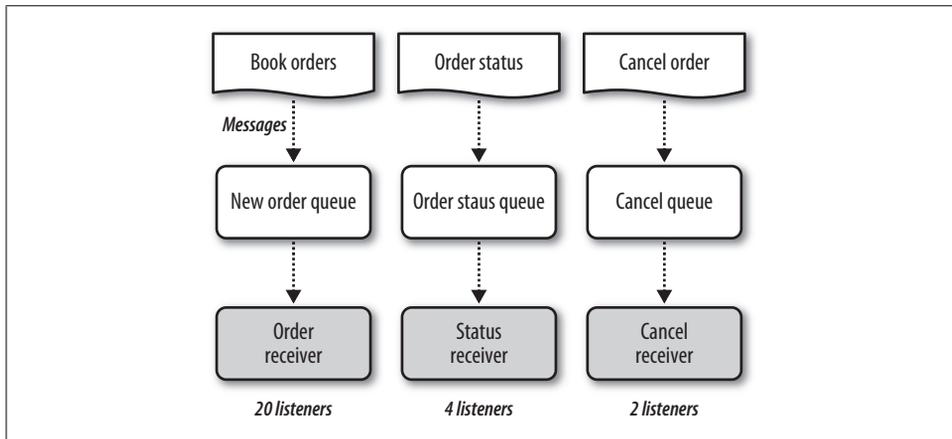


Figure 11-4. Multiple queues—different message types

Notice in Figure 11-4 how each message receiver can now be tuned to add the number of concurrent listener threads based on the message distribution. More importantly however, with this design 20 new book orders can still be processed concurrently, but order status requests and cancel order requests can immediately be processed at the same time. This is a significant improvement over the previous single-purpose queue design. With the multiple queue approach, adding new message types involves adding a new queue (configuration), changing the message producer code to send the new request to the new queue, and adding the new message listener. Since there is no central “routing listener,” no additional coding changes are required.

What about a single queue that handles the same message type? In most cases this is exactly what we want, but in some cases this too can present issues. Take, for example, the book order messages from the previous example. After applying the multiple queue approach, we now have a queue dedicated to book orders. However, suppose that book orders can come from an online web-based source but also in batch form from a book store. They are the same `NEW_BOOK_ORDER` message, but they come from different client channels (one online, one batch). Now suppose the online web-based channel requires immediate feedback about the book order, whereas the batch orders from the book store (typically hundreds of book orders in a single batch) do not require a response. This is where additional issues can arise.

Just because the message type (e.g., `NEW_BOOK_ORDER`) is the same does not necessarily mean the messages themselves are the same. In the previous example, even though the message format and structure are exactly the same, there are actually *two types* of `NEW_BOOK_ORDER` messages: online and batch. When this scenario occurs, the same problems described with the multiple message types can occur here as well. Consider for a moment the case where a batch order comes in for 400 books. The queue depth for the *new order queue* is now 400. Immediately after the batch messages are received, an online request is received for a `NEW_BOOK_ORDER`. The online order is placed at position

401 in the new order queue, and will not be processed for quite some time. Meanwhile, the online web-based customer is waiting for the response for the book order.

A temporary solution to this problem is to use *message priority* to assign a higher priority to online messages, therefore effectively moving online messages to the front of the queue. However, using this approach leads to another messaging anti-pattern known as *message priority overuse*. This messaging anti-pattern is described in the following section.

## Message Priority Overuse

In the previous section, we used message priority as a way to solve the problem where the same message type (e.g., `NEW_BOOK_ORDER`) is being used in two different ways. In the previous example, we had new book orders coming from an online web-based channel as well as a batch channel. Using message priority effectively moved the online orders ahead of the batch orders, solving the long wait problem for online book orders—or did it?

While the use of message priority as a way of processing certain messages faster may seem like a good permanent solution, the problem is that all of the listener threads available to process the higher priority messages may be tied up processing lower priority messages. Therefore, priority messages are processed slower than they otherwise could be.

In the example used in the previous section, online orders were given a higher priority than batch orders, meaning that they were pushed to the front of the queue. Suppose that, because of special validation and processing, batch orders take one minute to process, whereas online orders take 200 milliseconds to process. In this scenario most of the available message listeners will be tied up processing batch orders, leaving the online orders waiting on the queue to be processed. Once again, this is a case where the new order queue should be split into two queues: an online new order queue and a batch new order queue. This way, more listener threads can be assigned to the online new order queue where an immediate response is needed.

There are times when setting the message priority makes sense. However, a general rule of thumb to apply is as follows: when using message priority, always ask yourself if it would make more sense to use separate queues instead. Too often message priority is used to mask deeper rooted problems.

## Message Header Misuse

Most of the message header properties are set by the JMS provider, even though the header property setter methods are exposed to the developer through the JMS API. This creates potential hard-to-find bugs, particularly when setting the message expiration and message priority.

The expiration date of a message is contained in the `JMSEExpiration` message header with corresponding `getJMSEExpiration` and `setJMSEExpiration` methods to get and set the expiration date. All too often, a developer will attempt to set the expiration date of a message as follows:

```
public class QBorrower {
    ...
    private void sendLoanRequest(double salary, double loanAmt) {
        try {
            // Create JMS message
            MapMessage msg = qSession.createMapMessage();
            msg.setDouble("Salary", salary);
            msg.setDouble("LoanAmount", loanAmt);
            msg.setJMSReplyTo(responseQ);

            //set the message expiration to 30 seconds (incorrect!)
            msg.setJMSEExpiration(new Date().getTime() + 30000);

            // Create the sender and send the message
            QueueSender qSender = qSession.createSender(requestQ);
            qSender.send(msg);
        }
        ...
    }
    ...
}
```

Notice the use of the `setJMSEExpiration` method on the message object. This compiles fine and, when executed, will put the message on the queue. The message will then expire after 30 seconds if not received, right? Wrong. Using the code just shown, the message sent to the queue will never expire. Why? Because the time to live property on the message is set to zero (the default value). When the message is sent, the JMS provider adds the value in the time to live property to the current system time and sets the `JMSEExpiration` header property itself. In the previous code example, the `JMSEExpiration` header property was in fact getting overridden by the JMS provider.

This is a very common mistake and, unfortunately, is something that is rarely tested. So, if the `setJMSEExpiration` method is off-limits to application developers, what is the proper way to set the message expiration? There are two ways to set the message expiration. The first technique is to invoke the `setTimeToLive()` method on the `Message Producer` (`QueueSender` or `TopicPublisher`), which sets the message expiration for all messages sent using that sender:

```
//set the default message expiration for all messages to 30 seconds
QueueSender qSender = qSession.createSender(requestQ);
qSender.setTimeToLive(30000);
...
qSender.send(msg);
...
```

The other technique is to set the message expiration when sending the message:

```
QueueSender qSender = qSession.createSender(requestQ);
...
//set the message expiration for this message to 20 seconds
qSender.send(msg, DeliveryMode.PERSISTENT, 4, 20000);
...
```

Notice in this code snippet that messages sent using the `qSender` will, *by default*, have no message expiration. However, the message being sent in the code snippet has a message expiration of 20 seconds. Unfortunately, when using the second approach, you have to specify the message delivery mode and message priority as well.

There may be cases when you want to have a default message expiration for all messages sent by a `QueueSender` but still have the flexibility to override it for certain message types sent by that same `QueueSender`. In this case, you can use both forms together:

```
//set the default message expiration for all messages to 30 seconds
QueueSender qSender = qSession.createSender(requestQ);
qSender.setTimeToLive(30000);
...
qSender.send(msg1);

//this message should expire in 20 seconds
qSender.send(msg2, DeliveryMode.PERSISTENT, 0, 20000);
...
```

In this example, `msg1` will go on the queue and expire in 30 seconds (the default), whereas `msg2` will go on the same queue but expire in 20 seconds.

The same problem holds true for message priority, which is even harder to test than the message expiration. Here, a common mistake is to set the message priority directly using the `setJMSPriority()` method on the `Message` object prior to sending the message:

```
public class QBorrower {
    ...
    private void sendLoanRequest(double salary, double loanAmt) {
        try {
            // Create JMS message
            MapMessage msg = qSession.createMapMessage();
            msg.setDouble("Salary", salary);
            msg.setDouble("LoanAmount", loanAmt);

            //incorrect!
            msg.setJMSPriority(9);

            // Create the sender and send the message
            QueueSender qSender = qSession.createSender(requestQ);
            qSender.send(msg);
        }
    }
    ...
}
```

In this code, the message priority is set to **9**, indicating this is a high-priority message. However, when the message is sent, the message will have a priority of **4** (normal priority). The reason? Like the message expiration, the JMS provider will look at the message priority property on the message and invoke the `setJMSPriority` method prior to placing the message on the queue. Since the default message priority is **4** (normal priority), the message priority will not be set to a high priority message, as the developer had originally intended.

Like the message expiration, there are two ways of setting the message priority: you can invoke the `setPriority()` method on the `MessageProducer` (`QueueSender` or `TopicPublisher`) or set the message priority when sending the message:

```
//set the default message priority for all messages to 9 (high)
QueueSender qSender = qSession.createSender(requestQ);
qSender.setPriority(9);
...
qSender.send(msg1);

//this message is low priority
qSender.send(msg2, DeliveryMode.PERSISTENT, 1, 30000);
...
```

In this example, `msg1` will be sent with a priority of **9** (high priority), whereas `msg2` will be sent with a priority of **1** (low priority).

Understanding these messaging anti-patterns will help you build more robust messaging systems and help you avoid some of the more common mistakes associated with messaging.

