CHAPTER 2
Messaging

This chapter describes how an enterprise uses Luna to interface with message services with the following topics:

Message-service concepts

The Luna system enables your application to access both relational databases and non-relational data such as information from pre-relational legacy systems or information exchange via a Web server. A *message service* provides a means for a set of dissimilar data-management systems to exchange information. Messaging particularly suits trends in enterprise business practices, such as peer-to-peer computing, heterogeneous platforms, modular applications, and asynchronous business-to-business communications.

Two types of applications act as clients to the message service: *consumers* request information, *producers* maintain and distribute the source data of interest to consumers.

Each of many message-service vendors offers a proprietary API that supports an individual addressing syntax and routing architecture. Factors that differ among vendors include the services that each offers, such as authentication, delivery acknowledgment, transaction support, and asynchronous or synchronous communication. For efficient processing of disparate legacy and new applications, an enterprise might communicate across a variety of message services.

The Java Message Service (JMS) specification

The Java Message Service (JMS) specification attempts to eliminate concern with vendor-specific messaging so that an enterprise can request and consume relevant information from any producer and can produce and send information to any authorized requestor. To achieve vendor-independence, JMS defines an application programming interface (API) to a common set of message features. A piece of software called a *JMS provider* implements the JMS API, encapsulating the details of the specific message-service architecture. For background information about the JMS specification, point your Web browser to **http://jsp.java.sun.com/products/jms/**.

JMS domains

JMS supports two messaging models, or *domains*: publish/subscribe and point-to-point. This section describes the differences between the JMS domains.

Publish/subscribe (pub/sub)

In a pub/sub message model, a consumer requests notification of particular events, such as changes in data values. In response to one request, called a *subscription*, a consumer might receive many messages.

The producer routes messages to a *topic* within the JMS provider, rather than to particular consumers. Each subscription provides a consumer access to one topic. Multiple consumers can subscribe to the same topic and each consumer can subscribe to multiple topics. The JMS provider routes the messages for each topic to the appropriate subscribers.

Point-to-point (PTP)

In a PTP model, a consumer expects a specific piece of information from a specific producer. PTP message services route messages to *queues* rather than to topics. A PTP consumer does not subscribe to a queue. Instead a consumer and producer agree together about the content and purpose of the messages on a queue.

Differences

In addition to termonology such as subscribe, topic, and queue, semantic subtlties distinguish between the two domains. For example, class names and minor features differ. The underlying differences accommodate implementers who use JMS interfaces to invoke vendor-specific legacy messaging systems that require these distinctions. Despite the differences in terminology, the pub/sub and PTP models function similarly.

Selectors

A *message selector* is a string that specifies which messages a consumer wants from a topic or queue. A selector contains attribute-value expressions that are compared with matching attributes in the message header. For example, if the producer includes a `priority` value in each message header, the subscription can specify the message selector string `priority >= '7'` to indicate that the consumer wants only messages of priority 7 or higher.

Luna Server JMS overview

The JMS specification addresses a wide range of objectives. Luna applies JMS to achieve the following two objectives:

- Send notices to external systems when a *Luna Server* application creates, updates, or deletes entity Beans.
- Receive notices from external systems to update entity Beans in *Luna Server*.

Luna messaging architecture and data flow

The *Luna Server* interfaces with a JMS provider through which server-side applications exchange messages with external message services. An object called a *message data source* encapsulates a configuration of a single topic or queue.

A message data source that receives messages includes a *message handler.* Each Luna message handler includes a JMS-specified *message listener* that listens for messages from the JMS provider.

In addition to listening for messages, the message handler decodes incoming messages and inserts the data from each message into cache. The default message handler expects a text message that contains Luna XML in the message body.

Figure 1 shows an example in which messages flow into a *Luna Server*.

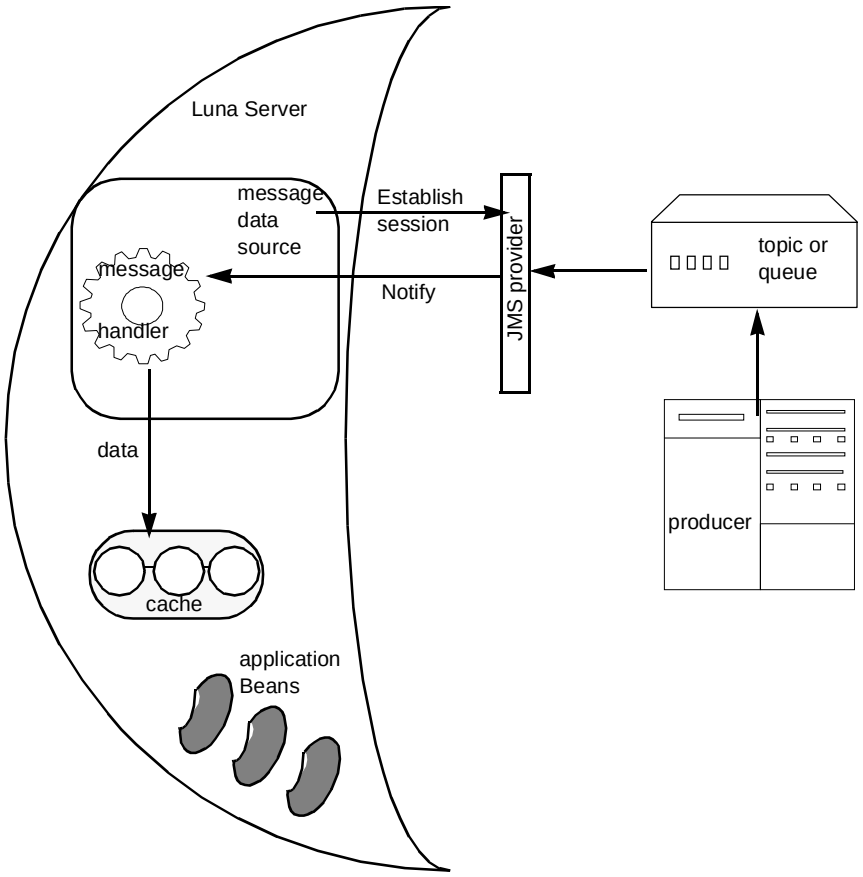

Figure 1. *Luna Server* consuming messages

If a *Luna Server* application sends information, a *message builder* creates a message from results of operations that the application completes. Figure 2 shows an example in which messages flow out of a *Luna Server* each time the Bank of Luna application creates a new Trade entity Bean.

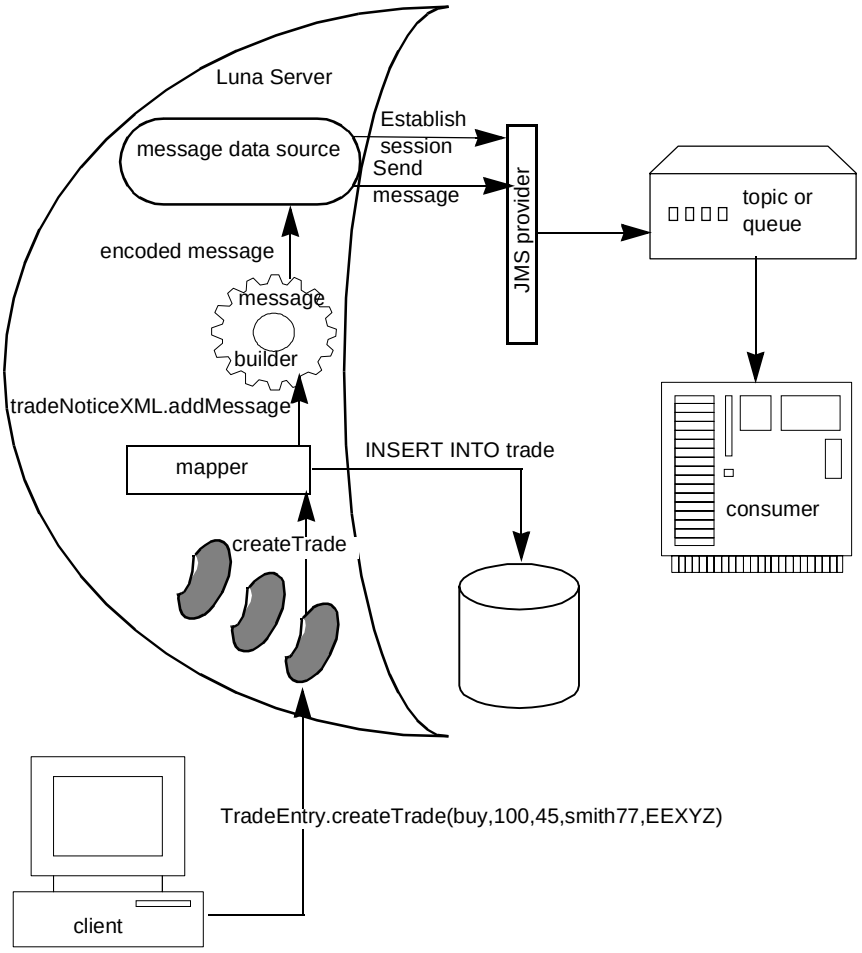

Figure 2. *Luna Server* creating and sending messages

For more information about Luna XML syntax, refer to Chapter 1, "*Luna XML*." To send messages that do not contain Luna XML, refer to *"User-defined message classes"* on page 24.

Application messaging requirements

To set up messaging for a particular *Luna Server* application, a developer performs one or both of the following tasks in the *Luna Development Center*:

- Add and configure one message data source for each topic or queue.
- Add message mappings that map operations and relationship operations to destination message data sources.

For the purposes of *Luna Server* applications, the provider or consumer that your Luna messaging connects determines which model you select for an application—pub/sub or PTP. When you configure a message data source, you identify the JNDI factory for a particular named topic or queue. To change message services any time during the life of the application without changing server or client code, simply use the *Luna Development Center* to change the message data source configuration .

Any topic or queue can receive, send, or both receive and send messages. To indicate that a message data source can receive messages, insert a `handler` property in the message data source configuration. To indicate that a message data source can send messages, insert it as a *destination* for an operation in a message mapping.

Because you can set up multiple message data sources for a single application, one application can:

- act as both consumer and producer.
- use both pub/sub and PTP messaging.

For an example of a single application that receives messages from a topic and sends messages to a queue, refer to *"Example: Bank of Luna messaging"* on page 8.

For a detailed description of messaging setup, refer to *"Developer configuration and mapping tasks"* on page 13.

Example: Bank of Luna messaging

The examples in this section describe the following messaging scenarios for the Bank of Luna project:

- Subscribing to a stock feed
- Sending point-to-point trade notices

Subscribing to a stock feed

As a typical example of a pub/sub consumer, consider how Bank of Luna might use an online *stock feed*. Think of a stock feed as an Internet ticker tape.

Bank of Luna maintains the records of customer holdings in a relational database. However, the bank cannot maintain current stock prices in that database because stock trades conducted outside of the bank dynamically change stock prices.

Bank of Luna wants to receive messages from an information producer that monitors stock trades and posts trade notices to topics. In this example, the producer publishes a message at the conclusion of major technology-company stock trades to a NASDAQ topic. Each message specifies the stock symbol and the unit value that results from the trade. To have current technology stock prices available for a customer, Bank of Luna subscribes to the NASDAQ topic.

Figure 1 on page 5 shows the flow of messages from the topic to the Bank of Luna application. When the Bank of Luna message handler receives a group of messages from the NASDAQ topic, the message handler might take the following actions:

1. Disassemble the message into separate stock-price pairs
2. Update cache with the stock symbol and trade price

When the Bank of Luna application needs an equity price, it searches for a cached Equity Bean with the appropriate stock symbol. If the equity is traded by an external broker, the message handler assures that cache reflects the latest price.

Sending point-to-point trade notices

Each time a Bank of Luna customer asks the bank to buy or sell equities, the Bank sends a notice that describes the trade to a centralized stock-market mainframe. In its notification message, the bank includes the stock symbol, quantity, price, and the buy or sell indicator. Figure 2 on page 6 shows the flow of data that creates the trade and transforms the trade to a queued message.

A Bank of Luna developer adds and configures a new message data source, named `OUTq`, in the *Luna Development Center*. As part of the `OUTq` configuration, the developer sets *`handler`* to null because `OUTq` represents a queue that sends, but does not receive messages.

To enable `OUTq` to send messages, the developer also creates a message mapping object, named `MM1`, that does the following:

- Maps `OUTq` as a destination for the CREATE operation
- Assigns a user-defined class named `TradeNoticeXML` as the `OUTq` message builder for a CREATE operation
- Associates Bank of Luna Trade entity Bean with `MM1`

Figure 3 shows how the *Developer Center Explorer* represents the association between the `MM1` message mapping and the Trade Bean.

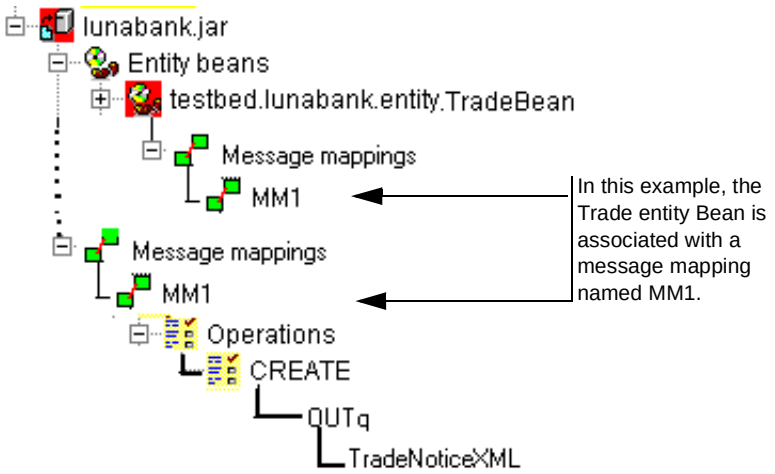

Figure 3. Associated Bean and message mapping

When the Bank of Luna creates a new trade, the *Luna Server* takes the following actions:

1. Finds the `MM1` message mapping.
2. Calls `TradeNoticeXML.addMessage` to add the operation output to the outgoing message.
3. Calls the `TradeNoticeXML.ReadyToSend` method when the last trade is created for this transaction
4. Uses the `OUTq` message data source connection to send the message

For more information about the `addMessage` and `ReadyToSend` methods, refer to *"MessageBuilder interface"* on page 26.

Luna messaging details

This section adds more detailed information to the *"Luna Server JMS overview"* on page 3.

Features and functionality

Luna generates the same message data source class for both topics and queues. The Luna-generated message data source supports the following features:

- Asynchronous communication

  To receive messages, a message data source must contain a message handler with its embedded listener object. The message handler receives messages asychronously, as the JMS provider forwards them. Luna messaging does not include synchronous communication.

- Client acknowledgment

  When the default message handler successfully decodes an incoming message, it sends an explicit acknowledgement of message receipt to the JMS provider. A disruption, such as a problem or quiescent state, prevents the handler from acknowledging a message, so the provider resends unacknowledged messages when a message data source connects to a topic or queue.

- Automatic transaction management

  All sessions automatically start in transaction mode with a `localBegin` method. A `localCommit` or `localRollback` causes the session to release its message consumer or message producer resources.

- Mapping of messages to operations

  Luna associates message creation with operations. A destination topic or queue can assemble messages from transactions that involve multiple beans. Messages remain independent of Beans and can be reused. For more information about mapping message data sources, refer to *"Mapping operations to*

outgoing messages" on page 19.

- No recoding of application

  A simple, graphical interface relieves the developer from coding messaging details. For details, refer to *"Developer configuration and mapping tasks"* on page 13.

Message flow

For incoming messages, the following actions occur:

1. The message data source establishes a connection and session for a queue or topic.
2. The JMS provider sends a message to message handler.
3. The message handler converts the message to data and operations for an application entity Bean or collection.
4. The message handler places the data from the incoming message in cache.
5. *Luna Server* applies the message data to the appropriate entity Bean or collection.
6. When a transaction commits, new values from the message are written to the data base.

For outgoing messages, the following actions occur:

1. An application performs an operation on an entity Bean that is associated with a message mapping.

   For detailed information about the contents of a message mapping, refer to *"Mapping operations to outgoing messages"* on page 19.
2. The message data source establishes a connection and session for the application.
3. The message builder formats the updated data appropriately for message recipient and creates the outgoing message.
4. The application commits the transaction that includes message-mapped operations.

5. The mapping layer routes messages that result from committed operations to the appropriate message data sources.
6. Each mapped message data source sends messages to its respective queue or topic.

Developer configuration and mapping tasks

This section describes the developer activities that enable an application to receive and send messages through a JMS provider.

Configuring a message data source

To add a new message data source:

1. Right click on `Message Data Sources` in the *Luna Development Center Explorer* window.
2. Select **Add New Message Data Source** from the popup menu.
3. The *JMS Source Wizard* starts. Use the wizard to configure the message data source.

Using the wizard

This section describes how to fill in the wizard dialogs.

The wizard begins with the dialog in Figure 4. Fill in the user name, password, and agreement with which you log into *Luna Server*.

If you do not fill in a user name or agreement, you log in as anonymous.

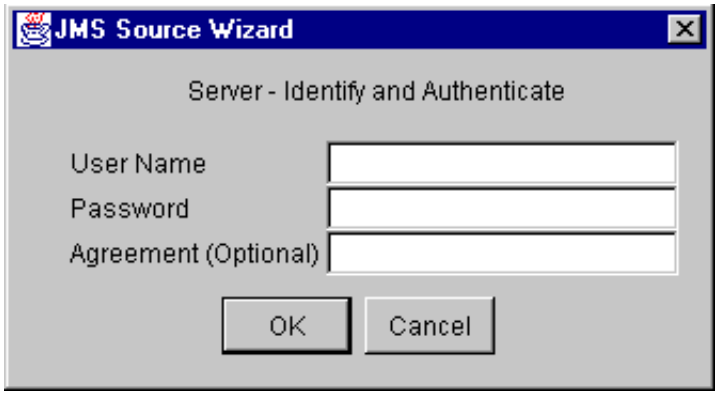

Figure 4. Server - Identify and Authenticate

Luna Server checks the user information and issues the message in Figure 5. Click the **OK** button to continue.

Figure 5. Authentication verification message

The wizard then displays the dialog in Figure 6, which is the only information that distinguishes the message data source as PTP or pub/sub.

For PTP messaging:

- Select the **Queue** radio button
- Fill in the JNDI lookup for an initial queue connection factory.
- Fill in the JNDI lookup for queue name.

For pub/sub messaging

- Select the **Topic** radio button.
- Fill in the JNDI lookup for an initial topic connection factory.
- FIll in the JNDI lookpu for topic name.

The wizard uses the connection factory and queue or topic name to assemble the *ConnectionURL* in the **URL** field. Figure 6 shows an example for a queue.

Click the **Next** > button to continue.

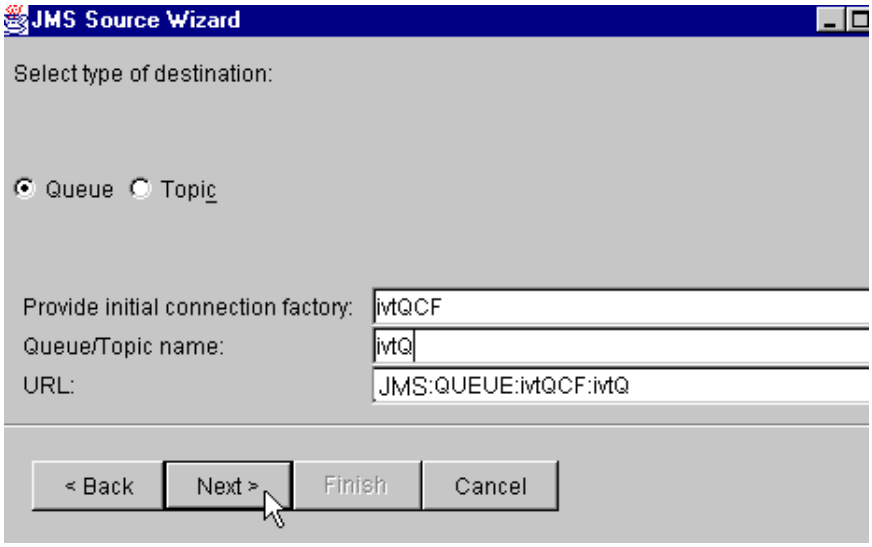

Figure 6. Select type of destination

In Figure 7, fill in each value of the *ConnectionProperty* string that this message data source requires. For *ConnectionProperty* details, refer to Table 1 on page 18.

Click the **Next >** button to continue.

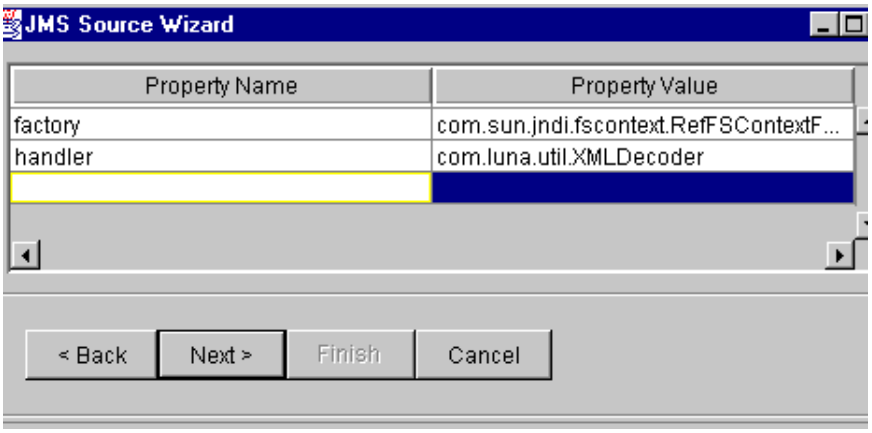

Figure 7. ConnectionProperties

In Figure 8, the wizard again displays the *`ConnectionURL`*. Fill in the user and password that identifies the message data source to the JMS provider queue.

Click the **Finish** button.

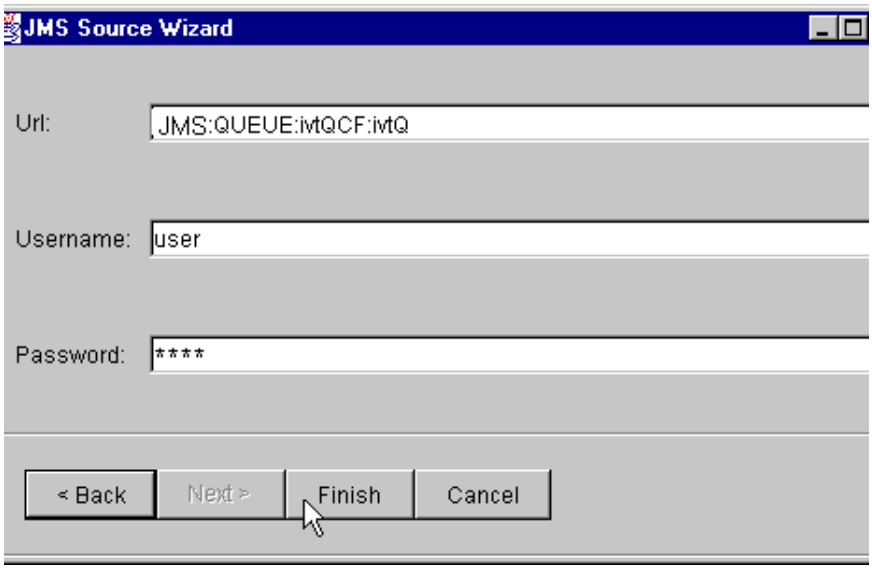

Figure 8. Queue or topic Authentication

Note! You can only use the *JMS Source Wizard* one time per message data source, when you first add a new message data source. To change the settings that you made in the wizard, use the *Inspector* window.

Using the *Inspector*

After you use the *JMS Source Wizard* to set up a new message data source, the *Inspector* window shows the following configuration properties:

Table 1. Message Data Source configuration properties.

Configuration Setting	Description

ConnectionURL	Required JNDI lookup information to establish a connection and session. Use the following format:
	`JMS:[TOPIC | QUEUE]:`
	`connectionFactory_lookup``:`
	`[`*`topic_name`* `|` *`queue_name`*`]`

UserName	Required user name that authorizes use of a queue or topic. *Luna Server* uses this name to connect to the topic or queue.
Password	Required password that completes the *UserName* authorization informaton for the queue or topic. [default: `<not visible>`]
Connection Properties	A string that contains the following editable properties: • *icf* - (Required.) Initial JNDI context factory class that creates a JNDI object in your application space. [default: `com.sun.jndi.fscontext.RefFSContext`] • *provider* - (Required.) The directory where you place the initial context factory. • *handler* - Name of the class that interprets incoming messages. Required for receive and null for send-only. [default: `LunaMessageHandler`] To implement or extend `com.luna.access.MessageHandler`, refer to *"MessageHandler interface"* on page 25. • *selector* - An optional expression, similar to the filter criteria in an SQL WHERE clause, that tells the JMS provider which subset of notices from a topic or queue the message data source should receive. Enclose the expression in quotes.

To change a value in the *Inspector* window, such as `ConnectionProperties`, click in the corresponding Value field for that Property, as Figure 9 shows. .

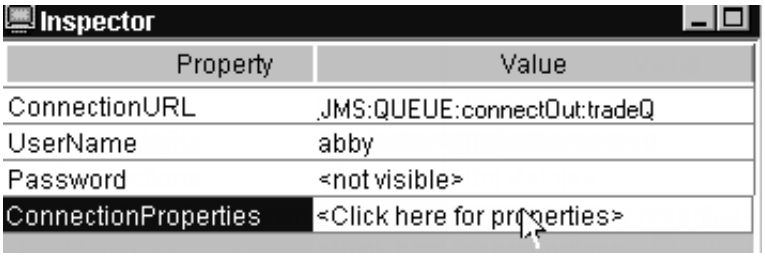

Figure 9. Editing a message data source property.

Mapping operations to outgoing messages

This section pertains only to applications that send messages.

In addition to adding and configuring a message data source, the developer must map the operations in the application to create messages. Using the *Luna Development Center*, the developer performs the following setup tasks:

- In the *Explorer* window, add a named *message mapping* object to the project.
- Use the *Message Mapping Tool* in the *Workspace* to fill in the Java Message Descriptor (JMD) for a message mapping.

  The JMD assigns the following properties to an individual message mapping:

  - Operations that a message describes
  - Destinations (message data sources) that send messages
  - Message-builder class that formats the message header and body for a particular operation-destination pair.

  For detailed information about JMDs, refer to *"Message Descriptors"* on page 20.

You associate a message mapping with one or more entity Beans. Figure 3 on page 10 shows an example of how the *Explorer* window looks after you associate a Bean with a message mapping.

Note! Luna separates message mapping objects from entity Beans so that you can map multiple entity Beans to the same message mapping.

There are two ways to create an association:

- Select the Bean in the *Explorer* window and the *JMS Map* tab in the Workplace.
- Select the message mapping in the *Explorer*, which displays the *Workspace* view that Figure 11 on page 23 shows. Click the **Create Association** button.

Message Descriptors

Each message mapping represents a Java Message Descriptor (JMD). Each JMD associates operations with a destination queue or topic and message-builder class, as Figure 10 shows.

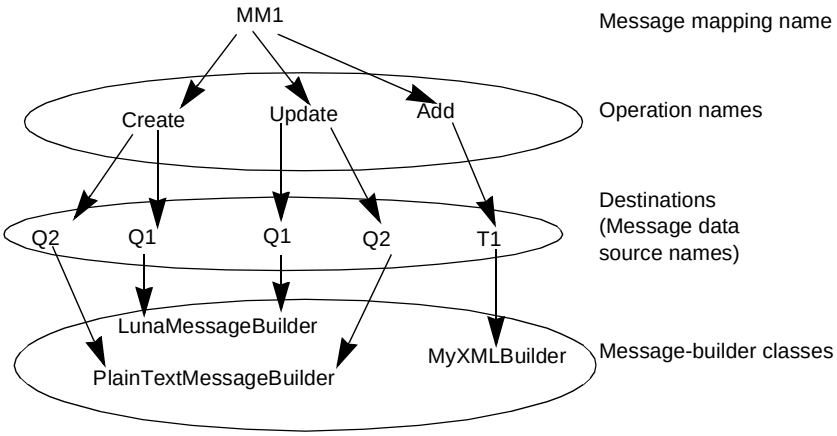

Figure 10. JMD Hierarchy

A single message mapping descriptor can include one or more *operation descriptors* as the following table shows.

Message Mapping Descriptor (JMD)	
Detail	**Description**
Name	A unique String value used to associate Beans with this mapping.
Operations	List of operation names that this JMD describes.

A single operation descriptor can route a message to one or more message data sources, as the following table shows.

Message Operation Descriptor (JMDOperationDescriptor)	
Detail	**Description**
OperationName	One of the following: • Entity Bean operation `DELETE`, `UPDATE`, or `CREATE` • Relation operation `LOAD`, `REMOVE`, or `ADD` • User-defined operation
Destinations	List of message data source names that send messages as a result of this operation.

The following table shows the information that you specify for each message data source that sends operation results in messages.

Destination Descriptor (JMDDestination)	
Detail	**Description**
`DataSourceName`	The name of the message data source that connects to the destination topic or queue.
`MessageBuilder-Class`	The class that inserts operation results in a message. [default: `LunaMessageBuilder`] See *"Message-Builder interface"* on page 26.

To create the JMD and its subclasses, JMDOperation and JMDDestination descriptors, use the *Message Mapping Tool*.

Message Mapping Tool

The *Message Mapping Tool* allows you to do the following:

- Add, remove, or alter a mapping between an operation and a destination message data source.
- Create or remove the association between an entity Bean and the selected Message Mapping object.

To display the *Message Mapping Tool*, select a message mapping name in the *Explorer* window.

The dialog examples in this section build the JMD hierarchy from Figure 10 on page 20. Figure 11 shows the *Workspace* window with the *Message Mapping Tool* filled in for the example message mapping, named `MM1`, from Figure 10.

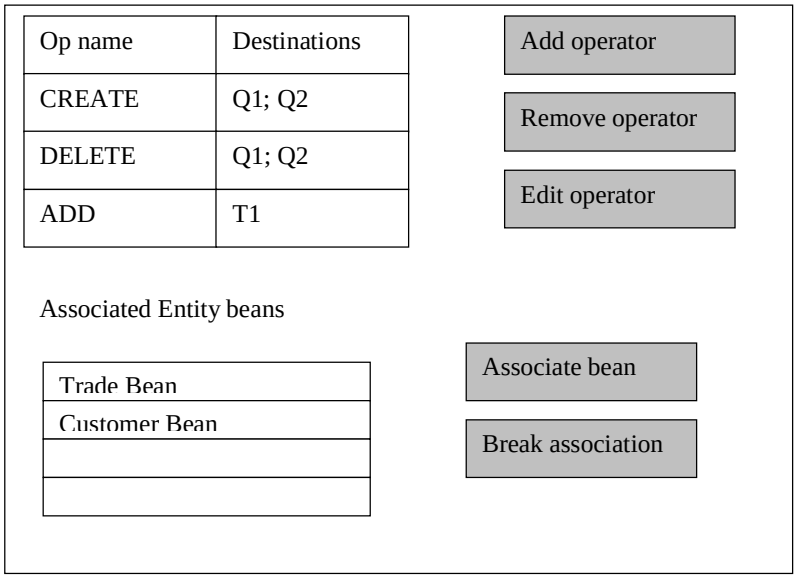

Figure 11. Message mapping tool in Workspace

When you select the **Add operator** button, the Workspace displays the dialog in Figure 12.

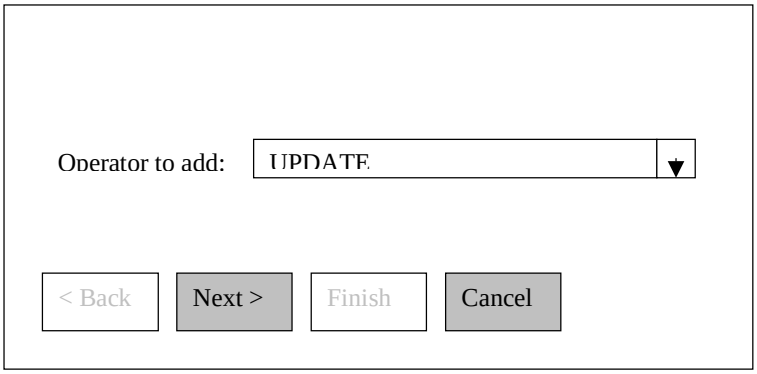

Figure 12. Add operator (JMDOperation)

When you select the **Next >** button in Figure 12, the *Workspace* displays the dialog in Figure 13. In this dialog, you specify each destination that sends a message for the operation in Figure 12 and the message-builder class that creates the message.

Click the **Finish** button after you add all the destinations for this operation and the *Workspace* returns to Figure 11 on page 23.

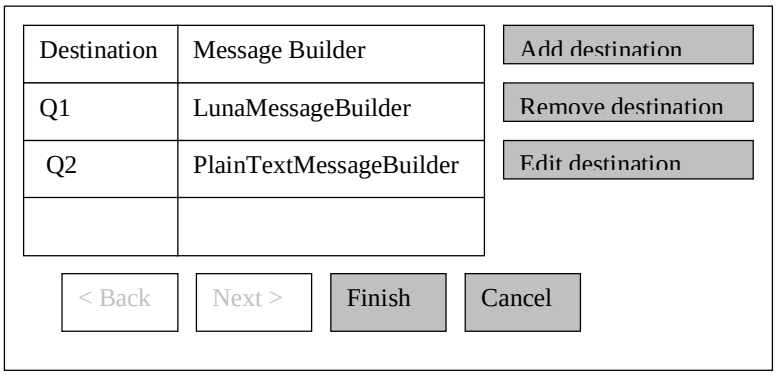

Figure 13. JMDDestination

For detailed instructions in using the *Message Mapping Tool*, refer to the *Luna Development Center* online help.

User-defined message classes

By default, a message data source expects incoming messages to contain Luna XML and creates outgoing messages that contain Luna XML. The Luna XML vocabulary provides script-like element to call methods and return results. For a detailed description of the Luna XML vocabulary, refer to Chapter 1, "*Luna XML*."

Luna provides interfaces and default implementation classes to enable a message data source to interpret incoming messages or build outgoing messages. As a *Luna Server* developer, you can extend the default classes or implement the interfaces that this section describes.

MessageHandler interface

Implement the `MessageHandler` interface to transform incoming messages into application data.

The `MessageHandler` interface has one method:

```
void processMessage(javax.jms.Message)
```

The parameter contains a reference to an incoming message. The body of the message might contain any of the subtypes that the JMS specification defines for a `Message`. Typical messages contain text or name-value pairs, though JMS also permits a stream of java primitives, a serializable java object, or even uninterpreted bytes.

For example, a message might contain:

- Data records from a legacy system
- Client-specific XML
- An XML vocabulary that is maintained by an industry consortium, such as Financial Information eXchange protocol (FIX), or RosettaNet

  For information about XML consortiums, point your browser to the XML Catalog at the following URL:

  http://www.xml.org/xmlorg_registry/index.shtml

The default `MessageHandler` implementation, `LunaMessageHandler`, expects a String of Luna XML. This default class executes the incoming Luna XML script and stores the results in cache.

MessageBuilder interface

The default `LunaMessageBuilder` class, inserts Luna XML in the body of the outgoing message. The recipient of the message might, in turn, transform the Luna XML in the body of the message to a different format. A user-defined message handler class can extend the default class to provide additional functionality.

As an alternative to sending Luna XML, create a user-defined class that implements the `MessageBuilder` interface to assemble message a different type of message. For example, you implement your own message-builder class to build format the message body that contains one of the following:

- A different XML vocabulary
- Plain text,
- Any of the non-text message body data types that the JMS specification allows in a javax.jms.Message object. (See

To create a Luna XML message that has additional features, extend the `LunaMessageBuilder` class. For example, you can extend the message-builder to insert attributes in the message header that allow a consumer to specify in a selector. (See *"Selectors"* on page 3).

You can provide several `MessageBuilder` implementations, each of which creates an operation-specific message. If various operations can result in the same message format, you can use the same `MessageBuilder` implementation for multiple operations.

If you either implement a new `MessageBuilder` class or extend the default, `LunaMessageBuilder`, you must use the *Developer Center Workspace* to specify the user-defined class when you map the operation, as *"Mapping operations to outgoing messages"* on page 19 describes.

The `MessageBuilder` interface specifies three methods—`createMessage`, `addMessage`, and `ReadyToSend`.

MessageBuilder.createMessage

The first message in a transaction context calls `createMessage` with the following signature:

```
void createMessage(LunaXAConnection)
```

If *`isConcatinate=true`*, this method perform the tasks required for a `localBegin`.

The default Luna implementation creates an empty text message.

MessageBuilder.addMessage

Each operation results in a call to `addMessage`, which has the following signature:

```
addMessage(javax.jms.Message, MessageParams)
```

javax.jms.Message

The first parameter contains the results from the operation that the message builder formats and inserts in the body of the message. A javax.jms.Message object can contain any of the following in the body of a message:

- A stream of Java primitive values
- A set of name-value pairs where values are Java primitive types
- A java.util.String
- A stream of uninterpreted bytes

If *`isConcatinate`*`=true`, the first operation in a transaction context starts the message body and each subsequent call to `addMessage` appends the content that it formats to the message body. If *`isConcatinate`*`=false`, the message body describes only one operation.

The default implementation creates a String that consists of Luna XML instructions for the operation. Typically, *`isConcatinate`*`=true` and the final message body contains a

sequence of Luna XML elements, such as `<BeanReferenceTag>` and `<MethodCallTag>` elements, to describe the operations in the current transaction.

MessageParams

The `MessageParams` parameter contains the name of the operation and the entity Bean that is operated on.

The `MessageParams` object has the following subclasses:

- `RelationshipMessageParams` is added to `MessageParams` for a relationship operation and references the entity Bean that the operator adds to, removes from, or loads into a cached collection.

  For example, if the Bank of Luna application adds a trade to a customer portfolio, the `RelationshipMessageParams` references the added Trade Bean.

- `UpdateMessageParams` is added to `MessageParams` for an UPDATE operation and specifies the fields that are updated in the entity Bean.

The default `LunaMessageBuilder` uses `MessageParams` and its subclasses to optimize the contents of a Luna XML message.

MessageBuilder.ReadyToSend

The `readyToSend` method follows the `addMessage` method for the last operation in the transaction context and has the following signature:

```
void readyToSend(javax.jms.Message msg)
```

This method might include behavior that reflects the results of the transaction. For example:

- If `localRollback` ends the transaction, cancel the message or assemble and send a message that notifies the recipient of the rollback.

- If `localCommit` ends the transaction, add any message text that indicates a successful transaction then send the completed message to the topic or queue.

The default implementation wraps the concatenated message in a pair of Luna XML `<TransactionTag>` `</TransactionTag>` elements.